Persistance et disponibilite dans les systemes repartis : application a Guide

Pierre-Yves Chevalier

THESE

présentée par

Pierre−Yves Chevalier

pour obtenir le titre de

Docteur de l'Université Joseph Fourier − Grenoble I

(arrêté ministériel du 23 novembre 1988)

Spécialité : Informatique

Persistance et disponibilité dans les systèmes répartis : application à Guide

Thèse Soutenue devant la commission d'examen le : 14 Octobre 1994

J.−P. Verjus	Président
X. Rousset de Pina	Directeur de thèse
M. Banâtre	Rapporteur
P. Valduriez	Rapporteur
G. Bogo	Examinateur
A. Giacalone	Examinateur
S. Krakowiak	Examinateur

Thèse préparée au sein du Laboratoire Bull−IMAG/Systèmes

Je tiens à remercier ici l'ensemble des personnes qui, par leurs conseils, leurs remarques et leur encouragements, ont contribué à l'aboutissement de ce travail :

Jean−Pierre Verjus, Professeur à l'Institut National Polytechnique de Grenoble (ENSIMAG), pour m'avoir fait l'honneur d'accepter de présider le jury de cette thèse.

Michel Banâtre, Directeur de Recherche à l'Institut Recherche en Informatique et Systèmes Aléatoires de Rennes, et Patrick Valduriez, Directeur de Recherche à l'Institut National de la Recherche en Informatique et Automatique de Rocquencourt, qui ont accepté de consacrer une partie de leur temps pour juger ce travail.

Xavier Rousset de Pina, Professeur à l'Institut National Polytechnique de Grenoble (ENSERG), mon directeur de thèse, pour m'avoir fait confiance depuis le début et pour les innombrables discussions que nous avons eues. Sa rigueur intellectuelle et morale est pour moi plus qu'un exemple à suivre, un modèle.

Sacha Krakowiak et Roland Balter, Professeurs à l'Université Joseph Fourier et directeurs de l'Unité Mixte Bull−IMAG/Systèmes, pour l'honneur qu'ils m'ont fait en m'accueillant dans le projet Guide.

Gilles Bogo, Directeur de l'équipe DOM du centre de recherche Bull à Gières, qui m'a fait l'honneur d'accepter de faire partie du jury et dont la compagnie a financé mon travail.

Alessandro Giacalone, Directeur de l'European Computer−Industry Research Centre à Munich, qui m'a fait l'honneur d'accepter de faire partie du jury et qui m'a permis de terminer la rédaction de cette thèse à mon arrivée dans son équipe.

L'ensemble des participants au projet Guide et plus particulièrement les membres fondateurs de l'Eliott Task Force à savoir : Jacques Cayuela, André Freyssinet, Serge Lacourte et Daniel Hagimont, qui ont permis à ce petit dragon vert de s'envoler au dessus de contrées hostiles et lointaines.

Philippe Bernadat et Eric Paire, ingénieurs à l'Open Software Foundation, pour leur assistance et, surtout, pour avoir eu la patience et la gentillesse de répondre à mes nombreuses questions sur Mach.

Pascal Dechamboux, Stéphane Bressan, Philippe Bonnet et Francis Kaner pour leur aide et les nombreuses discussions que nous avons eues et qui visaient à comparer et rapprocher nos champs de recherche respectifs.

Je tiens à remercier spécialement l'ensemble du personnel de Bull−IMAG pour l'ambiance exceptionnelle et le cadre de travail que j'y ai trouvé. La sympathie et l'amitié qui ont régné entre nous ne sont pas étrangères aux résultats d'ensemble du laboratoire.

Résumé :

L’objectif de cette thèse est d’étudier les mécanismes que doit offrir une plate−forme répartie pour permettre l’implantation efficace des objets persistants, et d’appliquer les résultats de cette étude au système Guide. Ce travail s’est plus particulièrement inscrit dans la phase de conception et de réalisation de la seconde version de ce système, appelée Guide−2. Dans ce cadre, un accent particulier a porté sur la modularité de la réalisation, sur l’ouverture de la plate−forme, sur l’utilisation des nouvelles architectures de systèmes telle que la technologie micro−noyau, ainsi que sur la fiabilité générale du système. Notre proposition intègre de manière cohérente et efficace un ensemble de mécanismes nécessaire à la mise en œuvre de la persistance et assurant la fiabilité de ce support en présence de panne.

The goal of this thesis is to study the underlying mechanism that must be provided by a plateform to efficiently support distributed persistent objects. This work was conducted for the design and the development of the second version of the Guide system. In this framework, the main concerns was to define a modular and open architecture, to improve the overall reliability of the system and to make use of advanced micro−kernel technology. Our proposal coherently integrates a set of basic mechanisms used to provide persistency in the Guide distributed system while enhancing the availability of this support in face of failures.

Chapitre I

Introduction

Chapitre II

Support de la persistance

Chapitre III

L'environnement Guide

Chapitre IV

Le support de la persistance dans Guide

Chapitre V

Evaluation du support de la persistance

Chapitre VI

Vers une fiabilité accrue

Chapitre VII

Pour un stockage fiable dans le système Guide−2

Chapitre VIII

Conclusion

Table des figures

Chapitre I

Introduction

Les applications logicielles de grande taille, telles que les environnements de génie logiciel ou les applications de production de documents structurés, ont besoin de modèles de données avancés. En effet, les données qu'elles utilisent doivent être organisées en structures complexes et sont de natures diverses (texte, image, programme, etc). Le cycle de vie de ces données peut être long et complexe. En conséquence, les données gérées par ce type d'applications doivent être persistantes(1). C'est à dire qu'elles doivent être conservées sur un ou des supports permanents de stockage de manière à résister à l'arrêt des machines sur lesquelles les applications s'exécutent. Par ailleurs, ces applications requièrent l'interaction et la coopération de plusieurs utilisateurs ; cela entraîne la possibilité d'exécutions concurrentes, des communications fréquentes et un haut degré de partage.

Pour réduire les coûts de développement, des environnements (langages et outils) permettant la réutilisation du code, une maintenance aisée et assurant son évolution, sont alors nécessaires. Les langages à objets ont des propriétés qui répondent pour l'essentiel à ces besoins : le modèle objet permet de représenter la complexité des structures, le mécanisme de spécialisation par sous−typage permet la réutilisation du code, l'héritage et la modularité facilitent la maintenance et l'évolution du code. Pour fournir, de manière homogène, les modèles d'exécution nécessaires, ces langages ont été étendus à la répartition, la persistance, le parallélisme et la synchronisation.

Avec de tels langages, se pose alors le problème de définir une plate−forme qui puisse les supporter de manière efficace. Cette démarche a guidé un grand nombre de projets dans le domaine des systèmes d'exploitation répartis (Argus [Liskov 85], Clouds [Dasgupta 90], Emerald [Black 86], Gothic [Banâtre 92] et Guide [Balter 91]) et des bases de données (Casper [Vaughan 92], Cricket [Shekita 90], Eos [Gruber 92] et O_2 [Deux 91]).

L'objectif de cette thèse est d'étudier les mécanismes que doit offrir une plate−forme répartie pour permettre l'implantation efficace des objets persistants, et d'appliquer les résultats de cette étude au système Guide.

---

(1) Par abus de langage, nous assimilons la persistance à la permanence, c'est à dire à la conservation fiable des données sur un support de stockage.

Notre travail s'est plus particulièrement inscrit dans la phase de conception et de réalisation de la seconde version de ce système, appelée Guide−2. Dans ce cadre, un accent particulier a porté sur la modularité de la réalisation, sur l'ouverture de la plate−forme, sur l'utilisation des nouvelles architectures de systèmes telle que la technologie micro−noyau, ainsi que sur la fiabilité générale du système.

Ces contraintes et ces objectifs nous ont conduit à privilégier le caractère générique du système que nous avons défini, réalisé et évalué. Notre proposition intègre de manière cohérente et efficace un ensemble de mécanismes nécessaire à la mise en œuvre de la persistance et assurant la fiabilité de ce support en présence de panne.

I.1 Support de la persistance

La notion de ***persistance*** est fournie dans les langages de programmation pour unifier l'utilisation des données persistantes et temporaires [Atkinson 87]. Elle a été introduite d'abord dans les langages fonctionnels et notamment LISP, puis dans les langages de programmation de bases de données, tels que Pascal/R [Schmidt 77] et PS−Algol [Atkinson 83].

Il existe plusieurs manières de rendre visible la persistance au niveau des langages de programmation : dans les langages Trellis/Owl et Eiffel, elle est fournie comme un trait rajouté du langage d'origine ; dans le langage Guide, un objet est persistant dès lors qu'il existe un chemin d'accès menant à lui depuis un des objets appelés racines de persistance. On peut remarquer que cette technique est actuellement l'une des plus reconnue.

Dans tous les cas de figure, l'implantation de ce type de langages nécessite l'utilisation d'un support de la persistance. Ce support peut être basé soit sur le système de fichiers fourni par le système d'exploitation, soit sur un système de gestion de bases de données, soit sur un système de gestion d'objets persistants [Cockshot 84].

Ces systèmes de gestion associent de façon très fine la notion de persistance à celle d'objets. Le système offre ainsi des fonctions de stockage de haut niveau sémantique, c'est à dire dans lesquelles la notion d'objet est connue, avec comme corollaire la prise en compte au niveau du système de stockage des problèmes de désignation et de partage relatifs aux objets. L'objet est alors l'unité de persistance et de partage [Hosking 93], ce qui permet de coller idéalement au langage.

L'inconvénient majeur de cette solution est qu'elle impose l'objet comme unité de transfert entre la mémoire de travail et celle de stockage. Or l'expérience, acquise avec le langage Guide [Freyssinet 91], a montré que la taille moyenne des objets était inférieure à 500 octets, ce qui est petit relativement à la taille des pages (en général 4 Ko). De plus, un chargement objet par objet risque de mal convenir aux applications visées qui utilisent couramment des milliers sinon des millions d'objets

[Cahill 93]. Le coût d'une lecture objet par objet deviendrait vite prohibitif. C'est pourquoi, des études sur le regroupement d'objets dans des conteneurs, de façon à réduire le nombre des transferts entre la mémoire et le disque ont été menées [Benzaken 90][Deppish 91]. Des solutions basées sur le couplage de l'espace de stockage dans la mémoire virtuelle des processus ont été proposées [Millard 93] [Shekita 90]. Le gros avantage de ce type de solution, est qu'il permet de découpler l'unité d'accès à l'information de l'unité de transfert entre la mémoire et les disques. On accède aux objets de manière uniforme et on transfère des pages. On peut ainsi clairement séparer la gestion de l'accès aux objets de celle de leur persistance.

La solution que nous avons choisie pour le support des objets persistants des langages Guide et d'une version étendue de C++ intégrant la persistance et la distribution, s'appuie sur la définition d'une machine de stockage de bas niveau sémantique [Chevalier 94c]. L'approche retenue est celle du conteneur de stockage non typé et non structuré, appelé ***grappe***. La sémantique interne d'une grappe est définie par une couche supérieure du système : la machine à objets qui gère l'accès aux objets (synchronisation et protection) [Hagimont 93]. L'accès à une grappe est réalisé par son couplage dans l'espace d'adressage des applications. La machine à grappe est mise en œuvre sur la machine de stockage pour laquelle la grappe est une simple suite de blocs.

On peut caractériser notre proposition du support de la persistance pour les langages à objets par deux points forts :

- Un modèle de mémoire uniforme structuré autour de la notion de grappe, qui offre un support efficace pour le développement de la machine qui gère les objets.
- Une mise en œuvre répartie de ce support (par des gérants de mémoire(2)) qui permet de partager la charge associée entre un ensemble de machines.

Le mécanisme de couplage, fourni par la machine à grappes, permet d'offrir à la machine à objets, une mémoire de grappes uniforme au sens de Multics [Organick 72]. La machine de stockage fournit une interface suffisamment générale pour permettre la mise en œuvre du stockage des grappes par utilisation de serveurs de stockage externes à la plate−forme Guide.

I.2 Stockage fiable

La fiabilité du stockage revêt une importance capitale pour le système Guide. En effet, dès lors qu'un grand nombre de stations participent à la mise en œuvre des services fournis par la plate−forme, il devient très probable qu'une au moins soit en panne ou tout simplement arrêtée. Or, le fait d'avoir une ou plusieurs machines non présentes, va interdire, si on n'y prend pas garde, l'accès à un sous ensemble des

(2) Un gérant de mémoire a un rôle similaire à celui d'un serveur de page dans une base de données.

objets persistants utilisés par les applications et rendre par là même, un grand nombre d'applications indisponibles.

Afin de réduire la probabilité qu'un ensemble d'objets demeurent inaccessibles à la suite d'une panne de machine ou du réseau, de nombreux travaux ont été mené ces dernières années visant à augmenter la ***disponibilité*** des données; c'est à dire la probabilité qu'un objet reste accessible en dépit des pannes du système [Laprie 89]. Les techniques généralement mises en œuvre sont :

- La construction de matériels fiables [Shrivastava 92], [Bartlett 81] et [Banâtre 91], garantissant la robustesse du stockage.
- La duplication logicielle des données sur un ensemble de supports de stockage indépendants [Satyanarayanan 93][Birrell 93].

Les systèmes basés sur des matériels ad hoc ont pour eux une excellente qualité de service tant du point de vue des performances que de la fiabilité générale. Cependant leur développement peut s'avérer très coûteux. De plus, l'importance des problèmes matériels à résoudre (comme l'écriture de pilotes disques) et l'utilisation de machines dédiées est un obstacle majeur aussi bien à leur évolution qu'à leur diffusion.

La duplication logicielle a l'avantage de ne pas nécessiter de matériels particuliers (ou dédiés). La mise en œuvre peut être réalisée en utilisant les outils de développement disponibles sur le marché : compilateurs, éditeur de liens, metteur au point, ainsi que les services de bas niveau fournis par les systèmes d'exploitation : gestion des entrées/sorties sur le disque ou sur le réseau notamment. Par ailleurs, des solutions purement logicielles offrent tout de même des performances acceptables.

C'est pourquoi, dans notre étude sur la fiabilité et la disponibilité du support de la persistance, nous nous sommes principalement intéressé à ce type de solutions.

Un grand nombre de protocoles de contrôle de la duplication ont été proposés, témoignant ainsi de l'intense activité de recherche qui règne dans ce domaine. Les champs d'applications couverts sont très vastes et incluent des domaines aussi différents que la transmission fiable de messages, le support transactionnel ou le stockage fiable. Les protocoles (ou algorithmes) proposés sont, généralement, classés en deux catégories :

- Les algorithmes pessimistes qui garantissent le maintien de la cohérence (stricte) des données en cas de panne de machines ou du réseau de communication.
- Les algorithmes optimistes qui peuvent relâcher la contrainte de cohérence (et donc laisser des copies diverger) afin d'accroître la disponibilité des données.

De manière générale, les algorithmes pessimistes, en privilégiant la cohérence des données, vont réduire leur disponibilité. Ils sont aussi basés dans la plupart des cas sur des algorithmes complexes qui peuvent s'avérer coûteux même en l'absence de pannes. Les algorithmes optimistes, eux, en privilégiant la disponibilité et les performances, ne seront pas capables de garantir la cohérence des données.

Notre approche se caractérise par la volonté de privilégier les performances en l'absence de panne tout en offrant des garanties de cohérence stricte des données en cas de panne (et notamment de partitionnement du réseau de communication). Nous avons donc défini un serveur de stockage fiable basé principalement sur les techniques suivantes :

- La mise à jour asynchrone des copies (de grappe) afin d'obtenir les meilleures performances possibles en l'absence de panne. Ce protocole est pessimiste et utilise une copie primaire.
- La ré−utilisation de l'information de journalisation gérée par le service de stockage afin de mettre en œuvre, de manière simple et efficace, la restauration d'un site de stockage.

I.3 Organisation du mémoire

Nous pouvons distinguer deux grandes parties dans ce document. Dans la première partie, couverte par les chapitres II, III, IV et V, nous présentons les principaux concepts et mécanismes introduits pour fournir la persistance dans les systèmes répartis et leur application au système réparti à objets Guide. La deuxième partie, couverte par les chapitres VI et VII, analyse les problèmes liés au stockage fiable de données et à l'intégration à la machine de stockage du système Guide d'un tel service.

Chapitre II

Ce chapitre s'efforce de décrire les principales caractéristiques d'un système support de la persistance. Nous commençons par définir le cadre conceptuel de notre étude qui est basé sur le modèle de mémoire persistante et uniforme de Thatte. Par la suite, nous proposons une étude de cas de différents systèmes selon quatre critères : la désignation, le partage, le stockage et la protection. Nous essayons de présenter les compromis nécessaires à l'intégration de ces quatre points dans les systèmes présentés.

Chapitre III

Ce chapitre présente le système réparti à objets Guide qui a servi de cible à notre étude. Nous décrivons principalement l'architecture générale du système ainsi que son organisation au dessus du micro−noyau Mach 3.0. Un

des aspects fondamentaux de ce système réside dans la définition d'un modèle de mémoire, uniforme pour l'utilisateur, structuré en deux niveaux : une mémoire d'exécution qui réalise l'espace où sont couplés dynamiquement les objets et une mémoire de stockage qui sert de support de persistance des objets.

Chapitre IV

Ce chapitre présente les principes de réalisation du support de la persistance dans le système Guide. La principale caractéristique de ce support est qu'il s'agit d'un support de très bas niveau sémantique s'appuyant sur le concept de grappes. Nous nous attachons particulièrement à décrire la mise en œuvre de notre architecture à l'aide de paginateurs externes. Lors de la conception de ce support, nous avons porté une attention particulière à la modularité et à l'ouverture du système.

Chapitre V

Ce chapitre propose une évaluation, tant qualitative que quantitative, du support de la persistance présenté au chapitre précédent. L'évaluation commence par une comparaison des fonctions offertes avec celles de systèmes similaires. Puis, nous présentons des résultats préliminaires obtenus à l'aide d'applications de démonstrations ainsi que des mesures de performances des fonctions de bas niveau de notre support. Ce chapitre se conclut par une brève évaluation du support fourni par le micro−noyau Mach 3.0 à la construction d'un tel système.

Chapitre VI

Ce chapitre contient une présentation des principaux concepts et algorithmes de gestion de la duplication et leur application au stockage fiable des données. Cette description est basée sur la présentation et l'évaluation de deux services de stockage fiables représentatifs des deux grandes classes de solutions proposées. La principale conclusion de cette étude est que le stockage fiable des données ne nécessite ni algorithmes complexes ni la gestion d'un grand nombre de copies d'objets.

Chapitre VII

Ce chapitre présente l'architecture d'un service de stockage fiable pour le système Guide. La caractéristique principale de ce service, est qu'il tire partie de l'architecture du support de la persistance afin de concilier au mieux cohérence, performance et disponibilité. Le compromis que nous avons sélectionné nous permet de fournir un service qui ne privilégie pas l'un de ces aspects au dépend des deux autres. Une brève évaluation du prototype actuellement disponible est proposée pour conclure ce chapitre.

Chapitre VIII

Ce chapitre de conclusion résume les principaux aspects abordés dans ce mémoire et met l'accent sur les apports du travail effectué. Nous présentons ensuite notre vision de l'évolution du domaine des systèmes répartis en général, avec la prise en compte de nouveaux besoins liés au multimédia et au travail coopératif ainsi que l'impact des nouvelles technologies telles que ATM et processeurs 64 bits. Ce chapitre se termine par la présentation des perspectives nées de ce travail, notamment dans le support système des bases de données.

Chapitre II

Support de la persistance

Pour définir le cadre de notre étude, nous utilisons la définition que donne Thatte d'un système support de la persistance : "*Un système à mémoire persistante offre une architecture de stockage pour la* ***conservation fiable****, et à long terme, d'objets possédant des types et des* ***structures complexes*** *en* ***mémoire virtuelle***".

D'après cette définition, les trois caractéristiques principales d'un système support de la persistance sont : la conservation des données, la capacité à manipuler des données complexes ainsi que la capacité à calquer la représentation des données en mémoire de stockage sur celle en mémoire virtuelle.

Selon que l'on s'intéresse aux systèmes de gestion de fichiers, aux systèmes de gestion de base de données ou aux langages de programmation persistants, l'intégration de ces trois caractéristiques sera plus ou moins achevée.

Historiquement, les systèmes d'exploitation n'offraient qu'un seul outil de persistance : le fichier, c'est à dire une unité de conservation non structurée (si ce n'est sous la forme d'une suite contiguë d'octets). Les programmeurs d'applications étaient (et sont toujours) obligés de convertir les structures manipulées par leurs applications en une représentation pouvant être stockée de manière linéaire dans un fichier. Outre le coût en terme de développement, cette solution induit un coût important à l'exécution à cause de ces changements de représentation.

C'est pourquoi dans les années quatre−vingts des langages comme PS−Algol [Cockshot 84] ont introduit l'idée de la programmation persistante. L'idée de base est d'offrir aux programmeurs un langage pour lequel le système assure automatiquement la conservation de tout ou partie des données.

Le présent chapitre est structuré comme suit : la section II.1 est une courte présentation du support de la persistance fourni par les systèmes de gestion de fichiers ; la section II.2 présente le modèle de mémoire persistante de Thatte (similaire à celui de Multics) qui servira à définir le cadre conceptuel de notre étude ; la section II.3 définit les principales caractéristiques d'un système support de la persistance ; la section II.4 illustre de manière pratique, par la présentation de quatre systèmes, les divers choix de réalisation ; la section II.5, à la lumière des sections précédentes, s'attache à définir les principales qualités d'un "bon" support de la persistance.

II.1 La persistance dans les systèmes de gestion de fichiers

Nous allons tout d'abord faire un bref rappel sur la mise en œuvre de la persistance dans les systèmes de gestion de fichiers (SGF) fournis par les systèmes d'exploitation classiques.

Un SGF assure trois fonctions : il permet la ***conservation*** des données sous la forme de fichiers (c'est à dire, un ensemble contigu ou non d'octets), il fournit une fonction de ***désignation*** des fichiers afin que les utilisateurs puissent les manipuler au travers de noms symboliques et enfin il fournit une fonction de ***protection*** entre les utilisateurs.

Au niveau du traitement de la persistance des données (de leur conservation sous forme de fichier), les environnements classiquement bâtis au−dessus des SGF (comme l'environnement C/C++) souffrent de deux défauts majeurs :

- L'appel aux fonctions de conservation, que se soit pour l'écriture ou la lecture, est explicite. Le programmeur d'applications doit donc gérer les transferts entre la mémoire de ses applications et les disques de conservation.
- L'unité d'échange entre la mémoire de travail et la mémoire de stockage n'est pas le fichier mais l'enregistrement ou le bloc d'enregistrements (c.a.d. une chaîne de caractères quelconques). Il est donc nécessaire de convertir la représentation des données en mémoire virtuelle, en une représentation différente propre à être conservée. Le système introduit une dichotomie, au niveau des traitements, entre les données temporaires et les données persistantes.

Des efforts ont été entrepris depuis une dizaine d'années pour atténuer, voire éliminer les contraintes dues à la persistance classique. C'est ainsi qu'a été introduite au début des années 80, la notion de programmation persistante. L'idée de base consiste à associer aux données la propriété d'être persistantes et de laisser le système mettre en œuvre cette propriété de manière automatique.

De plus, le système essaye de conserver la même représentation des données en mémoire virtuelle qu'en espace de conservation éliminant par la même les conversions d'une représentation dans l'autre.

La mise en œuvre de ces différentes idées suppose que le système d'exploitation offre un support approprié. Dans la section suivante, nous allons donc étudier les problèmes liés à ce support.

II.2 Modèle de persistance

Thatte propose un modèle de mémoire à un niveau pour la réalisation de la persistance [Thatte 86]. En faisant abstraction des mémoires principale et secondaire, on est amené à considérer la mémoire comme un espace unique où se trouvent tous les objets. Le modèle comporte un mécanisme de gestion de la persistance dont le principe de base est le même que celui de PS−Algol [Cockshot 84] : la durée de vie d'un objet est indépendante de celle du programme qui l'a créé ; elle dépend d'un programme indépendant de nettoyage (le ramasse−miettes) qui obéit à des règles strictes pour déterminer ce qu'il peut récupérer et ce qu'il doit conserver.

Nous avons choisi le modèle de Thatte car il définit un cadre conceptuel dans lequel s'inscrivent ou duquel dérivent la plupart des modèles de persistance proposés par les systèmes, ou bases de données, actuels comme COOL[Amaral 92], Cricket [Shekita 90] et Guide [Balter 91]. Notons que le système Multics [Organick 72] fournissait déjà une mémoire segmentée uniforme à ses utilisateurs.

Thatte édifie sa mémoire persistante en utilisant ce qu'il appelle "l'abstraction de la mémoire uniforme". Il s'agit d'une abstraction parce que seules sont définies les caractéristiques externes de la mémoire. La mémoire est uniforme car aucune distinction n'est faite entre objets temporaires et persistants [Scioville 84].

Du point de vue du processeur qui l'utilise, une mémoire persistante possède les caractéristiques suivantes :

- Un objet est un segment de mémoire selon son sens usuel dans les systèmes d'exploitation ; c'est une suite contiguë d'octets dont la taille peut varier dynamiquement.
- Un objet peut contenir des pointeurs (adresses en mémoire) vers d'autres objets.

Les caractéristiques qui mettent en évidence la persistance sont les suivantes :

- Un objet particulier de la mémoire, localisé à une adresse fixe, est la racine de persistance de la mémoire.

  Les objets qui peuvent être atteints à partir de la racine en suivant les pointeurs sont des objets persistants. On définit ainsi une relation "est−persistant" qui lie un objet persistant aux objets qu'il peut adresser.

  La persistance d'un objet ne dépend donc que de son appartenance à la fermeture transitive de cette relation en considérant la racine comme objet initial.
- Le processeur peut accéder aux objets de la mémoire si ses registres contiennent des pointeurs d'objets. Les registres du processeur définissent la racine temporaire de la mémoire : tous les objets qui appartiennent à la fermeture transitive de la racine temporaire, mais qui n'appartiennent pas à celle de la racine persistante, sont des objets temporaires.

- Les objets qui ne sont ni persistants ni temporaires sont des miettes.
- Le processeur ne peut pas distinguer les objets temporaires des objets persistants ; ces deux classes d'objets ne peuvent donc pas être traitées de manière différente.

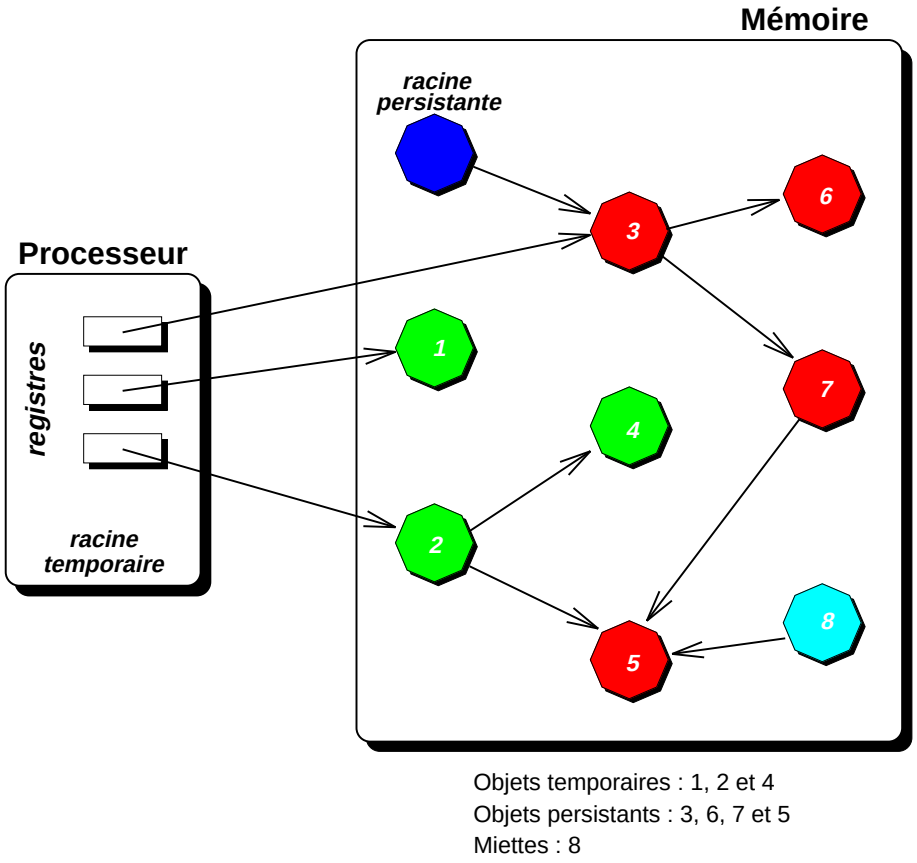

Fig. 2.1 : Modèle de la mémoire persistante

La figure Fig. 2.1 schématise le modèle de mémoire persistante. Trois considérations doivent être prises en compte lors de la mise en œuvre de ce modèle :

- Tous les utilisateurs peuvent accéder aux objets de la mémoire persistante. Son implantation doit donc offrir des mécanismes de partage d'objets et de contrôle de concurrence.
- La mémoire persistante est en même temps un support d'exécution et de conservation des objets. Une coopération entre ces deux fonctions est donc nécessaire pour son implantation. De ce point de vue, la mémoire persistante se présente comme une mémoire virtuelle d'objets dans laquelle la mémoire principale sert d'espace d'exécution, et la mémoire de pagination d'espace de conservation.

- La fonction de conservation doit garantir l'intégrité des objets. Cela signifie que l'état de l'espace de conservation doit être cohérent par rapport à celui de l'espace d'exécution. Il est nécessaire d'offrir des mécanismes de résistance aux pannes assurant aux objets robustesse et cohérence.

Le modèle de mémoire persistante définit un cadre conceptuel sur lequel reposent les langages de programmation persistants. Nous allons maintenant nous intéresser au support d'un tel modèle par le système d'exploitation sous−jacent.

II.3 Support d'un modèle de mémoire persistante

Le but de la présente section est d'évaluer les incidences du modèle de mémoire persistante sur la mise en œuvre de la persistance dans un système réparti et d'identifier notamment les principales caractéristiques d'un tel support. Dans cette optique, une étude de cas sur différents systèmes support de persistance est proposée dans la section II.4.

L'influence de la répartition sur la gestion de la persistance concerne principalement trois aspects :

- La ***désignation*** qui englobe les problèmes d'identification des objets, ainsi que ceux relatifs à leur accès. Le système doit être capable de retrouver les objets conservés mais aussi de gérer les accès aux objets composites (ensemble ou groupe d'objets).
- Le ***partage*** qui regroupe la gestion de la concurrence d'accès et le maintien de la cohérence des données en mémoire virtuelle. La gestion du partage, notamment en mémoire virtuelle, est ce qui différencie le plus un système support de persistance d'un SGF classique.
- La ***conservation*** qui doit garantir l'intégrité des données en espace de conservation, prenant notamment en compte les problèmes de reprise après panne. A ce niveau, on attend du support de la persistance qu'il offre une qualité de service proche de celle fournie par les systèmes de gestion de base de données.

II.3.1 La désignation

La taille de cette sous−section a été volontairement réduite car nous ne nous intéressons pas au problème de la désignation, dans les systèmes répartis, dans sa globalité. Ce qui nous intéresse, c'est que le choix de telle ou telle solution repose sur un compromis entre l'efficacité que l'on veut offrir et la mobilité des données nécessaire aux applications et aux utilisateurs. Le lecteur intéressé par une présentation plus complète du problème de la désignation dans les systèmes répartis pourra se

référer à [Hagimont 93] et [Layaida 93] ainsi qu'à la présentation de l'état de l'art du domaine dans [Legatheaux 88].

Dans les systèmes répartis à objets, tout objet peut référencer des objets qui se trouvent sur des sites différents. Les mécanismes de désignation doivent donc permettre de retrouver la représentation physique d'un objet à partir de l'identificateur logique qui sert à le désigner et ce, quelque soit son site de résidence. La localisation s'effectue de préférence de manière "transparente" aux applications et à leurs usagers.

Il existe de très nombreuses solutions aux problèmes induits par une telle désignation globale (ou uniforme) dans les systèmes répartis, allant de la connexion forte entre identificateurs logiques et localisation physique comme dans les systèmes Guide [Balter 91] et Thor [Day 92] jusqu'à la déconnexion totale entre ces deux aspects comme dans Clouds [Dasgupta 90] et Hermes [Black 90]. Les systèmes de fichiers NFS ou AFS maintiennent, par exemple, des tables qui réalisent la correspondance *<nom de fichier, site de résidence>*. La connexion forte permet une recherche beaucoup plus efficace en espace de conservation (cf [Day 92]). Par contre, elle rend beaucoup plus complexe la prise en compte des objets déplacés d'un site vers un autre (aussi appelée migration). Les SSP Chains [Shapiro 92] sont un exemple d'une solution offrant une très grande flexibilité, notamment pour ce qui touche aux déplacements des objets et au ramassage des miettes.

Le point qui nous paraît essentiel est que le choix de l'une ou l'autre des désignations résulte d'un compromis entre efficacité de la localisation et mobilité des objets, qui conditionne les performances ainsi que la réalisation du système.

Les bases de données [Deux 91] offrent par ailleurs des fonctions d'accès ensemblistes aux objets. Un ensemble est un objet composite qui peut être réalisé par un objet qui contient des références vers d'autres objets ou être un moyen d'accès intégré au schéma de désignation afin de l'accélérer. Dans le premier cas, l'appartenance d'un objet à un ensemble sera complètement indépendante du schéma de désignation. Dans le deuxième cas, elle pourra être inscrite dans l'identificateur logique ou physique de l'objet.

II.3.2 Le partage

Le modèle de mémoire persistante définit un espace d'adressage unique et partagé par toutes les applications. Un système réparti doit donc offrir un mécanisme de partage qui tienne compte de deux facteurs :

- La nécessité d'offrir aux applications une vision cohérente de la mémoire (au niveau de l'objet),
- La fourniture de mécanismes de gestion de la concurrence d'accès.

Le système Eos [Gruber 92] intègre ces deux points dans sa gestion de la mémoire alors que le système Clouds les traite de manière complètement indépendante.

La mise en œuvre d'une mémoire globale est basée sur l'utilisation, conjointe ou non, des trois mécanismes suivants :

- déplacement de l'exécution, le partage d'une donnée s'effectue alors de manière centralisée sur un seul site en utilisant les outils fournis par le système opératoire de ce site : mémoire partagée, par exemple. Cette solution est particulièrement intéressante lorsque deux applications accèdent souvent à un même objet simultanément.
- déplacement de l'objet sur le site où on l'accède, on se ramène là aussi à un partage centralisé. Cette solution peut s'avérer nécessaire lorsque une application référence de nombreuses fois un objet pendant des intervalles de temps où elle n'entre pas en compétition sur cet objet avec d'autres applications.
- duplication des objets sur l'ensemble des sites où on y accède. On a alors besoin de gérer la cohérence des copies en fonction des modifications effectuées sur les différents sites. Cette solution est souvent appelée Mémoire Partagée Répartie, on trouvera dans [Li 89] la définition et la réalisation d'une telle mémoire. Les systèmes Casper [Vaughan 92], Clouds, COOL [Amaral 92], Eos et Gothic [Banâtre 92] fournissent un exemple d'utilisation d'une mémoire répartie.

Selon que l'on cherche à ordonner l'accès à un ensemble de ressources ou à une seule ressource, il existe deux types de mécanismes pour le contrôle de la cohérence : la sérialisation de processus et la synchronisation de processus. Le premier s'applique lorsque la cohérence d'un ensemble de données partagées doit être préservée. Il est généralement réalisé en utilisant des verrouillages transactionnels comme dans Eos et O_2 [Deux 91]. Le deuxième sert à modéliser des interactions fines entre applications. Ce peut−être par exemple un problème de lecteurs−rédacteurs ou de producteurs−consommateurs. Sa réalisation utilise des outils classiques de synchronisation : verrous, moniteurs ou envois de message. C'est la technique employée par Clouds.

En ce qui concerne la gestion du partage, les concepteurs des mécanismes sous−jacents sont là aussi confrontés à un dilemme : soit ils offrent une très grande

flexibilité au niveau de la taille des objets partagés et des politiques de mise en œuvre du partage, soit ils cherchent à fournir une efficacité maximale en intégrant au niveau bas du système les mécanismes et la politique de partage.

II.3.3 Le stockage

Le modèle de mémoire persistante impose au système de respecter l'intégrité des données. Ceci signifie que l'état des données conservées doivent être le reflet cohérent de l'état des données à l'exécution dans la mesure où l'accès à des images incohérentes peut entraîner des erreurs d'exécution.

Dans les SGF classiques, cette propriété est plus ou moins implicitement réalisée par le mode de travail. Les objets sont d'abord chargés dans l'espace d'adressage de l'application, puis une fois les calculs terminés, les modifications sont reportées dans l'espace de conservation. En cas de panne lors de la phase de calcul, les calculs sont perdus et le système peut redémarrer en utilisant l'ancienne image des objets sur disque. Le point vulnérable de ce fonctionnement est atteint lors de la re−copie des objets. Il faut garantir que la re−copie est atomique : soit toutes les modifications sont reportées soit aucune. Cela revient à fournir dans un SGF des fonctions transactionnelles du type de celles offertes par un SGBD.

Dans les SGBD, on retrouve principalement deux techniques permettant de garantir l'atomicité des opérations de stockage :

- *La technique dite des pages fantômes* : les modifications ne sont pas reportées sur les pages contenant les données initiales, mais sur de nouvelles pages. L'ancienne et la nouvelle image des données coexistent dans la base jusqu'à la validation (marquée par la fin de la re−copie). Le système supprime alors les pages de l'ancienne image. La validation peut−être vue comme un basculement des données de l'ancienne image vers la nouvelle.

  Si une panne se produit avant la validation cela signifie que toutes les modifications n'ont pas pu être stockées et donc que la nouvelle image est incohérente. Le système conserve donc l'ancienne image et détruit les pages allouées pour la nouvelle image.
- *L'utilisation d'un journal "après"* (resp. *"avant"*) : les modifications (resp. l'image initiale) sont tout d'abord écrites de manière séquentielle dans un fichier appelé journal(1). Lorsque le système est sûr que le journal contient bien les modifications (resp. l'image initiale), il reporte les modifications (dans les deux cas) sur l'image des données en espace de stockage.

---

(1) Les modifications sont toujours ajoutées à la fin du journal. Logiquement, le journal est un fichier de taille continuement croissante.

Si une panne se produit lors de cette re−copie, l'image des données en espace de stockage est potentiellement incohérente. Le journal "après" permet de retrouver les modifications (éventuellement) non reportées et de les reporter une nouvelle fois en espace de stockage (les opérations stockées dans le journal sont idempotentes). Le journal "avant" permet de retrouver l'image initiale des données et de la restaurer en espace de stockage.

Les avantages liés à l'utilisation d'un journal sont :

+ L'ajout des modifications étant toujours réalisé à la fin du journal, la localité des entrées/sorties sur disque est très élevée. Cela procure de très bonnes performances. On se reportera à la description du système de fichiers journalisé Sprite LFS [Rosenblum 91].
+ La mise en œuvre du journal peut être réalisée à haut niveau (par exemple, au−dessus du système de fichiers Unix).
+ Le journal est utilisé pour la reprise après panne : enregistrement des opérations et des transactions en plus des seules données.
+ La base de données peut être mise à jour de manière asynchrone (dès que le journal a été validé).

Les points faibles de la journalisation sont :

− La reprise après panne est lente dans le cas où on utilise un journal "avant".
− La nécessité de rendre le journal fiable complique sa mise en œuvre (ce qui est d'autant plus important que la mise en œuvre est de haut niveau).

Les pages fantômes ont pour avantages :

+ La validation peut−être très efficace. Si le descripteur des pages de la base tient dans un bloc du disque, une seule entrée/sortie suffit.
+ La reprise après panne est immédiate puisque les pages qui n'appartiennent pas à la liste des pages valides sont considérées comme libres.

Le problème des pages fantômes vient de la complexité de leur mise en œuvre :

− La réalisation doit être effectuée au niveau des blocs du disque si l'on veut qu'elle soit efficace.
− La gestion des tampons est complexe surtout si l'on veut essayer de favoriser les écritures de pages contiguës.

La cohérence peut−être garantie à plusieurs niveaux :

- Au niveau de l'entité de conservation : le système garantit l'intégrité d'un objet indépendamment des autres objets. Un état de l'espace de stockage

peut ne pas être cohérent au sens des relations entre objets (c'est le cas dans Clouds).

- Au niveau d'un ensemble d'entités : l'atomicité des opérations de conservation est alors garantie pour tous les objets appartenant à cet ensemble. Un état de l'espace de stockage reflète alors un état cohérent de l'ensemble des objets les uns par rapport aux autres. Tous les systèmes issus du domaine des bases de données fournissent cette qualité de service, on la retrouve notamment dans Casper, Eos et O_2.

De même que pour le support du partage, il est ici aussi question de compromis entre flexibilité et efficacité. Il s'agit surtout de concilier deux objectifs plus ou moins antinomiques : le désir d'offrir une grande modularité et des fonctions capables de traiter des objets de petite taille, avec le besoin impérieux d'offrir de bonnes performances (ce qui implique de limiter le nombre des entrées/sorties sur disque et donc d'augmenter la taille des entités transferées sur disque).

II.3.4 La protection

La dernière caractéristique du modèle de mémoire persistante est liée aux problèmes de protection. On peut voir l'espace des objets persistants comme un grand sac dans lequel viennent piocher les utilisateurs. Or les utilisateurs ont parfois besoin de se protéger ou de protéger leurs objets. On peut faire ici l'analogie avec ce qui se passe dans les SGF classiques, dans lesquels un utilisateur se voit conférer des droits sur un ensemble de fichiers.

Dans la suite de cette sous−section, nous nous intéressons seulement aux aspects de la protection qui sont liés au support de la persistance. On trouvera dans [Hagimont 94] une description détaillée des problèmes et des politiques de protection dans les systèmes répartis.

Traditionnellement, deux techniques sont utilisées pour mettre en œuvre la protection des objets en espace de conservation :

- L'utilisation de capacités pour désigner les objets : une capacité est un identificateur protégé par le système qui permet à toute personne la possédant d'accéder à l'objet qu'elle désigne. La désignation de l'objet est, ainsi, associée étroitement à sa protection. En contrôlant la diffusion des capacités sur ses propres objets, un utilisateur assure ainsi leur protection. Les systèmes Clouds et COOL basent, en partie, leur protection sur ce mécanisme.

  En général, et afin de faciliter leur utilisation, les capacités sont regroupés dans des domaines de protection (ensemble d'entités munies de droits). Le problème (du point de vue des utilisateurs) est alors de définir ces domaines

de protection De plus, leur représentation ne doit pas pénaliser les performances d'accès et doit leur permettre d'évoluer (ajout d'un objet dans un domaine).

- Le contrôle par listes d'accès : une liste d'accès regroupe un ensemble de droits sur un objet associés à différents utilisateurs. Pour accéder à un objet, un utilisateur doit faire partie de la liste d'accès de cet objet. La protection des fichiers dans le système AFS est basée sur l'utilisation de liste d'accès.

  Là encore, un des problèmes à résoudre est celui de la représentation des listes d'accès.

Cependant, au niveau du support de la persistance, on s'aperçoit que l'on s'interesse, en fait, au seul contrôle de l'accès (chargement) aux objets. Il s'agit principalement de garantir l'isolation des objets d'un utilisateur, c'est à dire le couplage des données de différents utilisateurs dans des espaces d'adressage différents.

Cela conduit à définir un schéma de protection à deux niveaux :

1. Le support de la persistance protège, en les isolant, des entités grossières (semblables à des fichiers) à l'aide de droits Lecture/Écriture/Exécution associés aux utilisateurs. Le système Clouds utilise, par exemple, un espace d'adressage par objet.
2. Les langages de programmation mettent en œuvre une protection des objets exprimée en terme d'opérateurs (méthodes) applicables. Cette protection repose sur la garantie d'isolation fournie par la couche basse du système.

La notion d'utilisateur doit donc impérativement être prise en compte par le système support de la persistance.

II.4 Présentation de différents systèmes

Dans notre étude, nous nous intéressons au support de la persistance dans le système réparti Guide. L'objectif de ce projet est de définir et de réaliser un environnement de développement et d'exécution d'applications réparties.

Dans cette optique, il nous paraît important de souligner que fournir une solution optimale pour résoudre un problème particulier au support de la persistance est une chose très différente de celle consistant à fournir une solution globale cohérente et efficace à l'ensemble des problèmes rencontrés. C'est pourquoi, nous allons dans la suite de cette étude, nous intéresser à l'analyse de systèmes complets.

Pour illustrer notre propos, nous allons maintenant décrire quelques systèmes qui réalisent le modèle de mémoire persistante de Thatte. Cette description se

concentrera essentiellement sur le support de la persistance au travers du filtre défini dans la section II.3.

Le premier système présenté, le système Clouds, nous sert à illustrer la nécessaire séparation entre les objets utilisés par les langages de programmation (ceux définis dans le modèle de Thatte) et les entités de bas niveau manipulées par le système pour tout ce qui a trait à la gestion de la mémoire et de la persistance. Le système COOL est un autre exemple d'un système supportant des objets persistants issus du domaine des systèmes d'exploitation.

Le système DSR est un exemple de support de la persistance réalisé au niveau des couches hautes d'un environnement d'exécution (c.a.d., celui des applications dans Unix). Il illustre également la prise en compte, au niveau du modèle de persistance, des données et des exécutions.

Le système Cricket est basé sur l'utilisation des primitives de gestion de la mémoire virtuelle offerte par le micro−noyau Mach aux applications. L'architecture ainsi définie offre souplesse et efficacité. La mise en œuvre de Cricket est similaire à celle de Casper et d'Eos.

Enfin le serveur O2 est un représentant typique d'un serveur de page utilisé dans le domaine des bases de données. Sa réalisation est entièrement faite en espace utilisateur et ne bénéficie pas de la coopération avec le noyau que l'on trouve dans les systèmes précédants.

II.4.1 Le système Clouds

Clouds [Dasgupta 90] est un projet de recherche du Georgia Institute of Technology dont le but est la réalisation d'un système distribué à base d'objets. Deux des principaux domaines explorés sont la tolérance aux fautes et la reconfiguration dynamique.

Le système Clouds est réalisé au dessus du noyau Ra sur un ensemble de stations Sun 3/60 reliées par un réseau Ethernet. Ra [Bernabeu 88] est un noyau minimal, développé par la même équipe que Clouds, qui fournit les outils nécessaires pour la réalisation de systèmes d'exploitation distribués.

Clouds définit deux concepts :

- L'***objet*** est conceptuellement vu comme un espace virtuel d'adressage persistant(2). L'objet est une entité passive dont la durée de vie est indépendante de la durée de vie de tout flot d'exécution. Il survit aux arrêts et aux pannes du système. Sa destruction est explicite.
- Le ***flot d'exécution*** (ou "thread") est l'entité active du système. Il peut être vu comme un processus léger qui s'exécute dans les espaces virtuels d'adressage constitués par les objets sur lesquels il exécute des méthodes.

(2) Notons que Clouds gère aussi des objets temporaires.

Un flot peut ainsi "traverser" plusieurs objets. Plusieurs flots peuvent s'exécuter concurremment dans un même objet, la synchronisation de base est associée aux opérations de l'objet.

Désignation et localisation

Les objets sont désignés par des identificateurs universels (sysname). Les sysnames sont des identificateurs logiques ne contenant aucune information de localisation physique.

La localisation d'un objet se fait d'abord par recherche dans des tables locales à la machine courante. Puis par la consultation des disques locaux. Enfin, en cas d'échec, le système diffuse la demande de localisation à l'ensemble des machines.

Partage

Chaque objet est représenté par un espace d'adressage partagé distinct. Le partage d'objet est réalisé par partage de l'espace d'adressage qui le constitue par les différents flots d'exécution.

Dans la pratique, un espace d'adressage vu par un flot d'exécution est constitué de trois régions : la O−région dans laquelle se trouve l'objet proprement dit, la P−région dans laquelle se trouve la pile du flot d'exécution et la K−région utilisée par le noyau.

L'appel de méthode sur un objet O2 depuis un objet O1 provoque la désinstallation de l'objet courant O1 de la O−région, l'installation de O2 dans la O−région. Puis une fois l'invocation terminée, la réinstallation de O1.

Stockage

La mise en œuvre de la persistance des objets est assurée par le noyau au travers de segments. Le ***segment*** est un bloc non−typé et contigu de mémoire (le noyau n'associe aucune sémantique quant au contenu du segment). Un objet est réalisé par un ou plusieurs segments ; en général, un segment de code et un segment de données.

Le segment permet au noyau d'accéder directement aux pages des objets (par exemple, pour pouvoir les paginer) au travers des mécanismes de gestion de la mémoire virtuelle.

Les segments sont créés de manière explicite ; ils persistent jusqu'à leur destruction, elle aussi explicite. Les segments sont désignés par un nom global unique. La localisation d'un segment, similaire à celle d'un objet, se fait par recherche dans la table locale des segments actifs (en cours d'utilisation), puis par diffusion de la demande.

Les segments sont gérés par des serveurs de partitions. Une ***partition*** est une entité active responsable de la gestion d'un ensemble de segments dans l'espace de conservation. Le noyau lui réclame les segments lorsque cela est nécessaire (installation d'un segment), puis les libère lorsqu'ils ne sont plus utilisés.

Le noyau accède donc aux segments au travers de l'interface des partitions. Cette interface de bas niveau offre des fonctions de création, destruction, extension et installation ainsi que des fonctions permettant le chargement/déchargement ("page in/out") des pages de segments.

Notons que les segments et les partitions de Clouds sont des concepts respectivement proches des objets mémoires et des paginateurs externes du micro−noyau Mach (cf sous−section III.3.3).

Le stockage des segments est réalisé en utilisant le système de fichier Unix. Chaque segment est stocké dans un fichier Unix. Le serveur de partition responsable du segment fournit les pages de ce segment au noyau Ra à travers le réseau. Ce mécanisme est rendu transparent au noyau grâce à l'interposition d'un client de partition sur la machine du noyau (cf Fig. 2.2).

Protection

La protection dans Clouds est réalisée par isolation des objets. Chaque objet est représenté par un espace d'adressage distinct dans la mesure où le système ne place qu'un seul objet à un instant donné dans une O−région. On peut considérer qu'un flot d'exécution traverse les espaces d'adressage des objets qu'il invoque au cours de son exécution.

La protection entre utilisateurs ou applications est basée sur l'utilisation de capacités. Vu d'un utilisateur, un objet est identifié par une capacité qui contient l'identificateur de l'objet ainsi que des droits d'accès. Le propriétaire d'un objet peut contrôler la diffusion de la capacité associée à son objet vers les autres utilisateurs.

La figure Fig. 2.2 présente l'architecture générale de Clouds ainsi que les principaux concepts du système.

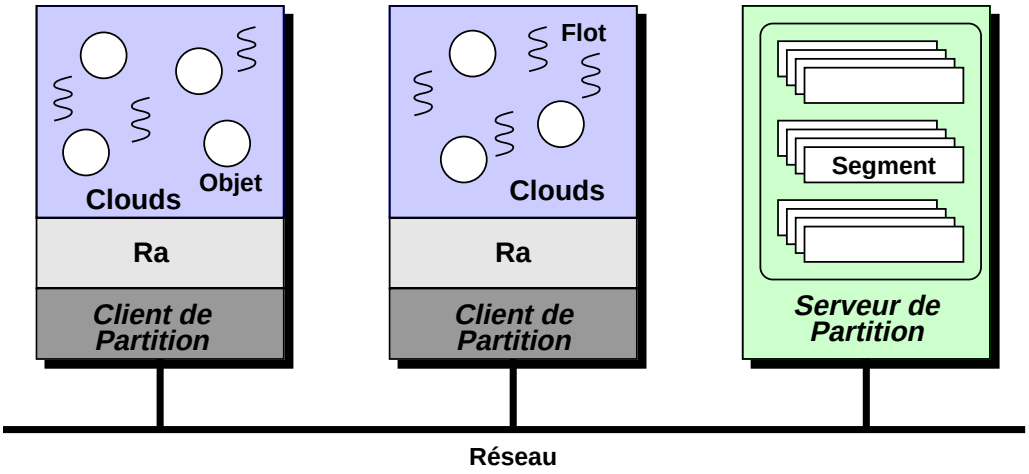

Fig. 2.2 : Architecture de Clouds

Les points forts de Clouds sont :

- L'intégration du support de la persistance avec la gestion de la mémoire virtuelle qui est rendue possible par la réalisation du système et du noyau sur machine nue.
- La protection par couplage des objets dans des espaces d'adressage séparés.

Le principal point faible de Clouds est le coût de l'appel inter−objets qui nécessite un changement d'espace d'adressage (commutation de contexte). Il en résulte que Clouds ne supporte pas directement des objets de petite taille, mais que ceux−ci sont mis en œuvre par les langages, dans des objets serveurs fournis par Clouds.

Par ailleurs, la séparation entre les machines de calcul qui supportent le système Clouds et les machines de stockage sur lesquelles s'exécutent les partitions entraîne un trafic réseau important que ce soit pour le chargement des objets mais aussi pour le traitement des fautes de pages.

Le point qui nous intéresse plus particulièrement dans Clouds est lié à la différence entre objets et segments. Les objets sont considérés comme étant des entités applicatives (c'est à dire, manipulées par les utilisateurs au travers de langages de programmation) alors que les segments ont été introduits pour permettre la manipulation de la mémoire par le noyau. Le noyau a en effet besoin d'une entité de bas niveau qui lui permette de faire le lien entre l'objet persistant et la gestion de la mémoire virtuelle. Ce lien repose sur la notion de page que l'on retrouve classiquement dans les systèmes d'exploitation.

De manière complémentaire à la notion de segment, Clouds utilise les partitions comme des serveurs de persistance. Les serveurs de partitions permettent une mise en

œuvre efficace du support de la persistance car ils font partie de l'espace du noyau, tout en offrant une grande flexibilité.

II.4.2 Le Distributed Shared Repository

Le Distributed Shared Repository (DSR) est un système de gestion d'information persistante et distribuée développé par une équipe des universités de Tsukuba et de Tokyo [Kato 93]. L'objectif majeur de ce projet est de définir une architecture permettant le traitement uniforme des calculs distribués et des traitements persistants. Cet objectif est basé sur la constatation que ces deux types de traitement ont en commun le fait d'engendrer un échange d'information entre l'espace d'adressage de l'utilisateur et le monde extérieur. Il convient donc de les unifier afin de simplifier l'écriture de telles applications. La solution proposée n'est pas dépendante d'un langage de programmation ni d'un matériel ad hoc, ce qui fait l'originalité de la proposition.

Les concepts de base du DSR sont :

- Le ***contexte*** représente l'état de l'exécution d'une activité. Dans le même temps, il définit l'unité de chargement de mémoire dans les tâches.

  Pour assurer l'uniformité requise, tous les programmes et les données qui sont persistants et utilisés de manière répartie sont assimilés à des contextes. Un contexte peut contenir un programme, des données manipulées par un programme, des piles d'exécution ainsi que l'état d'un processeur (CPU et registres). Les contextes sont conservés par le DSR (qui gère l'espace persistant).

  Les contextes sont regroupés en trois classes différentes. Les contextes de Type I contiennent exclusivement des données. Les contextes de Type II contiennent à la fois des données et du code exécutable. Les contextes de Type III contiennent en plus des données et du code, un segment dans lequel est rangé l'état d'une exécution (la pile du processus, les registres et l'état de la mémoire). Les contextes de Type III sont des contextes actifs.
- La ***tâche*** représente un espace d'adressage virtuel dans lequel les contextes sont chargés. Il s'agit là d'un chargement depuis le DSR et non pas d'un couplage.
- Le ***flot d'exécution*** représente le chemin d'exécution d'une activité vu par un processeur. Un flot s'exécute dans les contextes d'une tâche. Une tâche peut posséder plusieurs flots s'exécutant concurremment.

Le DSR est réparti sur un ensemble de sites et offre aux utilisateurs la vision d'un espace persistant unique composé de contextes.

Désignation

Les contextes sont désignés par une clé unique du système. Cette clé est indépendante de la localisation physique. Le système est capable d'identifier par sa clé et de charger dans l'espace d'adressage virtuel d'un utilisateur, n'importe quel contexte.

La localisation d'un contexte en espace de conservation est basée sur l'utilisation d'une table de préfixes basée sur le regroupement des contextes dans des domaines de stockage.

Partage

Le partage "instantané" de contextes entre tâches n'est pas géré (car les concepteurs du système se veulent indépendants de toute plate−forme matérielle). Par contre, deux tâches peuvent "échanger" des contextes en les faisant transiter par l'espace persistant. Notons que la communication entre tâches est supposée être réalisée par l'utilisation de primitives de communication du type envoi de message.

Stockage

Les contextes sont conservés dans des c−domaines qui sont des entités logiques de gestion de l'espace persistant. Les c−domaines sont aussi utilisés pour accélérer la localisation des contextes. Un contexte ne peut pas changer de c−domaine sans changer de clé. On peut donc logiquement conclure que la clé contient l'identification du c−domaine.

Le chargement d'un contexte dans une tâche se traduit par la disparition (logique) de ce contexte du DSR. Les autres tâches ne pourront charger ce contexte qu'après son déchargement. L'image du contexte dans le DSR reflète donc toujours un état cohérent. Les concepteurs du DSR ne décrivent cependant pas la manière dont ils assurent la recopie atomique d'un contexte dans le DSR.

Protection

Aucun mécanisme de protection n'est décrit dans [Kato 93].

La figure Fig. 2.3 présente le modèle de programmation du DSR.

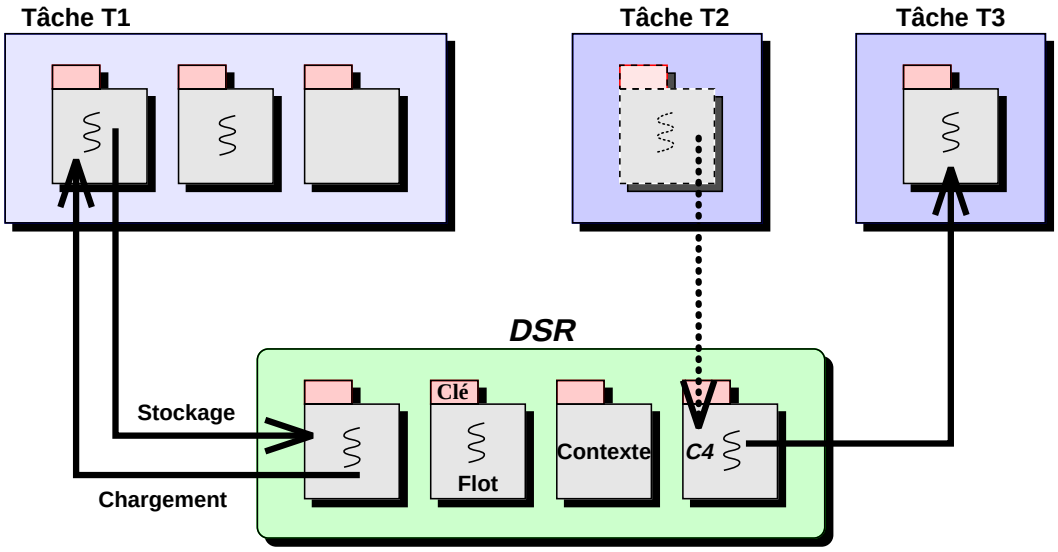

Fig. 2.3 : Modèle de programmation du DSR

> Sur cette figure, on aperçoit deux utilisations possibles du DSR. La première employée par la tâche T1 consiste à utiliser le DSR comme support de persistance grâce aux opérations de chargement et de déchargement de contextes.
>
> Les tâches T2 et T3 utilisent quand à elles le DSR pour s'échanger des données. T3 a en effet lu le contexte C4 que T2 venait de ranger dans le DSR.

L'intérêt principal du DSR tient dans son caractère général puisqu'il n'est basé sur l'utilisation d'aucun mécanisme système ou matériel particulier, ce qui est une réalisation diamétralement différente de celle du système Clouds.

Le contexte est un concept très puissant puisqu'il uniformise le traitement des données, du code et des structures d'exécutions.

Par contre, le désir d'indépendance vis à vis du matériel se paye au niveau des services rendus : pas de partage instantané de contextes et chargement des contextes coûteux, puisque ne pouvant pas bénéficier d'un chargement à la demande de la part des mécanismes de gestion de la mémoire virtuelle.

Le point qui nous paraît le plus discutable dans le DSR a trait au mode de communication entre tâches. Le DSR offre un espace persistant distribué composé de contextes auxquels les applications accèdent de manière transparente, mais il n'intègre pas de modèle de communication. Le programmeur doit utiliser des primitives de communication explicite (type envoi de message) pour que deux tâches coopèrent. Ainsi dans la figure Fig. 2.3, les tâches T2 et T3 peuvent s'échanger un contexte (et donc des données) mais il faut que T2 envoie un message à T3 pour que celle−ci sache à quel moment et quel contexte elle doit charger.

Il parait curieux que les concepteurs du DSR n'aient pas cherché à utiliser le contexte comme seul et unique moyen de communication. Cela aurait eu l'avantage de renforcer l'uniformisation des traitements répartis qui est l'objectif du projet.

Il est intéressant de noter que le DSR fait aussi la distinction entre les objets qui sont du niveau langage et les contextes qui sont du niveau système, notamment pour tout ce qui a trait au support de la persistance.

II.4.3 Le gestionnaire de persistance Cricket

Cricket [Shekita 90] est un gestionnaire de persistance développé à l'université de Wisconsin−Madison. Cricket utilise les primitives de gestion de la mémoire du micro−noyau Mach [Accetta 86] pour construire un espace persistant partagé à un niveau (c'est à dire que les données persistantes et volatiles sont vues de la même manière par les applications). Il s'agit en fait d'une réalisation du modèle de mémoire persistante de Thatte et plutôt qu'espace à un niveau, nous préférerons qualifier Cricket d'espace persistant uniforme.

Cricket est un projet menée par une équipe spécialisée dans les bases de données dont une précédente réalisation a été le serveur de stockage EXODUS. L'architecture et les fonctions de Cricket reflètent donc cet héritage.

L'architecture de Cricket est basée sur le paradigme client−serveur. Les applications clientes s'exécutent dans des processus distincts et utilisent une interface de type appel de procédure à distance pour envoyer des requêtes de bas niveau au serveur Cricket. Ces requêtes permettent d'établir une connexion avec le serveur, de démarrer puis de valider une transaction.

Désignation

Cricket ne fournit pas d'entités de haut−niveau telles que les objets. Il n'offre donc pas de mécanismes de désignation à proprement parler. Les applications accèdent simplement aux pages de la base qui est vue comme un espace d'adressage linéaire. Les pages sont désignées par leur adresse dans la base.

Partage

Les applications accèdent et partagent directement les données persistantes. La base de données Cricket est mise en œuvre par un objet mémoire Mach (cf sous−section III.3.3) qui est couplé dans l'espace d'adressage des applications(3). Le serveur Cricket joue le rôle du paginateur externe responsable du traitement des fautes de pages sur cet objet.

---

(3) La base est couplée à la même adresse par toutes les applications, ainsi les adresses contenues dans la base sont valides dans tous les espaces d'adressage.

Le contrôle des accès concurrents sur la base est traité de manière transparente (vis à vis des applications) grâce au mécanisme de traitement d'exceptions de Mach. L'accès à une page provoque si besoin est le déclenchement d'une exception transmise au serveur. Le serveur peut alors contrôler et positionner les droits d'accès à la page et donc contrôler la cohérence des diverses copies de la même page.

Stockage

La mise en œuvre du stockage est réalisée comme dans la plupart des bases de données à un niveau très bas qui correspond au bloc du disque. La base de données est structurée en "extents", contenant chacun le même nombre de pages (au moins 16K octets).

Cricket fournit des primitives d'allocation de blocs mémoires dans les "extents". Ces primitives sont appelées par les langages pour créer des objets (manipulables depuis les constructeurs du langage).

L'intégrité est assurée par l'utilisation de pages fantômes. Les pages modifiées sont stockées à une adresse sur disque différente de leur image initiale. Lorsque la transaction qui a modifié les pages, valide, les pages fantômes remplacent les pages initiales. L'avantage de cette technique tient au fait que la conservation de l'ancienne image est assurée tant que la transaction n'a pas été validée.

Cricket utilise un mécanisme complémentaire qui est la journalisation des opérations telles que l'allocation d'un bloc du disque ou la création des index. En fait, tout ce qui a trait aux méta−données (c'est à dire, les données propres au système) est journalisé.

Protection

Le premier niveau de protection offert par Cricket est l'isolation des applications dans des espaces d'adressage séparés. Seule la base de données est effectivement partagée par les applications. De plus, les droits d'accès sur les pages sont spécifiés en fonction de la politique de gestion de la concurrence et ne peuvent pas être modifiés par les applications.

De plus, toutes les fonctions de gestion de la persistance (tant celles concernant l'accès à l'espace de stockage que son allocation) sont isolées dans le serveur, offrant ainsi un niveau de protection supplémentaire.

La figure Fig. 2.4 donne une idée de l'architecture de Cricket dans sa configuration répartie (accès distant des clients à une base centralisée).

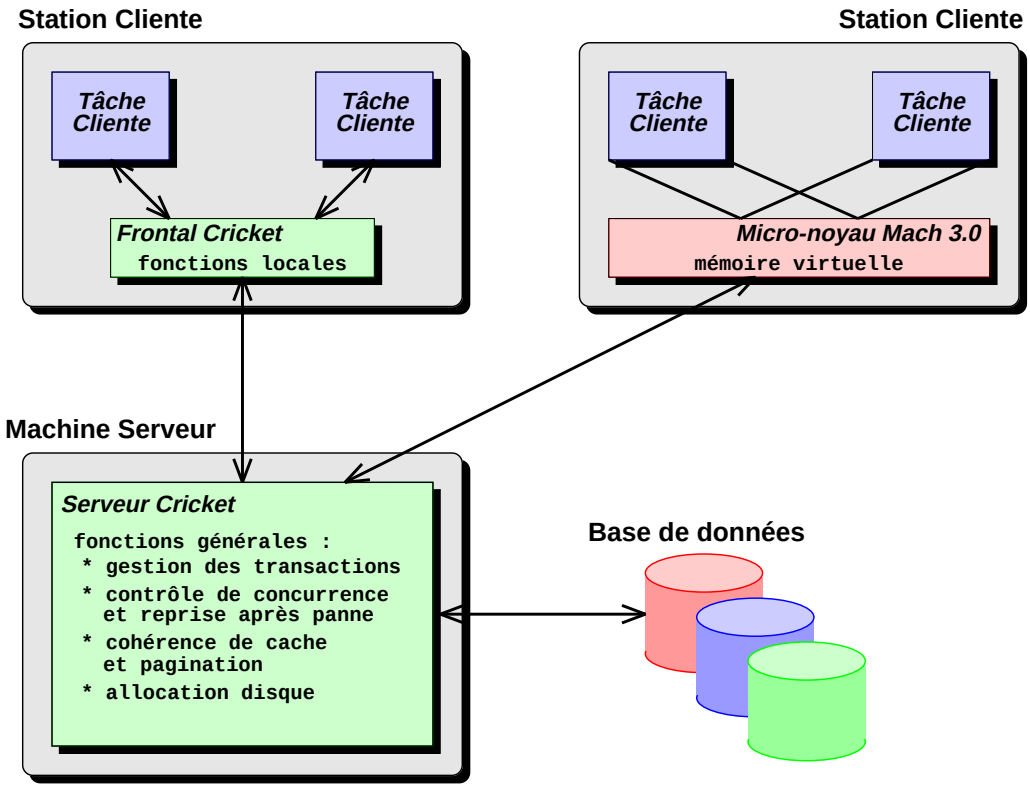

Fig. 2.4 : Architecture répartie de Cricket

La configuration présentée comporte trois machines : deux machines clientes et une serveur. Sur chaque station cliente, on retrouve un frontal Cricket chargé de transmettre au serveur les requêtes de connexion à la base (représentée par les trois disques de la machine serveur), de démarrage d'une transaction ou d'allocation mémoire. Le noyau Mach est, quant à lui, chargé de coupler la base dans l'espace d'adressage des tâches clientes et de renvoyer toutes les fautes de pages au serveur.

Le serveur offre donc deux interfaces : une interface de niveau applicatif avec le frontal de chaque station cliente et une interface de gestion mémoire avec les noyaux des différentes stations.

Les points forts de Cricket sont :

- L'utilisation des fonctions de gestion de la mémoire virtuelle offerte par le micro−noyau Mach, et notamment le couplage de la base de données, permet une mise en œuvre efficace du partage et de la persistance de la base.
- La réalisation est modulaire puisque le serveur Cricket ne fait pas partie du noyau (à la différence des partitions Clouds). Elle peut même être considérée comme relativement indépendante du micro−noyau. A titre d'exemple, le micro−noyau Chorus offre une interface de gestion de la

mémoire associée au concept de mappeur qui est très proche de celle de Mach.

- L'isolation de toute les fonctions sensibles du système dans le serveur (ou paginateur) Cricket offre une très grande protection et ne pénalise pas trop les performances.

On retrouve la séparation entre les entités de haut niveau gérées par les langages et le niveau bas, ici des pages, géré par le système. Comme dans Clouds, la gestion de page au niveau du support de la persistance s'impose du fait de l'interface entre le noyau et le paginateur pour la gestion de la mémoire virtuelle.

II.4.4 Le système O2

O_2 [Deux 91] est une base de données doublée d'un système à base d'objets développés au sein du GIP Altaïr puis de O_2 Technology.

La gestion de la base de données est réalisée par la machine O_2Engine. Cette machine fournit les fonctions de base pour accéder et modifier de large quantité de données persistantes, fiables, sûres et partageables. Elle est structurée en trois couches :

- Le niveau supérieur est le Gestionnaire des Schémas de la base. Il est responsable de la création et de la maintenance des classes et du code des applications. Il contrôle aussi la sémantique et la cohérence des schémas.
- Le Gestionnaire des Objets s'occupe de tout ce qui touche au modèle d'objets défini par O_2. C'est lui qui met en œuvre le modèle de persistance du système (le même que Thatte). Il est aussi chargé du ramassage des miettes (c.a.d., les objets qui ne sont plus référencés), de la gestion des index ainsi que des politiques de regroupement des objets.
- Le niveau le plus bas est une extension de WiSS (Wisconsin Storage Server) utilisée comme Gestionnaire de Disque. WiSS offre les structures persistantes suivantes : fichiers séquentiels à enregistrements, index hachés ou a base de B−Tree. Toutes ces structures sont réalisées par des pages. Les pages sont les entités de base pour la persistance.

La machine O_2Engine est mise en œuvre selon une architecture client/serveur dans laquelle le serveur ne gère que des pages et ne connaît pas la sémantique associée aux objets. On appelle une telle architecture ***serveur de page*** [DeWitt 90].

Les stations clientes interagissent avec le serveur en envoyant des requêtes sur des pages de la base. Le serveur retourne les pages au client en assurant le respect du modèle de contrôle de la concurrence. Sur chaque station cliente, les pages sont

temporairement placées dans un cache local. Lorsque le cache est plein, l'arrivée d'une nouvelle page provoque le déchargement d'une vieille page. Celle−ci est renvoyée au serveur qui pourra soit la placer dans son propre cache soit la recopier sur disque. La figure Fig. 2.5 schématise le fonctionnement du serveur O_2Engine.

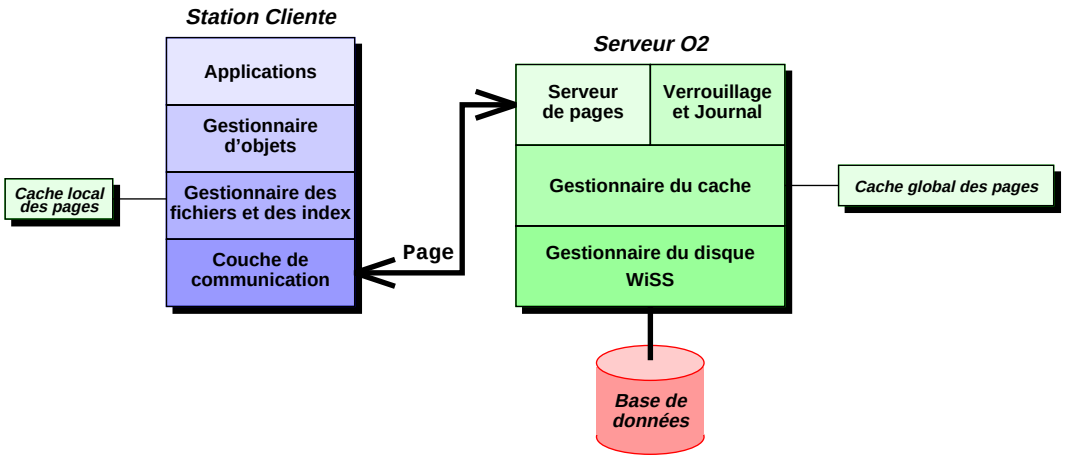

Fig. 2.5 : Le serveur de page O_2Engine

Il peut apparaître inutile de renvoyer une page complète (d'environ 4K) lorsque seul un objet de 100 octets est demandé. Deux faits doivent être pris en compte :

- Le coût CPU de l'envoi de 100 octets est quasiment le même que pour l'envoi de 4000.
- Par ailleurs, si une politique de regroupement efficace des objets est mise en œuvre, le chargement d'une page se traduira par le chargement d'un ensemble d'objets fortement reliés et donc réduira le nombre des requêtes entre les stations et le serveur.

L'avantage principal de cette architecture est dû au fait que la majeure partie du système (et de sa complexité) est placée sur les stations clientes déchargeant par là même le serveur, qui peut alors se concentrer sur les tâches que lui seul peut réaliser. On tend alors vers une exploitation optimale de la puissance CPU disponible sur les stations et le serveur.

Désignation

Les objets et les valeurs sont représentés par des enregistrements persistants alloués par WiSS dans des pages du disque. Les identificateurs des enregistrements sont utilisés comme identificateurs d'objets.

Les identificateurs ont la structure suivante : un identificateur de volume (codé sur 2 octets) suivit d'un identificateur de page dans le volume (4 octets) puis d'un identificateur relatif de l'enregistrement dans la page (2 octets).

Les concepteurs du système ont préféré utiliser des identificateurs physiques afin de supprimer les coûts de recherches et de lectures sur disque dans des tables de correspondances qui sont requises par les identificateurs logiques. De plus, de telles tables deviendraient vite trop grosses et complexes à gérer. On retrouve les mêmes arguments que ceux développés pour le système Thor [Day 92].

Partage

Les applications accèdent aux données de la base et les partagent en utilisant un cache des pages local à chaque station cliente. Les pages sont chargées dans le cache à chaque défaut, en respectant la politique de gestion des accès concurrents.

Le contrôle de la concurrence est réalisé par un protocole hiérarchique de verrouillage à deux phases sur les fichiers et les pages. Un gestionnaire global des verrous s'exécute sur le serveur pendant qu'un gestionnaire local cache les verrous acquis par chaque station (les verrous sont conservés localement tant qu'aucun autre site ne les réclame).

Stockage

L'allocation des pages est réalisée par les couches basses de WiSS au niveau des blocs du disque.

L'intégrité des données est assurée par un algorithme de restauration, basé sur la technique de la journalisation des opérations de modifications. Le journal est un journal "après" conservé par le serveur.

Cependant le système utilise des pages fantômes au niveau des données. Lorsqu'une page modifiée P est éliminée du cache local (pour faire de la place), elle est envoyée au serveur. Dans le cas où celui−ci doit la stocker sur disque, il le fait sur une page fantôme temporaire P'. Si la transaction est annulée, la page P' est détruite. Si la transaction est validée, P' remplace l'ancienne page P. Cette technique est appelée "politique sans vol" car elle n'entraîne pas de restauration de page en cas de panne ou d'arrêt d'une transaction. C'est la même politique qui est employée par Cricket.

Protection

L'architecture client/serveur de O_2 permet d'isoler les fonctions principales du système, notamment l'accès à la base de données, des applications clientes. De plus, la gestion de la concurrence est protégée puisque principalement mise en œuvre dans le serveur.

II.4.5 Tableau récapitulatif

Le tableau suivant donne une vision synthétique des caractéristiques les plus marquantes des différents systèmes décrits dans les sous−sections précédentes.

Système		**Clouds**	**DSR**	**Cricket**	**O_2**
Désignation	Type	Logique	Logique	Physique	Physique
	Entité	Segments	Contextes	Pages	Enregistre-ments
Partage	Type	Couplage en mémoire partagée répartie	Migration des contextes	Couplage en mémoire virtuelle	Cache local des pages
	Entité	Segments	Contextes	Base de données	Base de données
	Concurrence	Moniteurs	Externe	Transactions	Transactions
Stockage	Chargement	A la demande des pages	Complet de contexte	A la demande des pages	A la demande des pages
	Intégrité	Pages fantômes	Non	Pages fantômes	Journal avant et pages fantômes
	Gestion du disque	Fichier	Fichier	Bloc disque	Bloc disque
Protection	Isolation des utilisateurs	Capacité	Non	Non	Non
	Isolation des applications	Espaces d’adressage	Non	Espaces d’adressage	Espaces d’adressage
	Isolation du système	Espace noyau + Serveur	Non	Serveur	Serveur

II.5 Conclusion

Dans ce chapitre, nous avons présenté le modèle de persistance de Thatte afin de définir le cadre conceptuel de notre étude. Par la suite, nous avons étudié différents systèmes offrant des fonctions de support de la persistance selon quatre grands axes. Nous allons, dans cette conclusion, essayer de synthétiser les points forts et les points faibles de ces systèmes. L'objectif est de recenser les caractéristiques du système idéal tant du point de vue des fonctions fournies que de la réalisation.

Le modèle de mémoire persistante de Thatte vise à offrir une vision uniforme des objets temporaires ou persistants aux programmeurs d'applications. Cette vision passe en partie par la définition d'un schéma de localisation des objets, global à l'ensemble du système. On retrouve cette caractéristique dans tous les systèmes présentés et ce, quelle que soit la technique utilisée (désignation logique ou physique).

Il semble qu'au niveau des couches basses du système, la recherche de l'efficacité maximale rende la désignation physique préférable.

La mise en œuvre du partage est réalisée de manière fort différente par les quatre systèmes présentés. Clouds bénéficie de la réalisation sur machine nue pour une intégration efficace de cette fonction à la gestion de la mémoire virtuelle. Cricket utilise l'interface de pagination du micro−noyau Mach pour allier efficacité et modularité. Le DSR, entièrement réalisé au niveau utilisateur, est par contre très pauvre à ce niveau. Paradoxalement, O_2, lui aussi réalisé au niveau utilisateur, est plus riche. Cela est peut−être dû à l'architecture serveur de pages qui définit un cadre de partage plus rigoureux.

L'approche suivie par Cricket, mais aussi par les systèmes Casper et Eos, qui consiste à tirer partie de la technologie micro−noyau pour définir une architecture modulaire et ouverte, tout en étant efficace, nous semble préférable à l'approche intégrée mais rigide de Clouds ou à celle suivie par le DSR et O_2 qui pèche au niveau des performances. Les concepteurs de Cricket ont d'ailleurs comparé l'approche qu'ils ont suivie avec celle qu'ils avaient suivie pour le serveur de page EXODUS [Shekita 90], le gain en performance ne fait aucun doute.

En ce qui concerne la mise en œuvre des fonctions de stockage, le principal problème à résoudre est celui de l'intégrité des données persistantes. En effet, les données sont représentées dans deux espaces : un espace de travail dans lequel elles sont rendues accessibles aux applications et un espace de conservation réalisé par les disques de stockage. Les systèmes fournissent donc des fonctions permettant de

garantir la cohérence relatives des deux espaces. Cricket et O_2 utilisent la technique des pages fantômes pour réaliser la recopie atomique des modifications.

La cohérence d'un ensemble d'objets est généralement mise en œuvre par des mécanismes transactionnels. Sa visibilité au niveau applicatif repose aussi sur le modèle transactionnel.

On peut d'ailleurs remarquer que même les systèmes qui se targuent d'être à un niveau ("one−level−store"), comme Cricket ou Eos, fournissent cette séparation entre espace de travail et espace de conservation.

Au niveau du traitement de la protection, on constate que les systèmes de base de données ne visent qu'à protéger le système et qu'ils ne prennent pas en compte la notion d'utilisateur. C'est là une différence notable avec les systèmes opératoires classiques du type Clouds. On peut cependant remarquer que Cricket et Clouds qui sont les deux systèmes qui utilisent le couplage des données en mémoire virtuelle, sont les seuls capables d'isoler les applications tout en leur permettant de partager des données.

On constate que dans tous les systèmes présentés, les fonctions qui mettent en œuvre la persistance utilisent une notion de bas niveau sémantique qui n'est pas l'objet. Ainsi les partitions Clouds gèrent des segments qui sont des suites d'octets. Le DSR offre des contextes. Alors que Cricket et O_2[(4)] utilisent des pages. Cela est dû à la conjonction de deux facteurs :

- Les noyaux qui supportent ces systèmes ne savent manipuler en mémoire virtuelle que des entités qui ressemblent à des pages. C'est particulièrement vrai pour Clouds et Cricket où l'interface des partitions de Ra et celle de pagination de Mach portent uniquement sur des pages.
- La mise en œuvre de la persistance entraîne des accès au disque (d'au moins 512 octets sur Unix) et ceux−ci sont trop coûteux pour ne lire qu'une partie de chaque bloc. O_2 cherche par exemple à regrouper les objets dans les pages afin de factoriser les lectures et les écritures des objets en un petit nombre d'opérations sur les disques.

De plus, l'objet est considéré comme étant une entité de haut niveau gérée par les langages de programmation et dont la prise en compte au niveau des couches basses du système entraîne une trop grande complexité de celui−ci. Une telle prise en compte ne favorise d'ailleurs pas la modularité et l'efficacité des fonctions mettant en œuvre de la persistance.

---

(4) Bien que WiSS manipule des enregistrements.

Pour résumer, les principales qualités requises d'un système support de la persistance sont :

- La définition d'un modèle de mémoire uniforme permettant à des applications s'exécutant en environnement réparti, de partager des données persistantes.
- La réalisation de ce modèle de mémoire par deux espaces : un espace de travail utilisé par les applications pour modifier les données et un espace de conservation utilisé pour le stockage des données.

  Les transferts de données entre les deux espaces doivent offrir des garanties de cohérence et d'efficacité.
- L'accès aux fonctions mettant en œuvre la persistance doit se faire au travers d'une interface de bas niveau portant sur des entités dont la seule caractéristique est la persistance. Cela permet d'offrir de l'efficacité et une grande modularité lors de la réalisation.

La mise en œuvre d'un modèle de protection est plus particulièrement requise dans le cadre d'un système opératoire classique supportant un grand nombre d'applications différentes.

Le chapitre IV va donc essayer de présenter les principes de conception et de réalisation du support de la persistance dans le système Guide−2, au travers du cadre que nous venons de définir. Nous essayerons tout particulièrement de mesurer l'adéquation de notre réalisation à la définition du système idéal que nous venons de décrire.

Auparavant, dans le chapitre III, nous allons donner un aperçu du système Guide−2 et de l'environnement Mach qui ont servis de support à la réalisation de notre travail.

Chapitre III

L'environnement Guide

Ce chapitre présente les principaux choix de conception de la nouvelle version du système Guide, désignée sous le nom Guide−2. Cette nouvelle conception avait pour objectif de capitaliser l'expérience acquise dans le cadre du projet Guide−1 ainsi que de prendre en compte les récentes évolutions technologiques du domaine. On peut notamment citer parmi ces évolutions l'arrivée des micro−noyaux tels que Chorus [Rozier 88] et Mach [Accetta 86] utilisés comme plate−forme de mise en œuvre de sous−systèmes d'exploitation.

Ce chapitre n'a pas pour objet de présenter un historique du projet Guide et de son évolution depuis la réalisation du système Guide−1. On se reportera à [Rousset 93] pour une description des améliorations apportés par cette nouvelle version. Nous cherchons principalement à définir le cadre dans lequel s'est inscrit notre travail.

Ce chapitre se découpe en trois parties. La section III.1 présente les principales structures de base du système que sont le modèle de données, le modèle d'exécution ainsi que le modèle de mémoire. L'architecture générale de la machine virtuelle Guide−2 est décrite dans la section III.2. Le chapitre se termine en section III.3 par une description succincte du micro−noyau Mach 3.0 et des mécanismes de bases ayant servi au développement de Guide−2.

III.1 Introduction au système Guide

Cette section résume les principaux choix qui sont à la base de la conception du système Guide−2.

III.1.1 La machine virtuelle Guide

Le système Guide−2 est destiné à fournir une "machine virtuelle", pour le développement et l'exécution d'applications structurées selon un modèle à objets. Ces applications doivent pouvoir être écrites dans différents langages à objets. En conséquence, la machine virtuelle Guide−2 a un caractère générique : elle définit une organisation des données et un schéma d'exécution accessibles à travers différents langages. Néanmoins, chaque langage peut également gérer ses propres structures d'exécution au moyen d'un environnement d'exécution ("run−time") spécifique. La

machine virtuelle Guide fournit un support pour la gestion de ces structures spécifiques mais le détail de leur organisation n'est pas visible dans cette machine. Deux langages font l'objet d'une attention particulière : le langage Guide, qui sert de base à la définition de la machine virtuelle, et le langage C++, essentiellement pris en compte en raison de sa grande diffusion. La figure Fig. 3.1 schématise cette organisation générale.

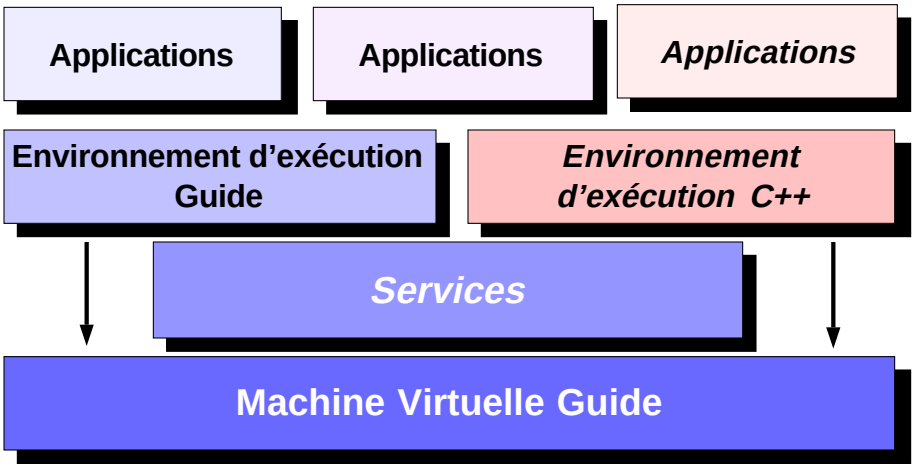

Fig. 3.1 : Organisation générale du système Guide−2

Outre les fonctions fournies par la machine virtuelle Guide et par les environnements d'exécution des langages supportés, les applications peuvent utiliser divers services tels qu'un serveur de désignation des objets ou un interpréteur primitif d'un langage de commandes. Ces services peuvent être réalisés dans le langage Guide bien que certaines fonctions spécifiques soient réalisées directement au−dessus de la machine virtuelle. Du point de vue de la structure, il n'y a pas de différence entre les applications et les services, les services étant, comme les applications, organisés selon un modèle à objets. La principale différence entre applications et services réside dans le fait que les services sont gérés par l'administration du système, qui définit leur protection ainsi que leur modalité de partage, de désignation et d'accès.

III.1.2 Le modèle de données

Nous résumons dans cette section les mécanismes nécessaires au support des langages utilisés. Nous nous limitons donc aux caractéristiques des langages qui sont prises en compte au niveau du système.

La machine virtuelle gère des entités appelées ***objets*** et des entités appelées ***classes***. Tout objet est un exemplaire (ou instance) d'une classe. Une classe définit la structure interne de ses instances et le programme des opérations (ou méthodes) qui leurs sont applicables. Une classe fonctionne comme un générateur pour ses instances. Le système maintient un lien explicite entre chaque objet et sa classe.

Un objet peut être ***persistant*** c'est à dire avoir une durée de vie indépendante de celle de l'activité qui le crée. Les objets persistants sont conservés dans une mémoire de stockage distribuée (cf III.1.4).

Objets et classes sont désignés par une ***référence***, c'est à dire une information qui permet au système de les identifier et de les retrouver. Le système fournit plusieurs sortes de références qui diffèrent par leur portée et leur durée de vie, ce qui permet de gérer des objets connus dans des environnements différents et ayant eux−mêmes différentes durées de vie. En particulier, les références universelles ou ***références systèmes*** (***sysref***) permettent de désigner les objets persistants.

III.1.3 Le modèle d'exécution

Le modèle d'exécution de Guide définit deux entités : les domaines et les activités. Un ***domaine*** est un espace d'adressage multi−sites, dans lequel des objets peuvent être rendus accessibles. L'ensemble des sites sur lesquels est représenté un domaine et l'ensemble des objets qu'il contient peuvent évoluer dynamiquement. Dans un domaine s'exécutent des ***activités***, flots séquentiels et distribués d'exécution. Concrètement, une application est représentée par un ou éventuellement plusieurs domaines.

L'espace d'adressage d'un domaine est constitué d'un ensemble de contextes. Un ***contexte*** est un espace d'adressage homogène (une mémoire virtuelle) local à un site. Cette décomposition est motivée par la répartition (un domaine actif sur plusieurs sites doit avoir au moins un contexte sur chaque site) et également par des considérations de protection qui sont développées en III.1.5.

La notion classique de processus du type Unix est réalisée par un domaine constitué d'un seul contexte dans lequel s'exécute une seule et unique activité.

Une des principales caractéristiques de Guide est le choix d'un ***modèle à objets passifs***, c'est à dire qu'il n'existe pas d'entité d'exécution, telle qu'une activité, attachée de manière spécifique à un objet. Le déroulement d'une activité est une suite synchrone d'appels de méthodes sur des objets. On se reportera à [Freyssinet 91] pour une définition plus rigoureuse du modèle à objets passifs de Guide.

Ce choix a un impact majeur en ce qui concerne la nature de l'information conservée dans l'espace de persistance. En effet, en séparant complètement les objets des entités d'exécution, il ne sera pas nécessaire de conserver l'état de ces entités (pile, registres, etc) pour accéder ultérieurement à l'objet. Ainsi, à la différence du Distributed Shared Repository (cf sous−section II.4.2), la gestion de la persistance s'en trouve grandement simplifiée. Notons tout de même que la conservation des structures d'exécution sera nécessaire au support d'un modèle d'exécution tolérant les pannes.

Une autre caractéristique importante du système a trait à la communication entre activités. Cette communication entre activités d'un même domaine ou de domaines différents, se fait uniquement par ***partage d'objets***. Le système doit donc fournir des mécanismes permettant la mise en œuvre de ce partage. De plus, il doit offrir des fonctions permettant de contrôler ce partage et de réaliser diverses politiques d'exécution : migration, répartition de charge. Le système fournit cependant une politique par défaut (cf sous−section IV.1.1.4).

III.1.4 Le modèle de mémoire

Le modèle de mémoire détermine les caractéristiques de l'espace d'exécution et de conservation des objets. La mémoire du système Guide constitue une réalisation du modèle de mémoire persistante de Thatte décrit dans la section II.2.

Dans le modèle d'exécution défini précédemment, les objets utilisés par les activités sont considérés comme faisant partie d'un espace unique que nous appelons l'***espace d'exécution***. Cet espace est vu globalement comme une mémoire commune (uniforme au sens de Thatte). Le support de cette mémoire est l'ensemble des mémoires principales des stations de travail et les parties des disques de ces stations qui sont dédiées à la pagination.

Cet espace est essentiellement un espace de travail. Il ne fournit aucune garantie de conservation de longue durée pour les informations qui y figurent, même si une partie de ces informations peut avoir un disque comme support. Il doit donc être complété par un espace de conservation permanente que nous appelons ***espace de stockage***. La figure Fig. 3.2 schématise ce modèle de mémoire uniforme (vis vis de la désignation) à deux niveaux (quand à sa structuration) :

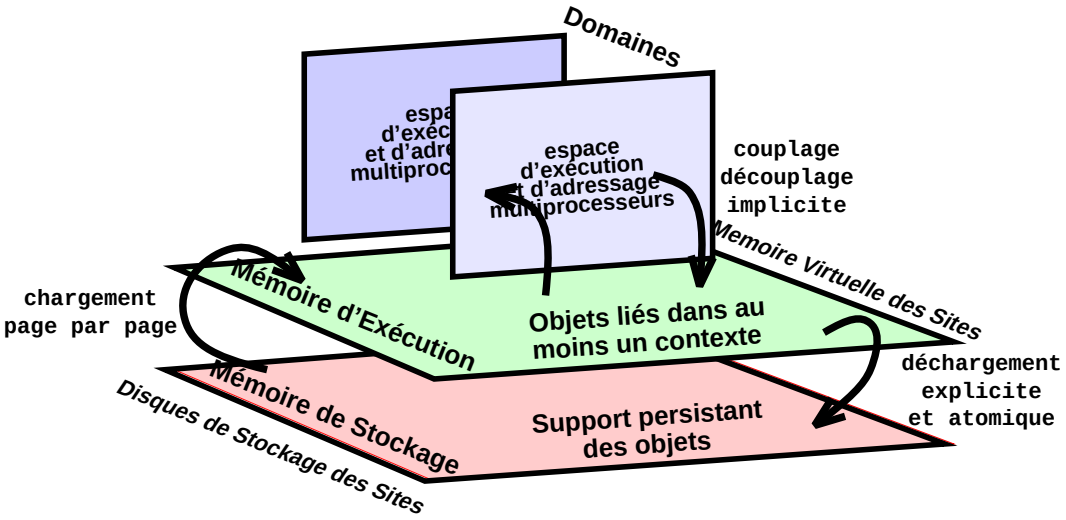

Fig. 3.2 : Une mémoire persistante et uniforme à deux niveaux

Cette figure appelle la remarque suivante : bien que l'interface de chargement/déchargement soit de type explicite, le chargement d'un objet en mémoire d'exécution est implicitement réalisé par le système au premier accès à cet objet par une application. Le déchargement est par contre effectivement déclenché par l'application en des points de cohérence qu'elle seule peut définir (par exemple, à la fin d'une transaction).

Il existe plusieurs façons de voir ce modèle de mémoire :

- Du point de vue des applications, le couplage et le chargement des objets sont implicites (réalisés par le système lors des accès aux objets) alors que le déchargement est explicite (déclenché par l'application en des points de cohérence qu'elle définit).
- Au niveau de la gestion de la mémoire d'exécution, le couplage et le déchargement sont explicites alors que le chargement est réalisé implicitement (transfert page par page).
- L'interface d'accès à la mémoire de stockage est de type chargement/déchargement explicite des pages.

La séparation entre les deux espaces est due à plusieurs motivations :

- La garantie de ***cohérence*** ; le système garantit que l'image en mémoire de stockage est cohérente par rapport à celle en mémoire d'exécution.
- La ***modularité***, qui est la possibilité de réutiliser des serveurs de stockage existants tels que AFS ou Pegasus [Leslie 93]. La réutilisation de produits existants se trouve facilitée par ce découpage entre les deux niveaux de mémoires.
- La ***sécurité***, qui est la possibilité de contrôler l'accès à l'interface de l'espace de stockage au niveau des serveurs de stockage.
- La spécification de ***qualité de service***, c'est à dire que l'utilisation de différents serveurs de stockage nous fourni cette fonction supplémentaire.

Cette séparation n'avait pas été explicitement définie dans le premier prototype du système Guide [Balter 91]. Cette nouvelle structuration nous a amené de fait à concevoir une architecture de système plus modulaire (en interne) et plus ouverte (vers l'extérieur).

III.1.5 Protection et sécurité

Les entités auxquelles sont associées la protection et la sécurité sont les ***usagers*** ou les ***groupes d'usagers*** du système. A tout objet du système est associé un usager appelé son ***propriétaire***.

La sécurité est réalisée par deux mécanismes complémentaires :

- La ***séparation des contextes***. L'exécution dans des contextes différents fournit un mécanisme de sécurité et de protection permettant d'isoler les usagers ou les programmes d'un même usager les uns des autres.
- ***L'authentification***. Cette procédure peut être appliquée lors de l'accès à certains serveurs de stockage utilisés pour la mise en œuvre du service de stockage et lors de la procédure de connexion d'un usager au système.

La protection s'appuie sur les concepts de vue et de groupe. Une ***vue*** est un ensemble de méthodes applicables sur un objet (sous ensemble des méthodes définies par la classe de l'objet). La protection est réalisée par deux mécanismes :

- La protection par ***liste d'accès***. Ce mode de protection est adapté à la programmation par objet de la manière suivante : un élément de liste d'accès associe une vue de l'objet à un usager ou à un groupe.
- Le contrôle de la visibilité des objets. A chaque objet est associé un attribut, appelé ***attribut de visibilité***, qui indique s'il peut ou non être accédé depuis un objet appartenant à un autre usager.

On pourra trouver une description détaillée du traitement de la protection au niveau des objets et des utilisateurs dans [Hagimont 93].

III.1.6 Les services du système Guide

L'espace de stockage est constitué de plusieurs parties disjointes correspondant à des services différents. Typiquement, on peut en distinguer deux que nous appelons le ***service de stockage fiable*** (SSF) et le ***service de stockage rapide*** (SSR) pour mettre l'accent sur leurs caractéristiques principales. Le SSF et le SSR se différencient par la qualité de service qu'ils assurent :

- Le SSF doit assurer aux objets une haute disponibilité et une bonne robustesse. Le SSF est réalisé par le serveur spécialisé Goofy géré par l'administrateur du système. Sa description détaillée fait l'objet du chapitre VII.

- Le SSR fournit des garanties limitées de disponibilité et de sécurité. Il peut être réalisé sur les disques attachés aux stations de travail et administré tel quel. Le SSR est directement intégré à Guide (toutes les stations qui le réalisent, possèdent nécessairement un noyau Guide).

Un système Guide est réalisé comme un assemblage de cellules. Dans la pratique, une cellule est physiquement réalisée par un réseau local et comporte de l'ordre de quelques dizaines de machines. La cellule est l'unité d'administration [Duda 93]. La réalisation des fonctions d'administration (gestion des machines, des usagers, des données et des services) nécessite de pouvoir désigner ces entités. Cette fonction est assurée par le service de désignation symbolique décrit dans [Layaida 93].

III.2 Architecture générale

La machine virtuelle Guide−2 qui réalise les modèles de données, d'exécution et de mémoire est mise en œuvre par une hiérarchie de machines (cf Fig. 3.3) dont les composantes sont brièvement présentées dans la présente section.

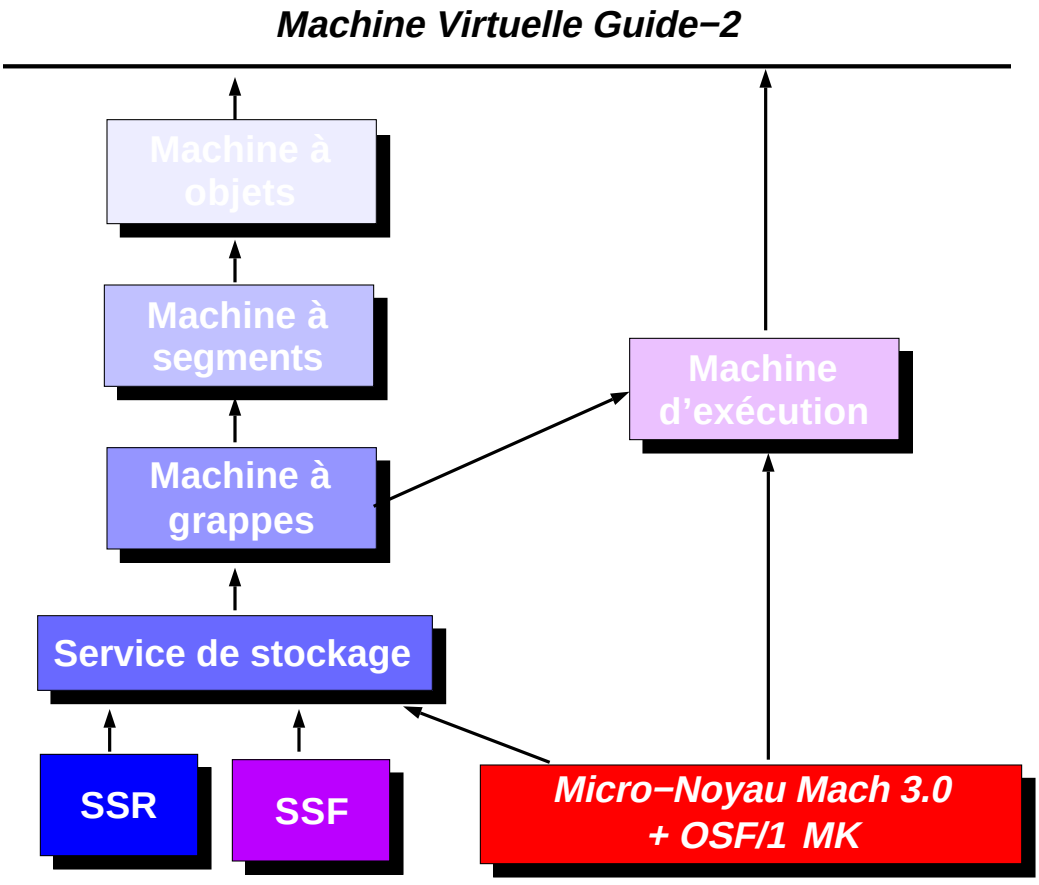

Fig. 3.3 : La machine virtuelle Guide−2

III.2.1 Machine à objets

C'est le nom donné à la machine qui met en œuvre le modèle d'objet décrit en III.1.2. Ce modèle d'objet ne définit pas de modèle d'héritage particulier pour les classes, cependant la machine à objets offre le concept de ***bibliothèque de méthodes*** qui permet la mise en œuvre de plusieurs modèles d'héritage.

Les problèmes posés par le partage d'objets entre des contextes qui peuvent être mis en œuvre sur des sites différents, ainsi que la volonté de ne pas avoir à recalculer le chemin d'accès à un objet à chaque appel de méthode sur cet objet, nous ont amenés à mettre en œuvre la machine à objets sur une autre machine, dite machine à segments, offrant un adressage segmenté. L'interface de la machine à objets et sa mise en œuvre sur la machine à segments sont décrites en détails dans [Hagimont 93].

III.2.2 Machine à segments

Un ***segment*** est une unité d'adressage logique constituée d'un ensemble d'informations contiguës. Une machine à adressage segmenté permet d'adresser directement les informations contenues dans les segments. Une telle machine permet de résoudre élégamment les problèmes posés par le partage d'objets entre des contextes différents. Nous ne disposons pas de cette machine, mais nous fournissons une machine abstraite réalisant des segments.

Un segment est rendu accessible dynamiquement par couplage dans un contexte lors de la première tentative d'accès à l'objet qu'il contient. On appelle ***couplage*** la mise en correspondance d'un segment avec une portion d'un contexte. A l'intérieur d'un contexte, un segment couplé est désigné par une adresse virtuelle.

La machine à segments offre un mécanisme de liaison dynamique au premier accès utilisant la méthode du segment de liaison [Organick 72]. Un segment possède un segment de liaison dans chaque contexte où il est couplé. Une entrée du segment de liaison est associée à la compilation à chaque référence vers un segment externe ; après ***liaison*** cette entrée est remplacée par l'adresse du segment lié.

III.2.3 Machine à grappes

Notre expérience avec le premier prototype du système Guide nous a montré que la plupart des objets étaient petits (moins de 300 octets, cf [Freyssinet 91]) comparés à l'unité de couplage (au moins une page de 4096 octets sur Mach). Se servir de l'objet comme unité de partage aurait donc eu deux inconvénients majeurs :

- Chaque défaut d'objet se serait traduit par une opération de couplage, qui est relativement coûteuse en temps d'exécution et en ressources (ports).
- La mémoire virtuelle des tâches aurait été fragmentée (puisque seuls les 300 premiers octets de chaque page auraient été utilisés).

La mise en œuvre de la machine à segments et surtout de la persistance des entités gérées par cette machine (les segments persistants qui représentent des objets), nous a donc conduit à définir de nouvelles entités, les grappes qui regroupent les segments.

Une ***grappe*** est une unité d'adressage logique constituée d'un ensemble d'informations contiguës. La grappe est l'unité de couplage dans un contexte, c'est aussi l'unité de structuration de la mémoire de stockage.

Si le regroupement d'objets peut être utilisé efficacement par les niveaux supérieurs du système (machine à objets ou langages) pour grouper dans une grappe des objets logiquement reliés, le couplage d'un objet impliquera alors le couplage du contexte de travail ("working set") de cet objet. De plus, en groupant plusieurs objets dans une même page, la lecture d'un objet induira la lecture des objets voisins, ce qui réduira le nombre d'accès au disque. Dans la mesure où les opérations d'entrée/sortie sur disque sont extrêmement coûteuses, cela améliorera les performances globales du système.

III.2.4 Service de stockage

L'unité physique de stockage est le ***volume***. Le volume est assimilé à une partition du disque. Afin d'être rendu accessible, il doit être monté par une station de travail (il s'agit d'une opération logique d'attachement par le système de la station).

Les services de stockage en charge de la gestion des volumes ont été présentés dans la sous−section III.1.6.

III.2.5 Machine d'exécution

La machine d'exécution fournit aux utilisateurs les structures nécessaires à l'exécution d'une application : création de domaines et d'activités, contrôle du parallélisme, synchronisation, mécanisme d'appel de procédure à distance, etc ...

III.3 Quelques caractéristiques de Mach 3.0

Le but de cette section est de décrire les caractéristiques de Mach 3.0 qui sont essentielles pour la réalisation du support de la persistance dans notre noyau.

Cette description est composée de trois parties :

- Les structures d'exécution : les *tâches* et les *threads*[(2)].
- Les communications fondées sur la notion de *port*.
- La gestion de la mémoire virtuelle à l'aide de *paginateurs externes*.

Le lecteur intéressé se reportera à [Lopere 92a] et [Lopere 92b] pour une description détaillée des fonctions fournies par le micro−noyau Mach.

III.3.1 Les tâches et les flots de contrôles

La ***tâche*** est l'unité de structuration du système. C'est un environnement d'exécution pour les flots de contrôles. Elle comprend un espace virtuel d'adressage privé et un ensemble de droits d'accès à des ressources du système (par exemple les ports). Une tâche est locale à un site.

Le ***flot de contrôle*** est l'unité de base d'allocation du processeur. Un flot de contrôle s'exécute dans l'espace virtuel d'une tâche et tous les flots de contrôles d'une tâche partagent ses ressources.

La notion de processus dans un système Unix est équivalente à une tâche avec un unique flot de contrôle.

III.3.2 Les communications

Les communications dans Mach sont fondées sur les notions de port et de message :

- Un ***message*** est une suite typée de données adressées à un port.
- Un ***port*** est une boîte aux lettres, c'est à dire une file de messages, à laquelle les messages peuvent être envoyés. A un instant donné, une et une seule tâche possède le droit de lecture sur un port, les autres tâches peuvent obtenir des droits d'émission de messages sur ce port. La tâche qui envoie un message sur un port peut s'exécuter sur un site différent de celui de la tâche qui possède le droit de lecture sur ce port.

Les ports sont aussi utilisés pour désigner les ressources dans Mach (les tâches et les flots de contrôles notamment).

Les ports dans Mach sont protégés. Le noyau contrôle la transmission des ports dans les messages (qui sont typés) entre les tâches. Un port est désigné, dans une tâche ayant des droits sur ce port, par un nom local à cette tâche. Lorsque ce nom local est utilisé dans un message pour transmettre des droits sur ce port à une autre tâche, ce nom local est traduit par le noyau, et la tâche réceptrice désigne ce port par un autre nom local privé alloué par le noyau. Le noyau Mach gère pour chaque tâche la correspondance entre ses noms locaux et les ports. Ainsi, une tâche ne peut pas s'allouer

(2) que nous appellerons dans la suite flots de contrôle

sans contrôle des droits sur un port. De même, il n'est pas possible de partager un nom de port entre deux tâches dans de la mémoire partagée. Une tâche ne peut utiliser que des droits qui lui ont été explicitement transmis.

III.3.3 La gestion de la mémoire virtuelle

Une des caractéristiques principales de Mach est qu'il offre la possibilité à une tâche utilisateur (par exemple, un processus Unix) de contrôler la gestion de la mémoire virtuelle. L'intérêt majeur tient dans la souplesse de choix des politiques de gestion de la mémoire ainsi rendue disponible au concepteur de sous−systèmes d'exploitation.

Pour ce faire, Mach définit la notion de ***memory object***(3) qui est un support de données dont l'accès et la gestion sont à la charge d'un serveur externe au noyau. Un tel serveur est appelé ***paginateur externe***. Un paginateur externe assure la pagination (le va et vient en mémoire centrale) des données d'un segment, depuis et vers les disques de stockage.

Les applications, qui sont réalisées par des tâches, peuvent associer (coupler) une région de leur espace d'adressage à un segment en utilisant l'appel noyau **`vm_map`**. Par la suite, toute faute de page (en lecture ou en écriture) sur la région couplée se traduira par un envoi de requête de la part du noyau au paginateur externe responsable du segment en question. Le noyau et les paginateurs externes coopèrent au travers de l'interface de pagination décrite dans [Lopere 92b].

Les tâches peuvent partager physiquement des données en couplant le même segment. Un segment partagé est créé par le paginateur et il est désigné par un port appartenant au paginateur. Lorsque le couplage d'un segment dans une tâche est demandé, le port désignant ce segment doit être spécifié. Une région est alors allouée dans l'espace virtuel de la tâche. L'adresse virtuelle de cette région peut être spécifiée lors de la demande de couplage, ou allouée par le noyau Mach dans une zone libre de l'espace virtuel de la tâche.

A chaque faute de page dans cette région, un message est envoyé par le noyau au paginateur sur le port identifiant le segment couplé. Le paginateur est alors chargé de répondre à cette faute de page en retournant la page au noyau. De même, si cette page doit être vidée de la mémoire centrale, elle est retournée par le noyau au paginateur. Ce principe est illustré sur la figure Fig. 3.4.

---

(3) Par la suite, nous utiliserons le terme segment pour désigner un memory object.

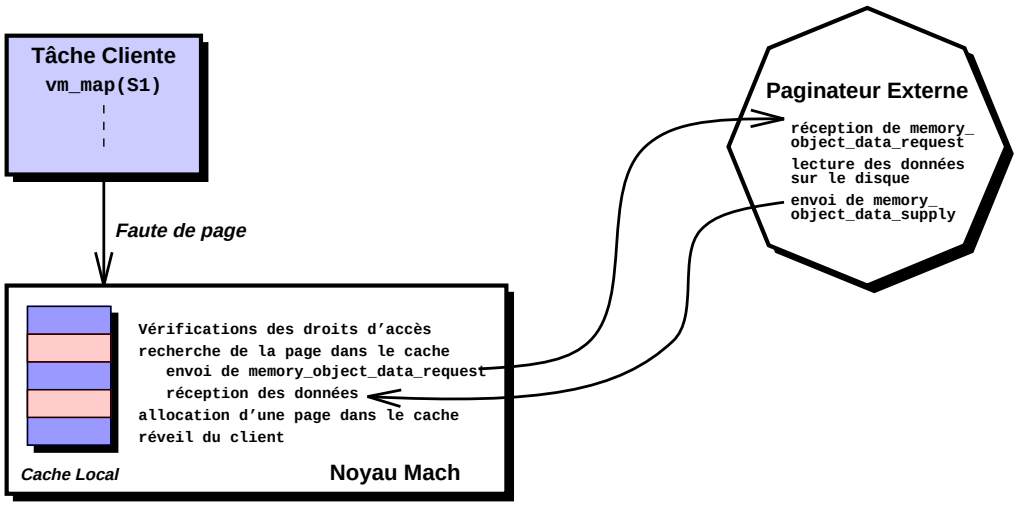

Fig. 3.4 : Traitement d'une faute de page

A chaque segment auquel on accéde sur le site, le noyau associe un cache local constitué des pages de mémoire physique contenant des portions de ce segment. Les défauts de page engendrés par l'accès aux portions inaccessibles d'un segment sont traités par le noyau. Celui−ci invoque le paginateur (par la requête `memory_object_data_request`) et remplit le cache local avec les données reçues (`memory_object_data_supply`).

Notons que la gestion de la mémoire centrale (qui peut être vue comme un cache local de mémoire par le paginateur) est à la charge du seul noyau. Celui−ci met en œuvre une politique classique de remplacement des pages basée sur le modèle LRU (Least Recently Used). Le paginateur ne peut pas diriger la politique de remplacement des pages dans la mémoire. Il peut cependant l'influencer en invalidant à l'avance certaines pages, néanmoins son impact reste limité.

Plusieurs tâches s'exécutant sur des sites différents peuvent coupler le même segment partagé. Le paginateur peut alors recevoir des fautes de pages provenant de plusieurs noyaux Mach. Il doit alors assurer la cohérence des données partagées, en limitant le nombre de copies de pages et les droits sur ces copies (lecture/écriture), donnés aux noyaux. Pour assurer la disponibilité des pages, le paginateur peut réclamer la copie d'une page à un noyau, ou demander que les droits d'accès à cette page soient modifiés.

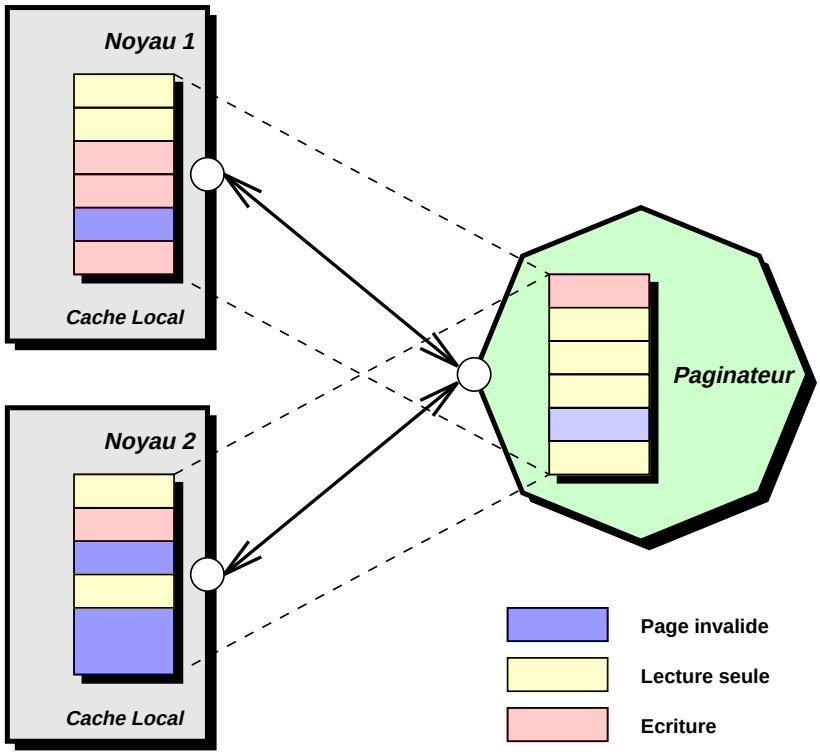

Fig. 3.5 : Partage réparti de mémoire

> Un noyau réclame une page en envoyant au paginateur la requête `memory_object_data_request`. Le paginateur lui retourne la page avec les droits associés par `memory_object_data_supply`.
>
> Si le paginateur a besoin d'invalider ou de réduire les droits d'un noyau sur une page, il lui envoi la requête `memory_object_lock_request`. Le noyau peut alors soit retourner la page modifiée par `memory_object_data_return`, soit indiquer qu'il a correctement réduit ses droits par `memory_object_lock_completed`.

III.4 Conclusion

Ce chapitre a présenté la machine virtuelle Guide−2 dont l'objectif est de servir de support au développement et à l'exécution d'applications réparties structurées selon un modèle à objets.

Les objets gérés par la machine virtuelle ont la propriété d'être persistants et passifs. Ils sont de plus désignés par des références universelles qui permettent de les localiser indépendamment de leur localisation physique à un instant donné.

Du point de vue du support de la persistance, on s'intéressera principalement au cours du chapitre suivant au modèle de mémoire défini par la machine virtuelle Guide−2 . Ce modèle, qui est un des aspects fondamentaux du système, se caractérise par :

La définition d'un modèle de mémoire uniforme vu de l'utilisateur qui assure notamment une transparence vis à vis de la répartition et de la persistance. En revanche, notre mémoire est volontairement structurée en deux niveaux :

- La ***mémoire d'exécution*** réalise l'espace où sont couplés dynamiquement les objets auxquels accèdent les applications. C'est à ce niveau que sont gérés les problèmes induits par la répartition.
- La ***mémoire de stockage*** assure quand à elle la gestion de la persistance des objets. Elle peut être réalisée par des services de stockage externes au système Guide.

La conception de l'architecture de la machine Guide−2 a été essentiellement guidée par un objectif de modularité qui s'est traduit par l'adoption de la technologie micro−noyau (en l'occurrence Mach 3.0) comme support de notre système mais aussi par la décomposition de notre machine virtuelle en une hiérarchie de machines : machine à objets, machine à segments et machine à grappes.

La plupart des micro−noyaux disponibles sur le marché et notamment les plus représentatifs d'entre eux, Chorus et Mach, offrent aux concepteurs de sous−systèmes opératoires la possibilité de participer à la gestion de la mémoire virtuelle au travers d'une interface de pagination externe.

Il est alors envisageable d'intégrer des mécanismes de gestion de la persistance à la gestion de la mémoire virtuelle, comme cela est réalisé dans le système Clouds, sans être obligé de modifier les couches basses du noyau. Nous verrons dans le chapitre suivant comment cette possibilité est exploitée pour la mise en œuvre du modèle de mémoire de Guide−2.

Chapitre IV

Le support de la persistance dans Guide

Ce chapitre présente les principes de réalisation du support de la persistance dans le système Guide−2. La caractéristique principale de notre réalisation a trait à sa nature répartie. En effet, l'objectif général du projet Guide est de concevoir et de réaliser un système d'exploitation réparti, opérant sur un ensemble de stations de travail reliées par un réseau local. Le support de la persistance doit donc à la fois s'intégrer et utiliser au mieux la nature répartie de l'environnement. C'est pourquoi, nous avons, dès le début de nos réflexions, écarté une solution centralisée du type serveur de pages et nous lui avons préféré une architecture multi−serveurs.

Le support de la persistance est réalisé par un ensemble de serveurs, appelés gérants de mémoire. Les gérants de mémoire sont présents sur tout ou partie des sites qui supportent le système Guide. Chaque gérant a la charge d'une partie de l'espace de persistance et ne communique pas nécessairement avec les autres.

Par ailleurs, nous avons cherché à définir un système, modulaire quant à son organisation interne et ouvert pour ce qui est de ses relations avec d'autres sous−systèmes d'exploitation. Cette considération a renforcé le besoin du découpage de la mémoire en une mémoire d'exécution et une mémoire de stockage (cf sous−section III.1.4).

Au niveau du support de la persistance proprement dit, l'approche que nous avons suivie tend à remonter le plus possible la gestion des objets dans les couches hautes du système. Cela concerne les aspects structurels mais aussi sémantiques de cette gestion Cette volonté combinée avec les constatations effectuées dans le cadre de la première version du système Guide (cf sous−section III.2.3), nous ont conduit à l'utilisation d'une technique de regroupement des objets basée sur la notion de grappe. La grappe est la seule entité manipulée par les couches basses du système qui mette en œuvre les fonctions de persistance. Dans la suite de ce chapitre, nous ne nous intéressons qu'au seul support de la persistance des grappes.

Nous présentons tout d'abord, dans la section IV.1, les différents niveaux d'abstraction dans lesquels se décompose le support de la persistance, ainsi que l'architecture générale qui en découle .

La section IV.2, plus technique, aborde les problèmes de mise en œuvre de quelques fonctions de base telles que la gestion de la pagination et la mise en œuvre de la journalisation des opérations de stockage. Elle commence par une description des structures d'exécution qui mettent en œuvre le support réparti de la persistance.

IV.1 Architecture générale

Une ***grappe*** est un ensemble non typé de pages (le système n'associe aucune sémantique particulière à son contenu). Elle définit l'unité de couplage. Une grappe est couplée dans l'espace d'adressage d'une tâche lorsqu'un de ses objets est utilisé pour la première fois dans cet espace. La grappe est une notion similaire aux segments Clouds et aux contextes du DSR (cf section II.4).

> Les grappes sont gérées par une machine appelée ***machine à grappes*** qui a pour fonction essentielle de permettre la mise en œuvre du partage et de la persistance de grappes.

La machine à grappes fournie des primitives de couplage/découplage des grappes, aux machines mises en œuvres dans les couches supérieures du système.

La grappe offre une excellente base sur laquelle nous pouvons construire une architecture ouverte dans la mesure où les opérations de stockage d'une grappe s'apparentent à celles effectuées sur des fichiers (la machine à grappes opère sur des pages comme on opère sur des blocs de fichiers). On peut donc très facilement utiliser des serveurs de stockage externes possédant une interface de type fichier. De plus, la notion de grappe n'implique pas de dépendance envers le modèle d'objets, ce qui simplifie d'autant la réalisation de la machine à grappes.

Afin d'accroître d'avantage la modularité de la machine, nous avons clairement séparé les aspects relatifs à la gestion de la mémoire d'exécution de ceux relatifs à la gestion de la persistance. L'architecture de la machine à grappe reflète ce double rôle : le niveau bas de la machine gère le stockage des grappes sur disques (ce qui correspond à la gestion des grappes en mémoire de stockage) alors que le niveau haut gère le partage de grappes entre applications (c'est à dire la gestion des grappes en mémoire d'exécution). La figure Fig. 4.1 donne une première vision de la machine à grappes.

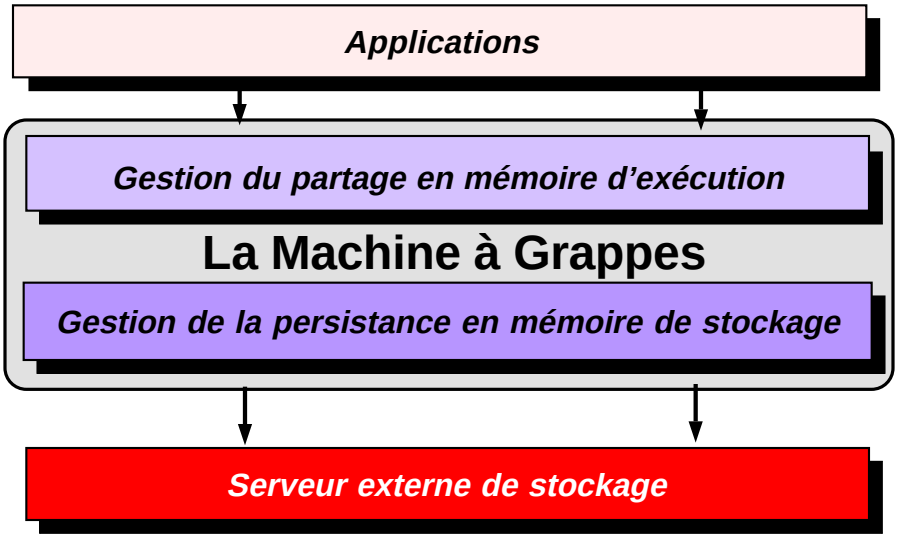

Fig. 4.1 : Vue générale de la machine à grappes

> Dans la pratique, la couche application est réalisée par les machines à objets et à segments de la figure Fig. 3.3. L'interface fournie par la machine à grappes contient les primitives de couplage/découplage des grappes, mais aussi des fonctions permettant de faire varier la taille des grappes à l'exécution. On se reportera en IV.1.1 pour une description détaillée de cette interface.

L'organisation de chacune des sous−machines est décrite respectivement en sous−sections IV.1.1 et IV.1.2.

IV.1.1 Gestion de grappes en mémoire d'exécution

IV.1.1.1 Organisation générale

Les micro−noyaux tels que Chorus [Rozier 88] et Mach [Accetta 86] permettent la mise en œuvre de gérants de mémoire, externes au noyau, appelés paginateurs externes (cf sous−section III.3.3). Un gérant de mémoire est un processus utilisateur qui permet le couplage par d'autres processus d'une région de mémoire identifiée par un port. Du point de vue du micro−noyau du site sur lequel la région a été couplée, le gérant de mémoire est responsable du traitement des fautes de pages qui peuvent survenir sur cette région de mémoire (cf Fig. 3.5).

Nous avons utilisé cette possibilité pour la mise en œuvre de la machine à grappes. Les grappes sont gérées par un ensemble de ***gérants de mémoire*** répartis sur l'ensemble des machines du réseau. La machine à grappes est constituée de la réunion de ses gérants de mémoire et d'une partie cliente, ou ***composant d'interface***, liée avec le code des applications.

Les grappes sont stockées dans des ***volumes*** (cf sous−section IV.1.2) qui peuvent être vus comme des partitions des disques. De manière similaire à une partition, un volume doit être monté pour être rendu accessible. L'opération de ***montage***(1) du volume en mémoire d'exécution est une opération d'administration du système qui est définie de manière statique par l'administrateur.

Pour être accessible, une grappe doit donc appartenir à un volume monté, c'est à dire à un volume dont le gérant de mémoire est disponible (actif). D'autre part, la localisation de la grappe implique la localisation du gérant de mémoire qui en a la charge. Ce problème est traité plus en avant dans la sous−section suivante.

Les grappes sont couplées dans les contextes à la demande des applications (machine à segments). Une requête de couplage de grappe est envoyée par message au gérant de mémoire du site sur lequel est monté le volume qui la contient (cf sous−section suivante). La grappe est alors couplée dans l'espace d'adressage de l'application à une adresse quelconque choisie par le micro−noyau(2).

Le gérant mémoire est alors responsable de la pagination de la grappe. Il répond donc aux fautes de page sur cette grappe. Les données de la grappe sont lues en mémoire de stockage et données au noyau Mach. Lorsque le noyau Mach envoie au gérant de mémoire une page pour faire de la place en mémoire centrale, cette page est stockée dans une zone temporaire (***partition de pagination***). Une grappe est recopiée dans la mémoire de stockage que lorsqu'elle n'est plus couplée dans aucun des contextes des tâches du site, ou lorsqu'une des tâches qui l'utilise le demande explicitement. Cette politique permet de conserver en mémoire de stockage, et durant toute l'exécution, un état cohérent de la grappe. La sous−section IV.2.3 aborde le problème du traitement de cette re−copie.

La figure Fig. 4.2 décrit l'architecture logicielle de la machine à grappes.

(1) Le terme montage vient de l'analogie avec la primitive `mount()` utilisée par le système Unix.

(2) L'interface autorise la couche applicative à forcer le couplage à une adresse particulière.

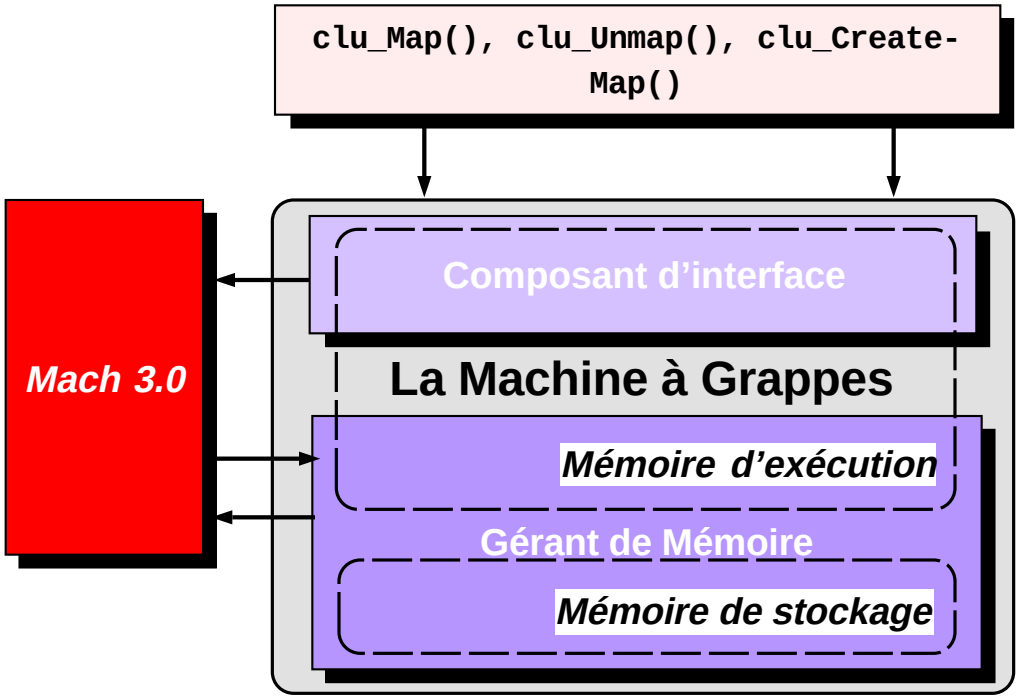

Fig. 4.2 : Architecture logicielle de la machine à grappes

> On peut décrire la machine à grappes selon deux points de vue : celui de la gestion de la mémoire ou celui de la structure logicielle.
>
> Au niveau de la gestion de la mémoire, on retrouve les deux niveaux de mémoire : mémoire d'exécution et mémoire de stockage. Au niveau logiciel, on a deux composants : un composant d'interface qui gère la mémoire d'exécution pour le compte des contextes et une partie serveur, le gérant de mémoire, qui gère à la fois la mémoire d'exécution pour ses relations avec le micro−noyau et la mémoire de stockage pour ce qui touche à la persistance des grappes.

La figure Fig. 4.3 présente l'architecture de la machine à grappes en terme d'entités d'exécution (processus Unix), ainsi que les interactions entre les différentes entités que sont les contextes utilisateurs, les gérants de mémoire et le micro−noyau Mach.

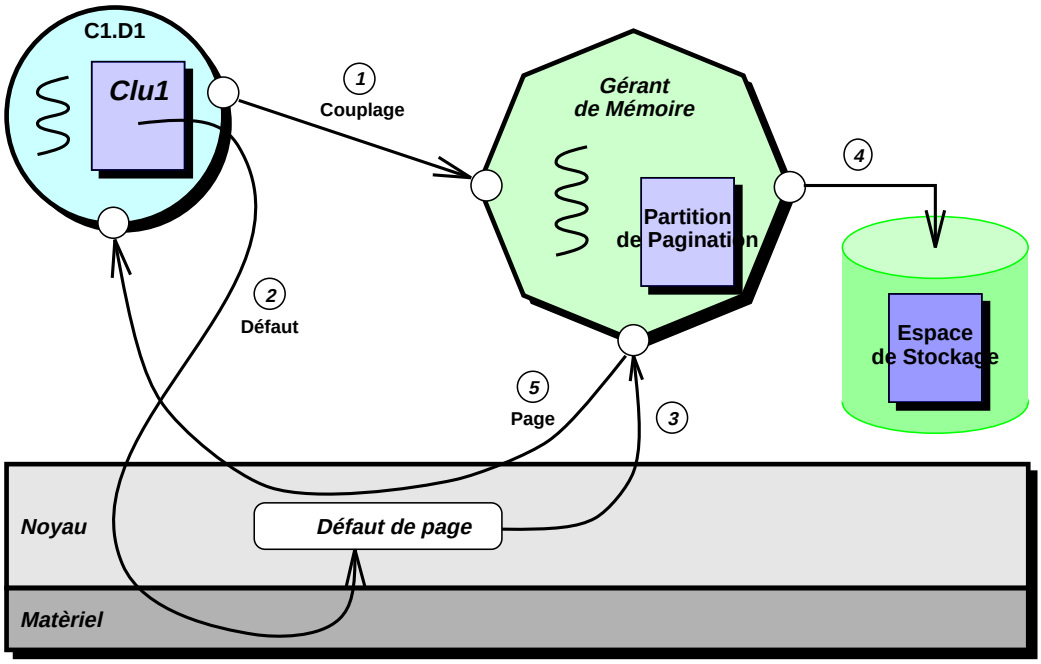

Fig. 4.3 : Mise en œuvre de la machine à grappe

> Cette figure montre l'utilisation d'une grappe Clu1 par un contexte C1 du domaine D1. Le contexte a tout d'abord demandé au gérant de mémoire le droit de coupler la grappe *{1}* (primitive `clu_Map`). Puis il déclenche des défauts de pages lorsqu'il essaye d'accéder à la grappe *{2}*. Le noyau demande au gérant de lui renvoyer la page réclamée par C1 *{3}* (requête `memory_object_data_request`). Le gérant va éventuellement lire la page en espace de stockage *{4}* (si il n'en possède pas une image dans la partition de pagination). Il retourne la page au noyau qui du même coup réveille C1 *{5}* (requête `memory_object_data_supply`).

IV.1.1.2 Désignation et localisation des grappes en mémoire d'exécution

Notre objectif principal quant à la désignation des grappes était de définir un format d'identificateur permettant une localisation rapide et uniforme des grappes tant en mémoire d'exécution qu'en mémoire de stockage.

Comme nous l'avons vu en sous−section II.3.1, il existe deux grandes familles d'identificateurs : les identificateurs logiques et les identificateurs physiques. Dans la mesure où nous voulions offrir des concepts de bas niveau, nous avons préféré utiliser des identificateurs permettant de localiser physiquement la grappe. Nous pensons en effet qu'il est préférable de laisser les couches hautes du système, machine à objets ou service de désignation, offrir une fonction de désignation logique des objets et qu'elle n'est pas nécessaire au niveau des grappes

Une identification de grappe est donc constituée des deux parties : un identificateur de volume et un identificateur local à ce volume (Fig. 4.4).

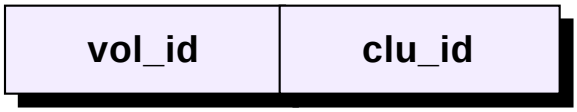

Fig. 4.4 : Format d'un identificateur de grappe

Les deux champs sont codés sur 16 bits, ce qui permet d'adresser 64K grappes par volume.

Ce format d'identificateur est de plus cohérent avec la mise en œuvre des grappes en mémoire d'exécution. En effet, pour coupler une grappe, la machine à objets doit envoyer une requête au gérant de mémoire responsable de la dite grappe. Or la seule information que possède la machine à objets est une information d'administration qui lui indique quel est le gérant de mémoire du volume. En consultant, le champ `vol_id` de l'identificateur de la grappe, la machine à objets est capable de localiser très simplement le gérant de mémoire à contacter.

La contre−partie est qu'un tel identificateur rend pratiquement impossible (sauf à sacrifier les performances) le changement de volume. Une grappe est créée dans un volume et ne pourra pas être déplacée dans un autre volume.

Une nouvelle fois nous pensons que la migration doit être traitée au niveau des objets qui sont les entités réellement manipulées par les utilisateurs. Il s'agit là d'un point essentiel dans la structure du système et seul l'expérience pourra nous dire si cette solution est viable ou non.

IV.1.1.3 Protection

Nous avons vu en section II.3 que la mise en œuvre d'un modèle de mémoire persistante demandait des mécanismes de protection. La protection englobe deux fonctions :

- Garantir l'isolation des utilisateurs entre eux.
- Garantir l'isolation des applications entre elles.

Isolation des utilisateurs

La protection des grappes repose sur la notion de propriétaire. Une grappe appartient à l'utilisateur qui l'a créée, cet utilisateur est identifié comme étant le ***propriétaire*** de la grappe.

L'isolation des utilisateurs est garantie par le fait qu'un contexte ne peut coupler que des grappes appartenant au propriétaire associé à ce contexte.

> Le gérant de mémoire vérifie à la réception d'une demande de couplage si le contexte client a le droit de coupler la grappe. Si le couplage est interdit, la machine à grappes retourne l'identité du propriétaire de la grappe. La machine à objets re−dirige la demande vers un contexte (du même domaine) associé au propriétaire et dans lequel le couplage[(3)] est permis.

Pour que cette protection soit sûre, il faut être capable d'authentifier le contexte demandant le couplage. Il faut de plus s'assurer que les contextes ne sont pas capables de falsifier ou de fabriquer cette information. Dans Guide, l'authentification des contextes est basée sur l'utilisation des ports Mach qui sont des entités protégées du système (cf sous−section III.3.2). A chaque contexte, la machine d'exécution associe un port d'identification ainsi qu'un identificateur d'utilisateur.

La demande de couplage d'une grappe consiste en une requête émise par la partie cliente de la machine à grappes vers le gérant de mémoire. Cette requête est réalisée par un envoi de message Mach. On associe à la requête le port du contexte client. Le gérant de mémoire, qui est une tâche privilégiée du système, peut alors identifier de manière sûre le contexte client. Le gérant de mémoire consulte alors la machine d'exécution pour connaître l'utilisateur pour le compte duquel s'exécute le contexte.

L'identification est sûre car les ports sont protégés, l'association utilisateur−contexte est maintenue par une tâche système (la machine d'exécution) et seule une tâche protégée (le gérant de mémoire) a accès à cette information.

Isolation des applications

> Les applications sont isolées par le fait même du couplage des grappes. En effet, comme il est définit en II.3.4, isoler des applications signifie qu'une erreur dans une application ne doit pas affecter les applications ne partageant pas de grappes avec elle. En ne couplant que les seules grappes qu'elle utilise, une application est, de fait, isolée des autres applications.
>
> Le couplage de grappe est donc particulièrement intéressant dans la mesure où il rend les applications moins vulnérables aux événements extérieurs. Ce qui est un progrès par rapport à la première version du système Guide, où toutes les taches d'un même site partageaient les portions de la mémoire d'exécution localisée sur ce site [Freyssinet 91].

---

(3) Rappel : un domaine est constitué d'un ensemble de contextes associés à différents propriétaires.

Isolation du système

Un facteur de protection supplémentaire est offert par l'isolation des fonctions sensibles du système et notamment les fonctions de gestion de la persistance (accès et gestion des disques) dans un serveur protégé du système.

Le gérant de mémoire est la seule entité du système qui connaisse la structure de l'espace de conservation. Cette complète isolation entre l'espace de conservation et les applications augmente grandement la sûreté du système.

IV.1.1.4 Partage

La mise en œuvre du partage, dans un système distribué tel que le notre, touche principalement à deux aspects :

- La définition d'un modèle de partage. C'est à dire définir si les grappes peuvent être partagées entre plusieurs sites et selon quel modèle de cohérence mémoire (stricte ou lâche).
- La prise en compte de la protection et de son influence sur le partage.

Au niveau du modèle, nous allons tout d'abord considérer qu'il n'existe qu'une seule image de la grappe en mémoire d'exécution. Cela signifie, par exemple, que le partage d'une grappe est réalisé par couplage, par les applications, de la grappe sur un même site. Ce qui entraîne le partage des mêmes pages physiques et donc du même exemplaire de la grappe en mémoire centrale(4).

Par la suite, nous étendons ce modèle au partage réparti de grappes. Cela signifie, alors, que le système gère plusieures images de la même grappe en mémoire d'exécution. Dans la version actuelle, le modèle de cohérence mémoire stricte, nous est imposé par l'utilisation des mécanismes sous−jacents du micro−noyau.

Dans le cas où il n'existe qu'une seule image de la grappe en mémoire d'exécution, la mise en œuvre du partage est similaire au traitement de la protection. Lorsque la machine à objets fait une demande de couplage, le gérant de mémoire s'assure que la grappe n'est pas déjà utilisée sur un autre site. Si c'est le cas, il re−dirige la demande vers le site utilisant déjà la grappe. Cette opération est appelée ***extension*** dans le modèle d'exécution de Guide. On bénéficie ainsi de la gestion centralisée d'une grappe par son gérant de mémoire. La figure Fig. 4.5 illustre ce mode de fonctionnement.

---

(4) C'est pourquoi, on considère, dans ce cas, qu'il n'existe qu'une seule image de la grappe en mémoire d'exécution.

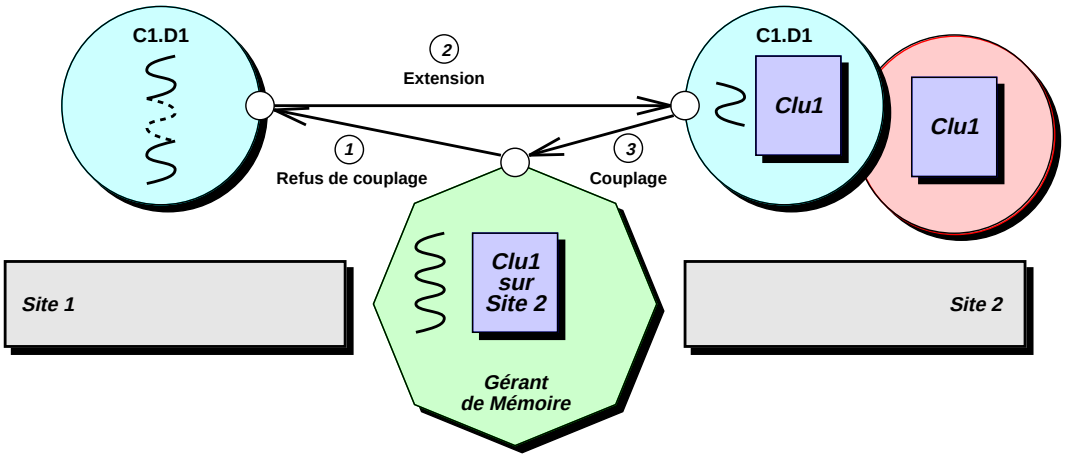

Fig. 4.5 : Couplage d'une grappe et extension de contextes

> Le contexte C1 du domaine D1 a envoyé une demande de couplage de la grappe Clu1 au gérant de mémoire. Le gérant constate que la grappe est déjà couplée sur le site 2, il refuse donc à C1.D1 le couplage sur le site 1 et il le redirige sur le site 2 (*étape 1*). Le contexte C1.D1 demande à la machine d'exécution de lui créer une extension sur le site 2 *(étape 2).* Une fois lancée, l'extension de C1.D1 sur le site 2 peut renvoyer la demande de couplage de Clu1 au gérant de mémoire *(étape 3).*

Parmi les points positifs du mécanisme d'extension pour la mise en œuvre du partage de grappes citons :

- La simplicité de la gestion de la mémoire. Il n'est pas nécessaire, en effet, de mettre en œuvre un protocole de cohérence de mémoire répartie puisque on accède aux grappes en mémoire d'exécution que sur un seul et même site.
- La mise en œuvre efficace de la synchronisation. Les grappes contiennent des objets servant à la synchronisation des applications, les outils de synchronisation classique du type "test&set" peuvent être utilisés directement, car l'opération est locale pour ces objets.
- La configuration des applications. En forçant le couplage d'une grappe sur un site donné, on permet aux applications de contrôler et de configurer leur exécution. Cela peut être utile à des fins de partage de charge voir même de protection (par exemple, une application a l'assurance que tel objet sensible n'est couplé que sur un site sûr).

Les inconvénients sont le pendant des avantages :

- La création d'une extension est coûteuse. Elle peut ainsi s'avérer prohibitive lorsque le taux d'accès à une grappe est faible.

- Le parallélisme réel entre applications est réduit de fait car les applications en partageant des grappes vont partager les ressources des mêmes machines.
- La gestion des extensions est relativement complexe, concernant aussi bien la gestion des structures d'exécution du système (domaines et activités) que son administration (arrêt d'un site et migration de domaine).

  On peut ainsi considérer que l'extension de contextes, par la complexité induite en terme de nombre de structures d'exécution et de la répartition généralisée de celles−ci, fragilise les domaines voire même le système.

Nous verrons par la suite comment une mémoire virtuelle répartie nous permet de remédier aux effets négatifs du mécanisme d'extension.

Lors de la conception de la machine à grappes, nous avons fait l'hypothèse que les applications utiliseraient deux familles de grappes dont les besoins en protection et en partage sont de nature différente :

- Les *grappes qui contiennent des programmes* ne seront pas modifiées au cours de l'exécution des applications. De plus, de la même manière que dans un système centralisé on cherche à partager le code des programmes ou des librairies, ces grappes doivent être partagées le plus possible (si possible sur plusieurs machines).
- Les *grappes qui contiennent des données* peuvent et seront modifiées au cours des exécutions. Par ailleurs, les données d'un utilisateur doivent être protégées.

Cette constatation nous a amené à définir différents types de grappes qui correspondent aux différentes utilisations que nous avons envisagées. Cependant, il nous a semblé intéressant de généraliser ces types d'utilisation et de ne pas se cantonner à définir des grappes code et des grappes données. Les différents types de grappes supportés par notre système sont :

Les grappes partageables protégées

Ce sont des grappes destinées à contenir des données. Elles sont dites protégées car elles ne seront couplées que par les applications (du moins les utilisateurs) ayant reçus le droit de le faire de leur propriétaire. De plus, elles ne seront couplées que dans des contextes appartenant au propriétaire (ce qui renforce l'isolation des utilisateurs).

Leur mode de couplage par défaut est le mode lecture−écriture.

Les grappes partageables

Ce sont des grappes destinées à contenir du code (celui des méthodes) et/ou des classes. Ces grappes ne sont pas censées être modifiées durant

l'exécution des applications. Elles sont donc couplées par défaut en mode lecture−seule.

Ces grappes peuvent alors être couplées dans tous les contextes qui le demandent et ce quel que soit le propriétaire associé au contexte, afin de permettre l'exécution du code des méthodes sur les objets de ce propriétaire. Le propriétaire d'une telle grappe peut toutefois en demander le couplage en écriture pour recompiler une classe (on se ramène alors à l'utilisation d'une grappe protégée).

Dans la mesure où de telles grappes ne sont jamais modifiées et qu'il n'y a donc pas lieu de mettre en œuvre un protocole de cohérence de mémoire répartie, le gérant de mémoire autorise leur couplage sur plusieurs sites. Ce qui revient à donner des copies en lecture des pages de la grappe sur tous les sites où celle−ci est couplée.

Les grappes Unix

Ce sont des grappes qui doivent permettre la gestion d'une partie d'un contexte comme un processus Unix. Elles comprennent deux parties destinées à contenir l'équivalent des sections "text" et "data" d'un processus Unix, ainsi que l'information permettant d'allouer la section "bss" lors du couplage. La partie "text" de la grappe pourra être partagée par différents contextes alors que la partie "data" sera locale à chaque contexte.

Ce type de grappe ne peut pas grandir. Un domaine ne peut coupler qu'une seule grappe Unix. Cette caractéristique est due au mode de connexion entre le système Guide et Unix. Sa description dépasse le cadre de cette étude, on se référera à [Rousset 93] pour plus de détails.

Il peut sembler paradoxal de présenter la grappe comme une entité de bas niveau sémantique (un ensemble de pages non−typées) et de lui associer un type et un mode de couplage. En fait, le type et le mode de couplage définissent une notion orthogonale à celle du contenu de la grappe qui est celle de son utilisation. C'est pourquoi nous avons appelé les grappes, grappes protégées ou grappes partageables et non pas grappes objets ou grappes classes.

Il existe une variante dans l'utilisation des grappes protégées. Le micro−noyau Mach 3.0 dans sa version NORMA offre une couche de mémoire virtuelle distribuée appelée XMM (eXtended Memory Management) [Bryant 93]. La XMM peut gérer la cohérence des grappes entre plusieurs machines de manière complètement transparente pour le gérant de mémoire. Dans une telle configuration, une grappe protégée peut donc être partagée par plusieurs sites sans induire une complexité supplémentaire au niveau du gérant de mémoire. La XMM met en œuvre un

protocole de gestion de la mémoire en cohérence stricte. La figure Fig. 4.6 nous montre le fonctionnement du système selon la vision d'un gérant de mémoire avec ou sans la XMM.

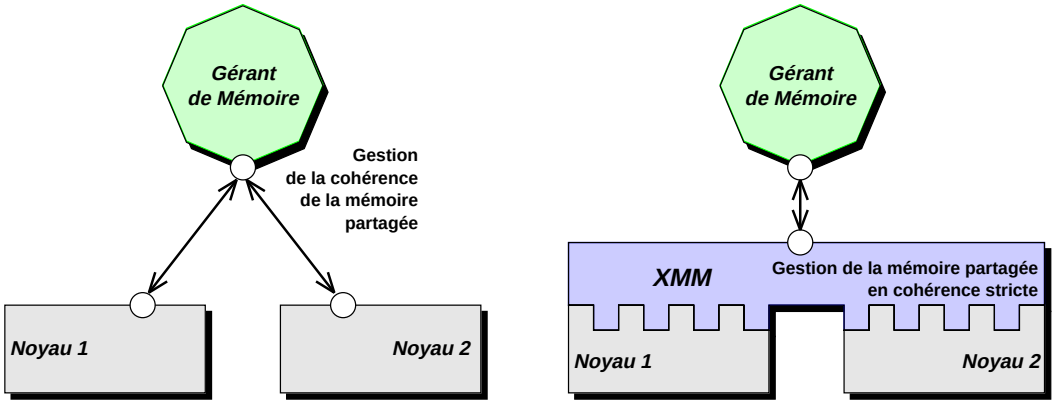

Fig. 4.6 : La Mémoire Virtuelle Distribuée de Mach/NORMA

> Le dessin de gauche nous montre une configuration sans la XMM. Le gérant de mémoire est alors chargé de mettre en œuvre un protocole de cohérence répartie de la mémoire partagée par les deux sites.
>
> Sur le dessin de droite, la XMM s'interpose entre les noyaux et le gérant de mémoire. La XMM présente au gérant de mémoire l'ensemble des sites comme étant un seul et même site.

Cette approche peut sembler paradoxale : les paginateurs externes sont censés permettre la définition d'un modèle de mémoire au niveau applicatif et rendre ainsi plus souple le système, or la XMM court−circuite complètement les paginateurs en plaquant son propre modèle de mémoire répartie qu'elle gère en cohérence stricte. Cette apparente contradiction trouve sa justification dans la constatation suivante : les paginateurs ont pour rôle d'associer un contenu logique à une mémoire, la XMM se contente de distribuer cette mémoire. Il n'est donc pas anormal que les paginateurs soient déchargés de la fastidieuse tâche qu'est la gestion de la cohérence de la mémoire répartie.

De manière plus générale, il semble que dans la pratique la recherche d'une efficacité maximale pousse à la mise en œuvre au niveau bas (dans le noyau) d'un certain nombre de fonctions de base telles que les communications ou la gestion de la mémoire virtuelle.

Lorsque l'on utilise la XMM, l'extension de contexte n'est plus nécessaire. En effet, le partage de grappes s'effectue au travers de la XMM et il n'est plus utile de regrouper les contextes sur le même site.

Cette constatation appelle deux remarques :

- le mécanisme d'extension doit toujours être utilisé à des fins de protection car une grappe est toujours couplée dans un contexte associé à son propriétaire et lorsqu'une application utilise des grappes appartenant à différents propriétaires, elle doit être capable de s'exécuter dans les différents contextes.
- les deux mécanismes sont en fait complémentaires. La XMM n'est par exemple pas adaptée au partage d'objets synchronisés, car cela génère trop d'invalidation de pages sur le réseau. Par contre, elle est beaucoup plus efficace en ce qui concerne les grappes à fort taux d'accès en lecture.

La conjonction de ces deux mécanismes est particulièrement intéressante tant pour les performances obtenues que par la qualité de service qu'ils offrent.

Leur utilisation ne pourra cependant pas se faire de manière efficace sans l'utilisation d'un configurateur d'application qui permette au programmeur de spécifier le mécanisme de partage associé aux différentes grappes. Un mini configurateur est disponible dans la version actuelle du système. Il permet de fixer le site de couplage associé à chaque grappe et donc de forcer l'extension sur ce site. Par défaut, le système applique la politique de chargement sur le premier site demandeur. Une évolution de ce configurateur serait d'ajouter une option autorisant le couplage multi−sites et donc l'utilisation de la XMM. Le programmeur aurait alors trois choix : soit le système applique la politique par défaut avec extension, soit le site de couplage est fixé, soit on utilise la mémoire distribuée de la XMM.

Une évaluation plus complète prenant en compte l'influence de chaque mécanisme sur la structure des applications, sur le partage de charge au niveau du système et sur la vision offerte au programmeur doit être menée dans un futur proche.

IV.1.1.5 Gestion dynamique de la taille des grappes

Le choix de regrouper les objets dans des grappes vient de contraintes de réalisation. Il n'est pas admissible que ce choix pénalise le programmeur en l'empêchant de créer de nouveaux objets lorsqu'une grappe est pleine. Il est donc impératif qu'une grappe puisse grandir dynamiquement de façon transparente pour le programmeur.

La machine à grappe offre des fonctions permettant d'accroître la taille des grappes en mémoire d'exécution. Ces fonctions sont basées sur l'utilisation des régions de mémoire dont le support est fourni par le micro−noyau.

Une tâche Mach a la possibilité de coupler plusieurs régions d'un même objet mémoire dans son espace d'adressage. Les contextes pourront donc coupler des portions successives de grappes au fur et à mesure de leur création.

Les figures suivantes montrent les différentes étapes de la création d'une grappe et de son agrandissement ainsi que de l'influence de cet agrandissement sur l'état de la mémoire virtuelle des contextes(5).

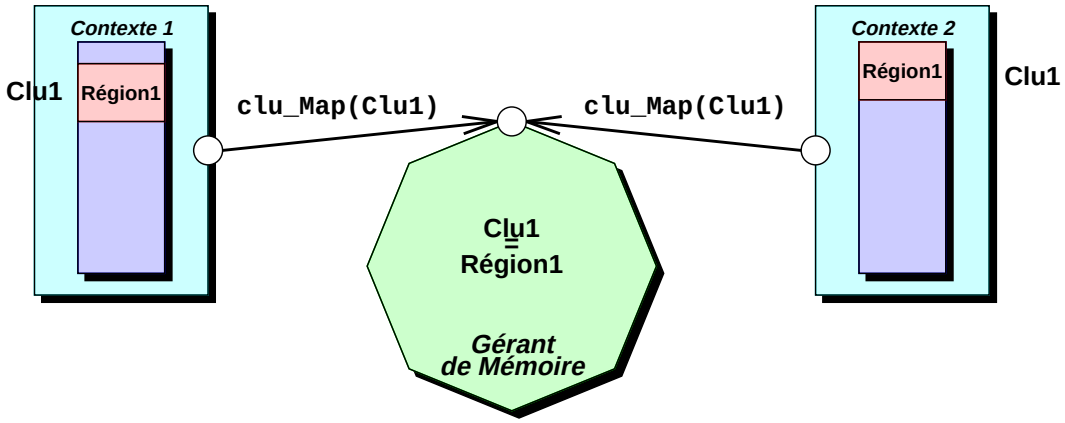

Fig. 4.7 : Étape 1, couplage de la grappe Clu1

> Sur cette figure, on aperçoit la grappe `Clu1` dont la région initiale `Région1` est couplée par deux contextes `Contexte1` et `Contexte2` (`clu_map(Clu1)`).

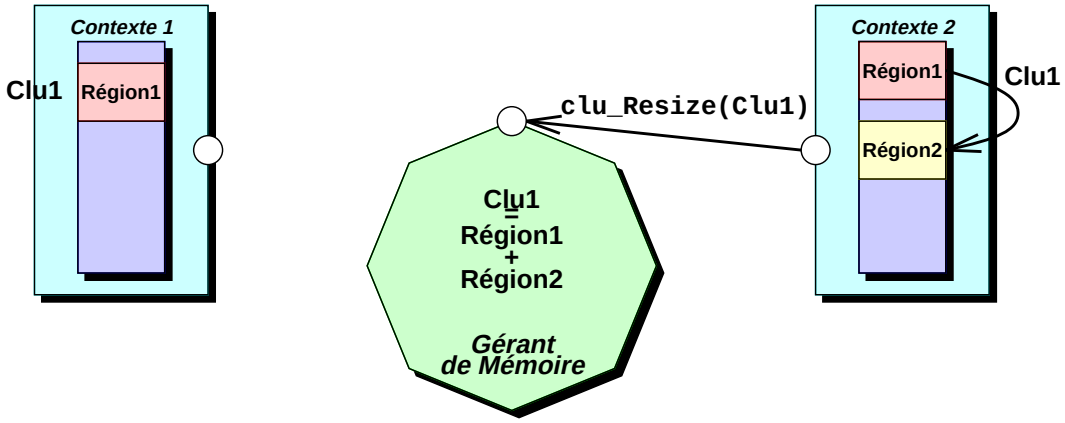

Fig. 4.8 : Étape 2, agrandissement de Clu1 par Contexte2

> La grappe est agrandie par `Contexte2` (`clu_Resize(Clu1)`). Elle est maintenant constituée de deux régions `Région1` et `Région2`. `Contexte2` a donc couplé la nouvelle région dans son espace d'adressage. Le couplage des deux régions ne s'est pas fait de manière contiguë car une autre grappe a été couplée entre−temps par `Contexte2`. Le composant d'interface de la machine à grappe maintient donc à jour le lien entre les différentes régions qui forme la grappe. On constate que pour sa part le gérant de mémoire se contente de mémoriser la taille totale de la grappe.

---

(5) Remarque : le scénario choisi est purement arbitraire.

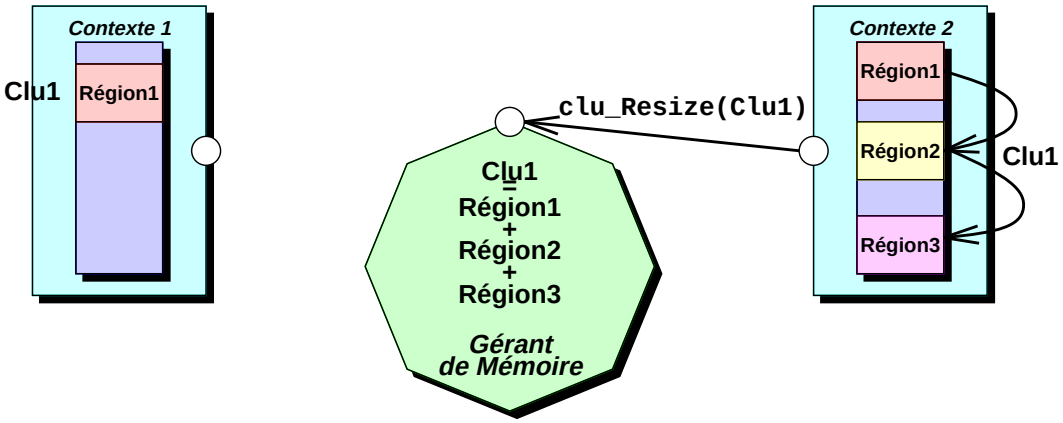

Fig. 4.9 : Étape 3, agrandissement de Clu1 par Contexte2

La grappe est de nouveau agrandie par `Contexte2` (`clu_Resize(Clu1)`). Elle est maintenant constituée des trois régions `Région1`, `Région2` et `Région3`. On notera que la liste des régions maintenue par le composant d'interface inclut maintenant la nouvelle région.

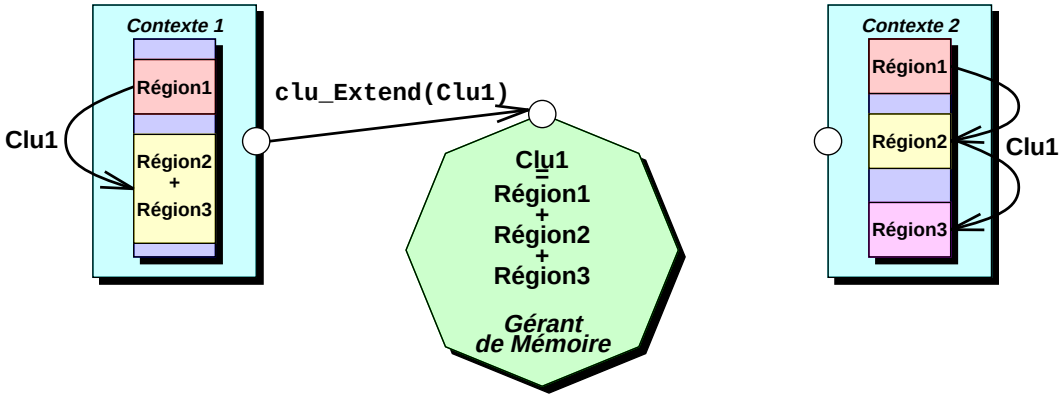

Fig. 4.10 : Étape 4, couplage par Contexte1 du "reste" de la grappe

Enfin pour terminer, `Contexte1` demande le couplage des régions nouvellement créées (`clu_Extend(Clu1)`). La seule information à sa connaissance est qu'il existe une suite à la grappe (cette information est stockée dans la grappe elle−même par le composant d'interface), mais il n'en connaît pas la taille. Cette demande se traduit par un couplage contigu des deux régions `Région2` et `Région3`.

Il faut noter que le fait que les contextes couplent une grappe en une ou plusieurs régions est complètement invisible au gérant de mémoire. Le gérant ne se préoccupe que de la gestion des fautes de pages survenant sur la grappe. Il doit donc uniquement s'assurer que ces fautes sont valides, c'est à dire que les couplages effectués par les contextes concernent bien des régions valides de la grappe.

Dans la pratique la gestion des listes de régions est remontée par le composant d'interface au niveau de la machine à objets, qui a besoin de prendre en compte la non−contiguïté des régions pour fonctionner correctement. On peut d'ailleurs faire remarquer que si cette gestion de l'accroissement de la taille des grappes est satisfaisante au niveau du gérant de mémoire puisqu'il ne se préoccupe que de la gestion des fautes de pages survenant sur une grappe de taille variable, elle induit une complexité accrue au niveau du composant d'interface et des couches supérieures. Cette solution est cependant plus élégante et efficace qu'une solution qui forcerait le découplage général de la grappe et son recouplage afin d'assurer la continuité de la grappe en espace virtuel.

IV.1.1.6 Interface

L'interface de la machine à grappes en mémoire d'exécution est composée des fonctions suivantes :

- **`clu_CreateMap`**

  Permet la création d'une grappe. La grappe est alors couplée dans la tâche courante et appartient au propriétaire associé à cette tâche.

- **`clu_Map`**

  Permet le couplage d'une grappe. La grappe est couplée (si le couplage est autorisé) à une adresse choisie par Mach dans l'espace virtuel de la tâche courante.

- **`clu_Unmap`**

  Permet d'annuler le couplage d'une grappe dans la tâche courante. Cette fonction est appelée dans chaque tâche à la fin de l'exécution du Domaine pour le compte duquel la tâche s'exécute.

- **`clu_Resize`**

  Permet d'ajouter une nouvelle région à la grappe et de coupler cette région dans la tâche courante.

- **`clu_Extend`**

  Permet de coupler dans la tâche courante une région de grappe nouvellement créée par une autre tâche.

- **`clu_Commit`** et **`clu_Abort`**

  Permet de forcer la re−copie d'une grappe en mémoire de stockage ou d'invalider les modifications effectuées en mémoire d'exécution (on se reporter à la sous−section IV.1.2.4).

IV.1.2 Gestion de grappes en mémoire de stockage

La mémoire de stockage (mémoire et interface d'accès) est réalisée par le ***service de stockage Guide***. Le rôle du service de stockage est d'offrir à la machine à grappes une abstraction uniforme des différents supports de stockage disponibles. Cette abstraction est le volume, un volume représente l'entité de stockage dans laquelle les grappes sont placées. Le volume fixe les caractéristiques des services associées à la grappe : fiabilité, performance ou mode d'accès.

Notre objectif majeur lors de la définition de l'interface du service de stockage fût de fournir une interface simple sur la base de volumes, visant en particulier à la réutilisation de serveurs de fichiers externes. Nous allons donc en premier lieu parler de la structuration de cette couche, avant d'en détailler l'interface.

IV.1.2.1 Organisation générale

Le service de stockage est rendu par des entités logiques, appelées ***serveurs de stockage***, pouvant être mises en œuvre par plusieurs processus s'exécutant sur des machines différentes et ne supportant pas nécessairement le système Guide. Le service de stockage fiable pourra par exemple être rendu par plusieurs serveurs Goofy.

Cette volonté de réutilisation de serveurs de stockages dits externes s'appuie sur la notion de volume. En effet, le volume offre à la machine à grappes une uniformisation des abstractions physiques manipulées par les différents serveurs. Le rôle du service de stockage est donc d'unifier les différentes interfaces offertes par des serveurs externes que ce soient des serveurs de fichiers du type Unix (UFS, AFS) ou même des serveurs de persistance pour bases de données comme la couche WiSS du système O_2 [Deux 91].

Dans un premier temps, la mémoire de stockage est construite sur un nombre restreint de services de stockage dont l'accès est rendu homogène par le service de stockage. Il existe deux services de stockage offrant des qualités de service distinctes :

- Le service de stockage rapide (SSR). Ce service est mis en œuvre sur chacune des stations de travail avec disque du système Guide (un serveur SSR par station de travail avec disque). Il offre des volumes de disponibilité réduite mais avec un temps d'accès local court.
- Le service de stockage fiable (SSF ou Goofy), ce service doit fournir un niveau de disponibilité accrue en cas de panne d'un support de stockage. Il utilise les idées développées dans [Chevalier 92b] et [Chevalier 93b]. Sa description fait l'objet du chapitre VI.

Ces différents services sont chapeautés par le service de stockage, qui présente l'interface décrite plus loin et qui redirige les requêtes vers le service de stockage concerné.

Analysons maintenant comment est stucturé le service de stockage. Le service de stockage doit principalement gérer la localisation des volumes dans les serveurs. En effet, les volumes appartenant à différents serveurs, à la réception d'une requête, le service de stockage doit identifier le serveur qui gère le volume en question. Cela lui permet de déterminer les modalités d'accès à l'entité physique qui représente le volume. Le service de stockage est donc structuré en deux parties :

- Une partie commune à tous les serveurs chargée de la localisation des volumes et de tous les traitements communs aux différents serveurs. Par abus de langage et dans la suite de ce chapitre, cette partie sera confondue avec le service de stockage.
- Une partie spécifique à chaque serveur, dite gérant de stockage, qui lie la notion de volume aux entités physiques manipulées par ce serveur.

La figure Fig. 4.11 illustre l'architecture d'un service de stockage s'appuyant sur le service de stockage rapide ainsi que sur le serveur externe de stockage fiable Goofy.

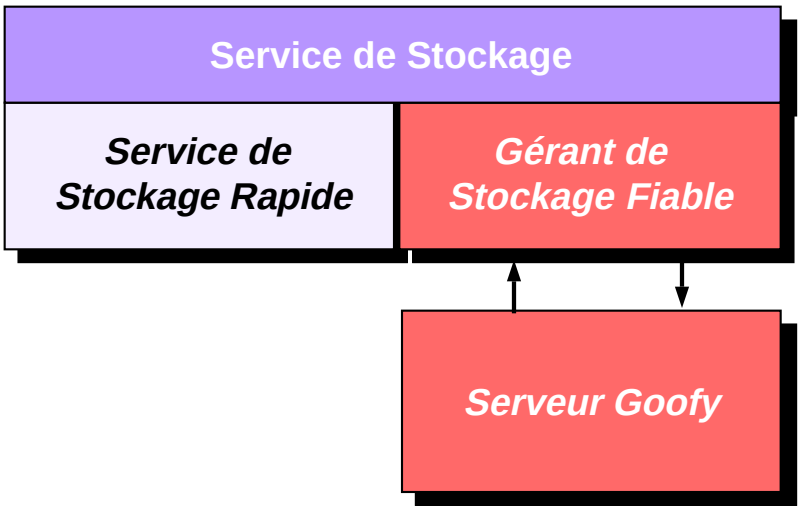

Fig. 4.11 : Architecture logicielle du service de stockage

Cette figure appelle deux remarques :

- Le service de stockage rapide n'a pas de gérant de stockage associé dans la mesure où pour des raisons d'efficacité il fait partie intégrante du service de stockage, c'est à dire qu'il est réalisé par une bibliothèque de primitives qui est liée avec le binaire de chaque gérant de mémoire.

- Le service de stockage fiable est quant à lui réalisé par deux composants : un gérant de stockage lié avec le binaire du gérant de mémoire et un processus externe, le serveur de stockage fiable Goofy.

L'interface de chacun des services de stockage utilisé (Goofy et SSR) se présente sous forme d'une bibliothèque de primitives dont la gestion est uniformisée par le service de stockage. Dans le cas de l'intégration d'un nouveau serveur de stockage externe, afin d'éviter une trop grande complexité au niveau du service de stockage, on préférera uniformiser l'accès à ce service par réécriture de la partie client qu'il fournit ou par écriture d'un gérant de stockage spécifique.

L'adresse de tous les points d'entrée fournis par chaque interface est regroupée dans une table. Le service de stockage appelle les différentes fonctions par indirection à travers cette table. Cette interface (de type pilote Unix) permet d'intégrer un nouveau gérant de stockage en modifiant de manière minime le code du service de stockage.

L'inconvénient principal de cette solution est que l'insertion d'un nouveau service de stockage dans le système est rendue plus complexe par le fait même que la table d'indirection est statique. Sa modification demande une régénération du gérant de mémoire. Cette intégration du service de stockage dans le binaire de gérant de mémoire réduit la souplesse de notre architecture. Nous avons réfléchi à certains aménagements :

- Restreindre le nombre des clients de service de stockage liés dans le binaire d'un gérant de mémoire particulier. On peut ainsi spécialiser un gérant de mémoire qui ne pourra monter que des volumes gérés par un SSR. Cela permet d'optimiser la couche de localisation des volumes, mais en interdisant le montage des volumes gérés, par exemple, par Goofy.
- fournir un client générique qui offre en dessous une interface message. Un tel client accepterait dynamiquement de nouveaux services de stockage. Il suffirait d'établir une liaison (numéro de port du serveur).

Cependant dans la réalisation actuelle, l'efficacité et la simplicité offerte par notre solution nous ont semblé largement suffisantes. L'intégration du service de stockage fiable Goofy (cf section VII.3) a par ailleurs été largement facilitée par l'utilisation de la table d'indirection.

IV.1.2.2 Concepts

L'espace de stockage est structuré autour de services de stockage, qui manipulent des volumes.

volume

Un volume est une unité logique d'administration et de conservation des grappes. Un volume est identifié de manière unique par un identificateur unique de volume (ou *vol_id*) alloué par le service d'administration.

Notons qu'il n'est pas fait de supposition quant à la représentation physique du volume, celle−ci est laissée à la charge de chaque service de stockage. Un volume pourra par exemple être réalisé sous forme d'un répertoire Unix dans lequel seront enregistrés des fichiers réalisant les grappes. Il pourra aussi être réalisé sous forme d'une partition du disque dans laquelle les grappes sont représentées par des listes chaînées de blocs.

Le volume est l'unité de gestion de l'espace de stockage. Il est affecté de manière indivisible à un service de stockage. Notons, qu'un service de stockage peut avoir la charge d'un nombre important de volumes.

Les volumes sont créés dynamiquement par l'administrateur du système. Leur destruction est explicite.

grappe

Au niveau du service de stockage, une grappe est vue comme une unité d'adressage logique constituée d'un ensemble d'informations contiguës. La contiguïté des informations de la grappe apparaît dans l'adressage, mais n'influe pas sur le mode physique de conservation, par exemple, sous forme d'arbre de blocs.

service de stockage

Un service de stockage est une entité répartie. Il peut exister plusieurs services de stockage de nature différente dans une même cellule Guide. Chaque service ne connaît que les volumes dont il a la charge. Il n'a pas une connaissance globale de l'ensemble des volumes de la cellule.

L'interface d'un service de stockage, décrite en IV.1.2.5, est suffisamment simple pour permettre la récupération (avec éventuellement réécriture) de serveurs existants, en particulier de serveurs de fichiers du type AFS ou Pegasus [Leslie 93].

IV.1.2.3 Désignation et localisation

Un volume est désigné, ainsi que nous l'avons noté plus haut, par un identificateur unique de volume ou `vol_id` codé sur 16 bits. Le bit de plus fort poids d'un `vol_id` est utilisé pour indiquer si le volume est local ou non, à la cellule

courante[6]. Les quinze bits de poids faible constituant l'identification unique du volume dans sa cellule. Ainsi, d'une part à un instant donné dans une cellule on peut adresser 32 K volumes appartenant à des cellules externes, et d'autre part une cellule peut gérer au plus 32 K volumes.

L'association entre un volume et le serveur de stockage qui le gère est conservée dans une table (ou catalogue) accessible par tous les gérants de mémoire. Cette table est mise à jour lors de chaque opération de création de volume. Elle permet de répondre aux opérations de montage (resp. démontage) d'un volume. La gestion de cette table et de sa mise à jour est du ressort de la couche d'administration du système.

IV.1.2.4 Conservation et intégrité

Dans la sous−section II.3.3, nous avons examiné la conservation des données sous l'angle de ses relations avec les problèmes de protection et de contrôle d'intégrité.

La présente sous−section a pour but de décrire la mise en œuvre des mécanismes de conservation en mémoire de stockage. Elle servira aussi de support à la description d'une solution au contrôle de l'intégrité, basée sur une technique de journalisation des modifications. Pour illustrer notre propos, nous allons examiner en détail la réalisation du service de stockage rapide.

Le service de stockage rapide met en œuvre les fonctions de bas niveau de conservation des grappes en mémoire de stockage. Il est mis en œuvre sur chacune des stations de travail avec disque supportant le système Guide. Il offre des volumes de disponibilité réduite (celle du disque ou de la station) mais le temps d'accès est court par rapport aux services de stockage plus évolués (dans la mesure où pour y accéder le système n'a pas à utiliser le réseau).

Le service de stockage rapide peut être réalisé de deux manières correspondant chacune à un choix de réalisation physique des volumes et des grappes :

1. La première solution vise la réutilisation maximale du système de gestion de fichiers Unix (UFS). Les grappes sont alors réalisées par des fichiers et les volumes par des répertoires contenant les fichiers−grappes.
2. La deuxième solution vise à une efficacité maximale en court−circuitant le système de fichiers Unix. Les volumes sont réalisés par des partitions des disques gérées en mode "raw". Les grappes sont réalisées par des listes de blocs dans la partition−volume. Sur Mach, l'efficacité maximale pourrait être atteinte en communiquant directement avec le pilote du disque grâce au mécanisme de couplage de pilote fourni par le noyau.

---

(6) La réalisation des mécanismes de désignation inter−cellules n'est pas mise en œuvre dans la version du système Guide actuellement disponible.

La première solution a l'avantage d'être simple et portable (elle n'est pas liée au micro−noyau Mach). De plus, elle autorise la réutilisation de tous les outils d'administration fournis par Unix.

La deuxième solution doit procurer un gain d'efficacité, mais elle impose la mise en œuvre d'algorithmes d'allocation/libération des blocs du disque, de recherche de l'implantation d'une grappe dans un volume, de gestion de caches d'accélération, etc ... Par ailleurs, il faut développer des outils d'administration spécifiques.

Nous avons donc privilégié la réalisation des grappes sous forme de fichiers Unix afin de disposer rapidement d'un service de stockage complet.

Nous avons vu en sous−section II.3.3 qu'un des principaux problèmes à résoudre pour supporter un espace persistant est celui du contrôle de l'intégrité des données. Le système doit en effet garantir que l'image des données sur disque est toujours le reflet d'un état cohérent obtenu à l'exécution.

Dans Guide, le problème consiste à garantir la re−copie atomique en mémoire de stockage des modifications effectuées sur une grappe en mémoire d'exécution. La cohérence est donc fournie au niveau de la seule grappe. Nous n'avons pas voulu garantir la re−copie cohérente d'un ensemble de grappes, car nous pensons qu'il est préférable, dans un premier temps, de laisser traiter ce problème par les mécanismes transactionnels qui doivent être mis en œuvre au niveau application.

La re−copie atomique est basée sur une technique de journalisation. Les modifications sont tout d'abord enregistrées dans un journal (représenté par un fichier Unix). Puis une fois cette opération validée, elles sont reportées sur l'image persistante de la grappe. En cas de panne au cours de cette dernière opération, le système sera capable de "rejouer" l'opération de re−copie à partir de l'image contenue dans le journal. Le détail de l'algorithme de journalisation est présenté dans la sous−section IV.2.3.

Nous avons préféré la journalisation, à la technique des pages fantômes, car les grappes étant réalisées par des fichiers, il était beaucoup plus simple de gérer un journal à ce niveau. Une solution aurait été de faire une copie du fichier qui réalise la grappe avant son utilisation, mais elle aurait entraîné de nombreuses et parfois inutiles copies. Par ailleurs, le journal nous offre une fonction de conservation et d'identification des opérations de stockage qui s'avère très intéressante. Elle est d'ailleurs utilisée pour la mise en œuvre du serveur de stockage fiable Goofy.

La question qui reste alors en suspens concernant la re−copie atomique est la suivante : à quel instant doit−elle prendre place ? Ce problème est particulièrement crucial dans la mesure où le système n'est pas à même de définir une "bonne" politique. La définition d'un état cohérent d'une grappe est du seul ressort des

applications, le système n'ayant pas connaissance de la sémantique du contenu des grappes. Le partage de grappes entre contextes augmente encore la difficulté puisqu'il s'agit alors de définir un état cohérent vis à vis de l'ensemble des applications qui partagent une grappe.

Le système n'étant pas capable de définir une "bonne" politique, nous avons décidé d'offrir une primitive, `clu_Commit`, qui permette aux applications de forcer la re−copie atomique d'une grappe. Par exemple, cette primitive peut être utilisée par un éditeur de texte pour fournir une fonction de sauvegarde d'un document, qui se traduira par la re−copie atomique de la grappe contenant le document.

Cependant, nous fournissons une politique par défaut qui consiste à recopier systématiquement les grappes lorsqu'elles ne sont plus utilisées par les applications. C'est le mode de fonctionnement qui simplifie le plus la tâche du programmeur d'application ; celui−ci ne se soucie ni du couplage ni de la re−copie des grappes. La contre−partie est que l'image d'une grappe peut ne jamais être recopiée en mémoire de stockage (du moins tant qu'elle se trouve couplée par une application).

Associée à la primitive de re−copie, il existe une primitive d'invalidation des modifications (`clu_Abort`) ; toutes les pages modifiées présentent en mémoire d'exécution sont détruites.

Le défaut majeur de notre approche est qu'elle induit une complexité supplémentaire au niveau des applications. En effet, remonter au niveau des applications (mais peut−il en être autrement) des primitives de manipulation des grappes, revient à remonter aussi la notion de grappe.

Néanmoins, la grappe a une importance primordiale au niveau des performances (liées en partie au regroupement d'objet), ainsi qu'au niveau de la protection. Il apparaît donc indispensable de permettre aux programmeurs d'applications avancées, de contrôler et de manipuler les grappes de son application (au travers de son langage de programmation ou de configuration). Le traitement de la re−copie atomique n'est qu'un facteur supplémentaire qui milite pour la prise en compte des grappes par les langages.

IV.1.2.5 Interface

L'interface du service de stockage comporte trois classes de primitives :

- Des fonctions d'administration permettant à l'administrateur du système de manipuler les volumes.
- Des fonctions d'utilisation des volumes permettant à la machine à grappes de monter les volumes ou d'accéder aux blocs contenant des grappes.
- Des fonctions de manipulation des grappes qui contiennent les traitements effectifs utilisés pour la gestion physique des grappes dans les volumes.

C’est à leur niveau (du moins celui du gérant de stockage) qu’est fixée la représentation physique des grappes : sous forme de fichier dans un répertoire ou d’un ensemble de blocs chaînés dans une partition du disque.

Primitives d’administration

La primitive **`vol_Create`** s’apparente à l’ajout d’un nouveau disque sur une machine. C’est donc une opération lourde utilisée par le seul administrateur du système.

La primitive **`vol_Resize`** permet d’ajuster la taille d’un volume.

La primitive **`vol_Delete`** permet la destruction d’un volume. Cette destruction ne pourra être entreprise par l’administrateur qu’après s’être assuré qu’il n’existe plus de référence sur des objets contenus dans le volume à détruire. A charge pour lui d’utiliser le mécanisme de ramasse−miettes qui sera fourni par le système.

Primitives d’utilisation

La primitive **`vol_Mount`** est utilisée par le gérant de mémoire pour le rendre accessible aux applications en mémoire d’exécution. Elle s’apparente au montage d’un système de fichiers sur Unix. Le montage d’un volume par son gérant de mémoire le rend accessible. Notons que l’appel à cette primitive est effectué par le gérant de mémoire "sur ordre" de l’administrateur(7).

La primitive **`vol_Unmount`** annule le montage et supprime donc l’accès aux grappes de ce volume par les applications.

Primitives de manipulation des grappes

On retrouve des primitives de création/destruction des grappes : **`vol_CluCreate`**, **`vol_CluGetDesc`**, **`vol_CluDestroy`**, **`vol_CluResize`**.

L’accès aux pages d’une grappe se fait par les primitives **`vol_CluRead`** et **`vol_CluWrite`**.

IV.2 Mise en œuvre

Dans cette section nous allons analyser deux aspects très particuliers de la mise en œuvre : la gestion de la pagination et la journalisation. L’objectif est de donner un aperçu des problèmes rencontrés lors de la réalisation du support de la persistance dans le noyau du système Guide−2 au travers de la description détaillée de deux points techniques.

(7) Les "ordres" de montage sont enregistrés dans une table gérée par le service d’administration qui permet de les répéter (ou rejouer) automatiquement à chaque lancement, des gérants de mémoire.

IV.2.1 Architecture

Dans la section précédente, la machine à grappes a été présentée sous l'angle de ses composants logiciels. L'objectif de la présente section est de donner un aperçu de la machine à grappes en terme des structures d'exécution (processus) qui la réalise.

Notre objectif était de bénéficier de la nature répartie de l'environnement d'exécution et notamment d'essayer de partager la charge associée au support de la persistance entre les différentes machines. Nous avons donc opté pour une architecture multi−serveurs. La machine à grappes qui réalise notre modèle de mémoire persistante est mise en œuvre par un ensemble de gérants de mémoire et de gérants de stockage répartis sur l'ensemble des sites du système.

Chaque gérant de stockage prend en charge une partie des grappes en mémoire de stockage (en gérant un ou plusieurs volumes). Chaque gérant de mémoire prend en charge un sous−ensemble des grappes en mémoire d'exécution (en montant un ou plusieurs volumes). La figure Fig. 4.12 schématise la réalisation de notre modèle de mémoire.

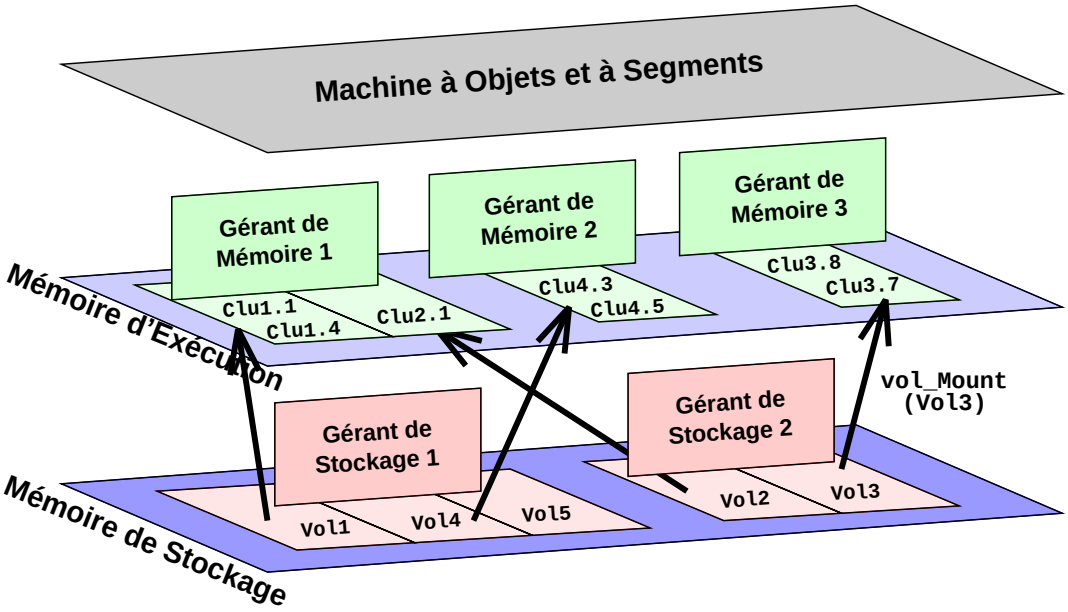

Fig. 4.12 : Mise en œuvre du modèle de mémoire persistante

Cette figure présente les deux niveaux de mémoire ainsi que les structures d'exécution chargées de la gestion de chacun des niveaux.

La mémoire de stockage est gérée par les gérants de stockage. Chaque gérant de stockage s'occupe d'un ensemble de volumes.

La mémoire d'exécution est gérée par les gérants de mémoire. Chaque gérant de mémoire "monte" les volumes dont il a la charge, afin de rendre accessible les grappes qu'ils contiennent aux applications. Ainsi le gérant de mémoire 1 monte les volumes `Vol1` et `Vol2` qui sont respectivement gérés par les gérants de stockage 1 et 2.

On peut remarquer sur cette figure que potentiellement il existe deux niveaux de répartition de la mémoire : un premier niveau représenté par l'ensemble des gérants de mémoire des différents sites et un deuxième niveau géré par les gérants de stockage et réalisé par l'ensemble des volumes de stockage.

La figure Fig. 4.13 présente une configuration de système constituée de trois machines. La vision ainsi décrite est plus fidèle notamment en ce qui concerne les structures d'exécution. Dans la pratique, la plupart des gérants de mémoire contiennent le code du service de stockage afin d'accéder directement aux volumes de leur site.

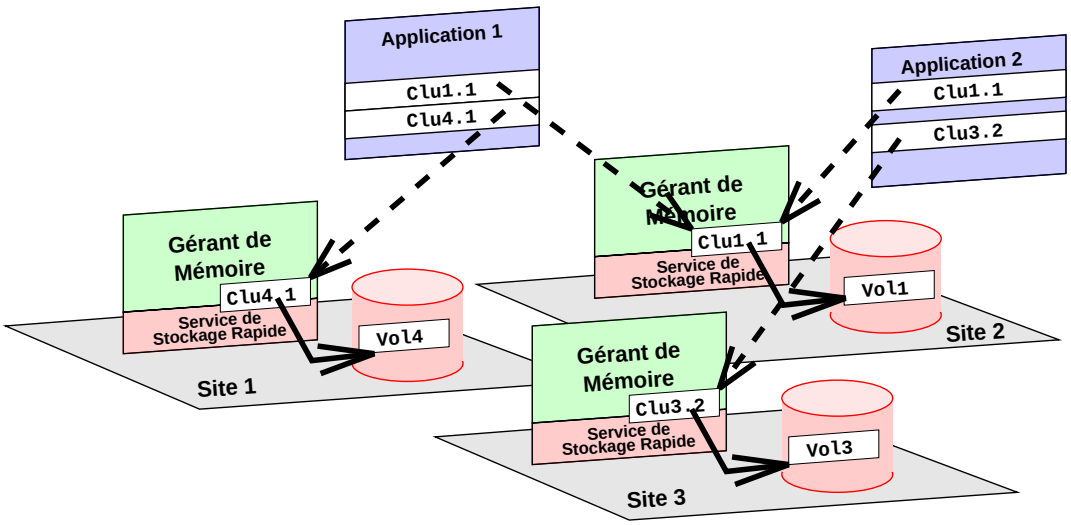

Fig. 4.13 : Une configuration du système Guide

Cette figure représente une configuration constituée de trois machines. Un gérant de mémoire est associé à chaque site. Il est responsable d'un volume qui dépend du service de stockage rapide (d'où l'inclusion du gérant de stockage rapide dans la même entité d'exécution que le gérant de mémoire).

Les applications présentées ont couplé des grappes appartenant aux différents volumes en s'adressant aux gérants de mémoire, répartis sur les trois sites.

IV.2.2 Gestion de la pagination

Un gérant de mémoire a pour rôle de traiter les demandes de pages que lui envoie le noyau du site sur lequel une application a couplé une grappe dont il a la charge. C'est au niveau de la gestion de la pagination que l'on retrouve les traits caractéristiques de notre modèle de mémoire : la persistance et la séparation entre les deux niveaux de mémoire.

Niveaux de mémoires et pagination

C'est l'utilisation d'un espace de pagination différent des disques de stockage qui nous permet de réaliser la séparation entre la mémoire d'exécution et la mémoire de stockage. Lorsque le noyau renvoie une page modifiée au gérant de mémoire celui−ci la conserve dans une partition de pagination (représentée par un fichier Unix) et non pas sur l'image permanente de la grappe.

Persistance et pagination

Nous avons choisi une politique de chargement des pages à la demande. L'opération de couplage de la grappe est une opération "logique" qui consiste uniquement à associer une région de mémoire virtuelle à une grappe. Le transfert des pages de la grappe en mémoire d'exécution (en l'occurence en mémoire centrale) ne sera effectué qu'à la suite des fautes de pages provoquées par les applications.

Cette politique de chargement à la demande influence la gestion de la pagination puisque les pages d'une grappe peuvent être soit en mémoire de stockage, soit en mémoire centrale ou soit en espace de pagination.

La gestion de la pagination est réalisée au travers d'une interface de pagination constituée de requêtes asynchrones émises par le noyau ou par le gérant. Par exemple, si le noyau a besoin d'une page, il émet la requête `memory_object_data_request`. Le gérant de mémoire doit répondre de manière asynchrone par la primitive `memory_object_data_supply`.

La pagination est mise en œuvre dans le gérant de mémoire par un automate basé sur l'état des pages. Un état d'une page définit la localisation physique de la page : en mémoire centrale, en espace de pagination ou sur disque. En fonction de l'état de la page, le gérant réalise le traitement approprié (lecture ou stockage de la page) puis il retourne si besoin est la page ou un acquittement. La figure Fig. 4.14 décrit les différents états d'une page ainsi que les transitions entre ces états.

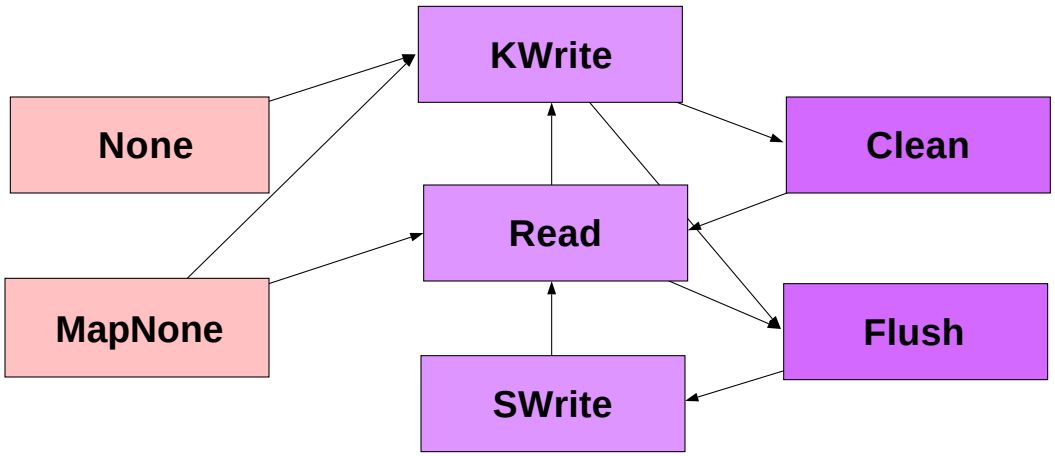

Fig. 4.14 : États et changements d'états d'une page

None

L'état None correspond à une grappe qui vient d'être créée, il n'existe pas de mémoire associée à la page. Lorsque le noyau demande la page, on lui permet d'allouer l'image mémoire initiale. Elle passe donc dans l'état KWrite, le gérant mémoire n'a donc pas une image "à jour" de la page.

MapNone

L'état MapNone correspond à une grappe qui vient d'être couplée. La grappe a une image persistante en mémoire de stockage. Lorsque le noyau demande la page, le gérant de mémoire va la lire en mémoire de stockage. Selon le mode d'accès demandé par le noyau (écriture ou lecture), la page passe dans l'état KWrite ou Read.

KWrite

L'état KWrite indique donc que le noyau est en train d'écrire la page (en fait, les activités qui s'exécutent sur ce noyau) et donc que le gérant de mémoire ne possède pas d'image à jour de la page. Si le gérant restreint les droits du noyau, celui−ci doit la renvoyer. Elle passera donc (du point de vue du gérant) dans l'état Read (la transition par Clean est temporaire). Si le noyau a besoin de place en mémoire principale ou bien si la page n'est plus couplée, il la renvoie au gérant. Elle passe alors dans l'état SWrite (la transition par Flush est temporaire).

Read

L'état Read indique que le noyau possède une image de la page à accès en lecture−seule. Le gérant possède donc une image à jour de cette page. La page n'est pas supposée être renvoyée par le noyau. Il lui suffit de la supprimer de la mémoire principale.

SWrite

L'état SWrite indique que le serveur possède une image à jour de la page en espace de pagination. Il peut la donner en mode écriture (KWrite) ou lecture (Read). Il pourra aussi la stocker en mémoire de stockage à la réception du `memory_object_terminate`.

Clean et Flush

Les états Flush et Clean ne sont pas pris en compte pour l'instant. Ils correspondent aux actions à effectuer en cas de réception d'un `clu_commit` ou d'un `clu_abort`. Flush correspond à l'action de renvoi des pages modifiées par le noyau (à la suite d'une réquisition de la part du serveur ou lorsque la grappe n'est plus couplée). Clean correspond à la destruction des pages par le noyau.

IV.2.3 La journalisation

La journalisation assure l'atomicité des opérations de modification des grappes en mémoire de stockage en cas de panne bénigne (par exemple, arrêt ou panne de la machine). Notre but n'est pas de rendre résistant aux pannes le système de fichier Unix, mais plutôt de protéger la mémoire de stockage.

Quand le stockage d'une nouvelle image d'une grappe nécessite l'écriture de plusieurs secteurs du disque, il ne faut surtout pas que quelques unes des pages de la nouvelle image soient sur le disque et pas les autres. On se retrouverait alors avec une image incohérente de la grappe constituée pour partie de pages de la nouvelle image et de pages de l'ancienne image.

Avant d'effectuer une opération, le service de stockage enregistre l'ensemble des modifications (les pages modifiées présentes dans la mémoire d'exécution) dans un journal (représenté par un fichier), et puis il les répercute sur l'image en mémoire de stockage. Si une panne survient lors de cette re−copie, le service de stockage pourra rejouer les opérations manquantes à partir du journal (reporter les pages modifiées en mémoire de stockage).

Le service de stockage enregistre les modifications sous forme d'opérations élémentaires. Une opération élémentaire est décrite par un enregistrement qui contient l'identificateur de la grappe, le déplacement dans cette grappe ainsi que les pages modifiées. Les descripteurs d'opérations sont alignés sur des pages du disque, de telle manière que l'écriture d'un nouvel enregistrement ne puisse pas altérer l'enregistrement précédent en cas de panne(8).

---

(8) Dans la suite de cette section, on raisonnera en terme d'opération mais il ne faut pas perdre de vue que le journal contient surtout la nouvelle image des pages d'une grappe.

Si une panne se produit lors de l'enregistrement dans le journal, l'opération est considérée comme non exécutée dans la mesure où toutes les informations permettant de la rejouer ne sont pas encore consignées dans le journal. Quand toutes les informations ont été écrites dans le journal, le service de stockage valide l'opération. Si une panne se produit après la validation, l'opération est considérée comme étant réussie. Le cas échéant, les opérations seront rejouées à la reprise. La validation marque donc l'instant à partir duquel une opération est considérée comme effectuée.

IV.2.3.1 Algorithme

La mise en œuvre est basée sur l'algorithme décrit dans [Gray 93]. Le gestionnaire du journal gère deux pointeurs, appelés LSN (Log Sequence Number) qui indiquent respectivement quelles opérations ont été consignées dans le journal (`logLSN`), et quelles opérations ont été répercutées sur l'image (`durableLSN`). Ces deux pointeurs sont maintenus en mémoire centrale en permanence. Ils sont copiés sur le disque au moment de la validation et en fin d'opération.

Ces pointeurs indiquent aussi l'identité du fichier journal dans lequel se trouvent les enregistrements en question, et le déplacement de l'enregistrement à l'intérieur de ce fichier.

Au début d'une opération, les deux pointeurs sont égaux :

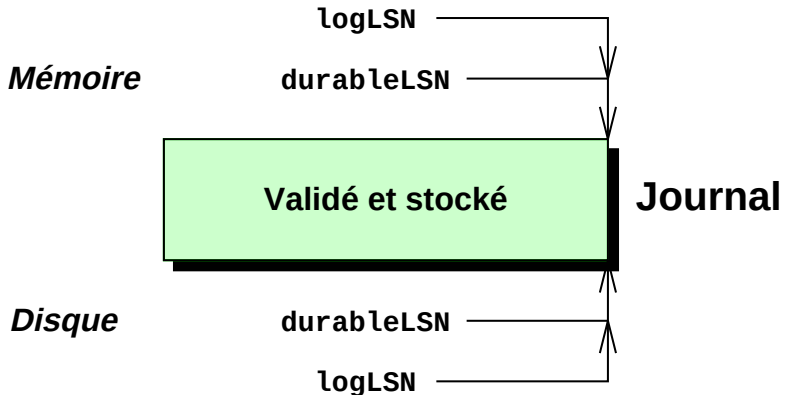

Fig. 4.15 : État initial du journal

Au cours de l'enregistrement des opérations dans le journal, le `logLSN` est modifié en mémoire mais il n'est pas encore reporté sur le disque :

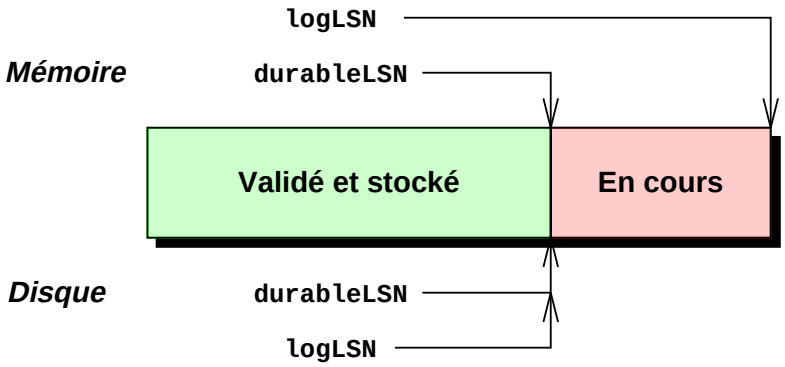

Fig. 4.16 : Enregistrement des modifications dans le journal

Au moment de la validation, les deux pointeurs sont écrits sur le disque dans un fichier particulier appelé ancre :

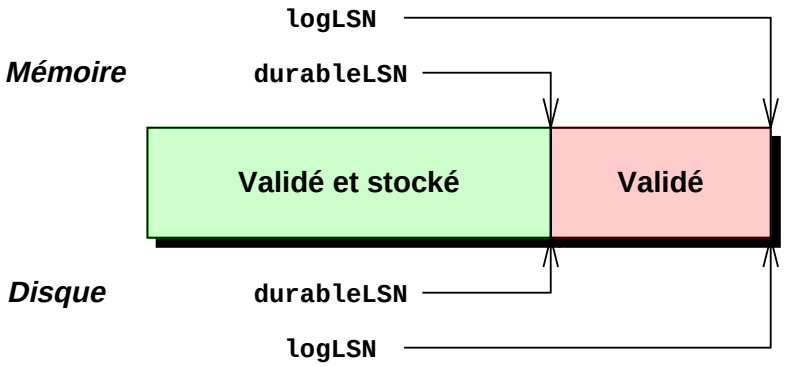

Fig. 4.17 : Validation du journal

Pendant la répercussion des opérations sur l'image en mémoire de stockage, le `durableLSN` est modifié en mémoire, mais il n'est pas encore écrit sur le disque :

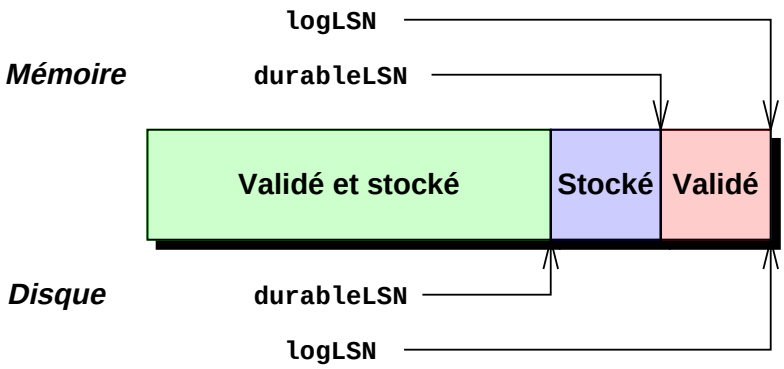

Fig. 4.18 : Mise à jour de l'image persistante

Quand toutes les opérations ont été répercutées sur l'image en mémoire de stockage, les deux pointeurs sont réécrits une seconde fois sur le disque :

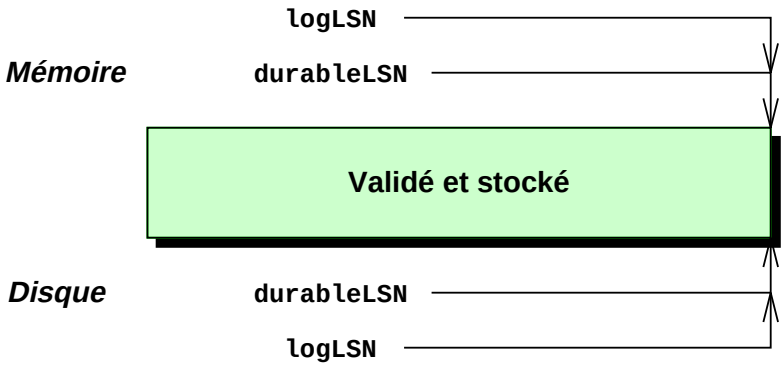

Fig. 4.19 : Mise à jour effectuée et validée

Lors du montage du volume, le gestionnaire du journal consulte le fichier ancre. Si les deux pointeurs sont identiques, cela veut dire ou bien qu'aucune opération n'était en cours pendant la panne (cas représenté par la figure Fig. 4.19), ou bien qu'une opération était en cours, mais qu'elle n'a pas encore été validée et doit donc être ignorée (c'est le cas pour la figure Fig. 4.16). Dans ce dernier cas, le journal est tronquée à la partie validée et stockée (on retire la partie entre le `durableLSN` et le `logLSN`).

Si le `logLSN` est plus avancé que le `durableLSN` (cas représenté par la figure Fig. 4.18), cela veut dire que la panne s'est passée alors qu'une opération validée était en train d'être répercutée en mémoire de stockage. Le gestionnaire du journal rejoue donc toutes les opérations élémentaires (il répercute une nouvelle fois les modifications comprises entre le `durableLSN` et le `logLSN`), y compris celles qui

ont déjà été effectuées avant la panne. Cette duplication d'opérations élémentaires n'est pas gênante car toutes les opérations élémentaires sont idempotentes.

Pour éviter qu'une opération élémentaire altère des données se trouvant sur le même secteur du disque, les opérations portent toujours sur des secteurs entiers du disque.

Pour éviter une perte du fichier ancre lors de sa modification, l'ancre est alternativement écrite dans la page 0 ou 1 du fichier ancre. Comme cela on garantit qu'une au moins des deux versions de l'ancre est intacte. Lors de la reprise, le système utilise la version intacte la plus récente.

IV.2.3.2 Limitations

Si on change la taille d'un fichier ou si l'on remplit une partie creuse d'un fichier, le système de fichiers Unix doit changer la table d'implantation. Si la panne se produit juste au moment où un secteur de la table d'implantation est écrit, ce secteur pourrait devenir illisible. Du coup le fichier tout entier est perdu, car le système ne sait plus trouver les secteurs qui appartiennent à ce fichier. Des pannes de ce genre sont irrécupérables, car le gestionnaire du journal ne peut pas faire de copie de sauvegarde de la table d'implantation.

Une méthode pour s'en protéger serait de faire en sorte de ne jamais avoir de fichiers creux et de faire une sauvegarde du fichier chaque fois qu'on en change la taille.

Le coût d'une telle solution est très important notamment lors de la création des grappes. C'est pourquoi dans la première réalisation du gestionnaire de journal, nous avons considéré que ce type de panne était suffisamment rare pour être négligé.

Si une panne se produit en cours de création d'un volume, ou en cours de destruction (volontaire) d'un volume, elle ne peut pas être récupérée par le journal. Ceci n'est pas très gênant, car un volume nouvellement créé ne contient pas encore de données, et la perte d'un volume qu'on voulait détruire de toute manière n'est pas gênante.

IV.2.3.3 Élimination des anciens journaux

Le journal se compose de plusieurs fichiers. Dès que le fichier journal courant est suffisamment gros (environ 200 Ko), un nouveau fichier journal est créé. Les fichiers journaux qui sont "suffisamment anciens" peuvent être détruits. Un fichier journal est suffisamment ancien si tous les enregistrements qu'il contient, ont déjà été répercutés sur l'image en mémoire de stockage. L'identité des fichiers journaux suffisamment vieux peut être déterminée en examinant le `durableLsn`. Le module de journalisation exporte une fonction qui détruit automatiquement les anciens fichiers journaux.

IV.3 Conclusion

Dans ce chapitre nous avons présenté le support de la persistance dans le noyau de système réparti Guide−2. Nous avons ainsi défini les principaux concepts manipulés par le système :

- La grappe qui est un regroupement d'objets, représentée et manipulée par le système sous la forme d'un ensemble logiquement contigu de pages.
- Le volume qui est un container de conservation des grappes (comme l'est une partition du disque d'un ensemble de fichiers).

La mise en œuvre de ces différents concepts et la réalisation des fonctions de support de la persistance sont l'apanage des gérants de mémoire. Les gérants de mémoire sont des serveurs systèmes, répartis sur l'ensemble des machines disponibles, dont le double rôle est :

- La gestion du partage des grappes entre les applications (par couplage dans l'espace d'adressage des contextes).
- La gestion du stockage des grappes en espace persistant (dans les volumes).

L'ensemble des gérants de mémoire forme la machine à grappes. La machine à grappes met en œuvre le modèle de mémoire persistante du système Guide.

La machine à grappes offre les mécanismes de base sur lesquels les couches applicatives (par exemple, la machine à objets et les applications) peuvent bâtir des politiques ad hoc ; notamment en ce qui concerne le regroupement des objets dans les grappes et le ramassage des miettes (la destruction des objets inutilisés).

La mise en œuvre du support de la persistance par un ensemble de gérants permet de répartir la charge associée à ce support sur plusieurs machines. Le système est alors à même de supporter un plus grand nombre d'applications que s'il utilisait une solution centralisée. Par ailleurs, cette approche nous permet d'accroître la souplesse de notre système car la configuration matérielle peut évoluer en cours d'exécution (par exemple, par l'ajout ou le retrait d'un gérant sur un site).

La principale caractéristique de notre modèle de mémoire réside dans la définition d'un modèle de mémoire uniforme structurée en deux niveaux. D'un point de vue conceptuel et si l'on suit la définition de Thatte, le modèle de mémoire est uniforme. Par contre, nous avons clairement fait ressortir la double structuration de notre mémoire en un espace de travail dit mémoire d'exécution et un espace de conservation dit mémoire de stockage. Cette approche est différente de celle suivie par les systèmes Cricket et O_2 qui mettent en œuvre des niveaux de mémoire équivalents aux nôtres (utilisation de pages fantômes pour garantir la cohérence de l'espace persistant, par exemple) sans les définir explicitement.

Le couplage d'une grappe est explicite mais son chargement en mémoire d'exécution est réalisé implicitement au fur et à mesure que des applications accèdent à la grappe. En revanche, les modifications effectuées en mémoire d'exécution ne sont reportées qu'à des instants particuliers définis par les applications, afin de garantir la cohérence de la mémoire de stockage.

L'intérêt de notre approche est qu'elle offre une définition claire et précise des deux niveaux de mémoire qui permet au programmeur d'applications de mieux identifier les mouvements d'information (tout en le déchargeant de leur gestion). Il peut alors définir des points de cohérence entre les deux espaces en évitant de recourir à un modèle transactionnel complexe à appréhender.

Notre modèle de mémoire nous offre :

- une modularité accrue du système par le découpage en deux niveaux de mémoire et la définition d'une interface de transfert explicite, qui permet notamment l'utilisation de gérants de stockage externes au système.
- une définition claire de la cohérence respective des deux niveaux de mémoire.

Il est intéressant de faire le parallèle entre la notion de grappe telle que nous l'avons définie et la notion de segments définie dans le système Clouds. Les deux concepts sont en effet similaires car :

- Ils définissent l'unité de partage et de persistance du système.
- Le système associe une sémantique de bas niveau à ces concepts, afin de les manipuler de manière simple et efficace.
- Les langages de programmation utilise la grappe ou le segment comme un espace de mémoire persistante, couplé dans la mémoire virtuelle des applications, dans lequel ils peuvent placer des objets.

L'architecture des deux systèmes est d'ailleurs très voisine dans la mesure où la gestion des grappes est réalisée par les gérants de mémoire, qui sont des serveurs de pagination externe vus du micro−noyau Mach, alors que la gestion des segments est réalisée par des partitions Clouds qui sont elles aussi vues par le noyau Ra comme des serveurs de pagination externe.

Les interactions entre les serveurs de pagination et les noyaux des deux systèmes sont basées sur une interface de pagination (toutes les requêtes et les traitements

associés manipulent des pages). Le serveur de base de données Cricket est lui aussi mis en œuvre par un paginateur externe Mach gérant le couplage de la base de données dans l'espace d'adressage des clients.

La réalisation des gérants de mémoire sous la forme de paginateurs externes a les avantages suivants :

- Une intégration efficace et souple de la gestion de la persistance à la gestion de la mémoire virtuelle du noyau Mach.
- Une protection forte du système puisque la gestion du partage et les fonctions d'accès à l'espace persistant sont isolées dans les gérants de mémoire. Les applications n'accèdent pas directement à l'espace persistant ce qui renforce la protection des grappes.
- L'isolation entre utilisateurs et applications est par ailleurs mise en œuvre par le gérant de mémoire grâce aux informations associées aux grappes telles que le propriétaire ou le type de la grappe que lui seul manipule.
- Les grappes permettent une factorisation des opérations de couplage (un seul couplage pour plusieurs objets) qui sont des opérations coûteuses. De plus, elles limitent la consommation des ressources systèmes nécessaire à chaque couplage (notamment les ports).
- Au niveau des performances, le regroupement d'objets réduit le nombre d'opérations d'entrée/sortie sur disque.

Chapitre V

Evaluation du support de la persistance

L'évaluation du support de la persistance fournie par le système Guide−2 au travers de la machine à grappes se découpera en trois étapes : une évaluation qualitative des fonctions offertes doublée d'une comparaison avec celles des systèmes similaires présentés au chapitre II, une évaluation quantitative des performances de la machine à grappes et enfin une brève évaluation du support fourni par le micro−noyau Mach à la construction de cette machine.

V.1 Évaluation qualitative

Le tableau suivant nous sert à récapituler les principales caractéristiques de la machine à grappes.

		La Machine à Grappes
Désignation	Type	Physique
	Entité	Grappe
Partage	Type	Couplage en mémoire partagée
	Entité	Grappe
	Concurrence	Sémaphores
Stockage	Chargement	A la demande des pages
	Intégrité	Journal
	Gestion du disque	Fichier

		La Machine à Grappes
Protection	Isolation des utilisateurs	Notion de propriétaire de grappes + espaces d'adressage différents
	Isolation des applications	Espaces d'adressage différents
	Isolation du système	Serveur

L'architecture de Guide−2 diffère de celle de Clouds dans la manière dont le support de la persistance est distribué sur l'ensemble des machines. Dans Clouds, les serveurs de partitions qui jouent un rôle semblable à celui des gérants de mémoire, sont toujours placés sur des machines Unix différentes des machines Ra qui supportent Clouds. Cela entraîne que toutes les fautes de pages sur un segment seront traitées à distance par un serveur de partition, ce qui peut s'avérer très coûteux.

Les gérants de mémoire peuvent s'exécuter sur n'importe quelle machine Mach/OSF, ce qui signifie que toute machine qui supporte le système Guide peut abriter un gérant de mémoire. Ainsi les volumes locaux à chaque machine, affectés en général à l'utilisateur principal de la machine, seront gérés par le gérant de mémoire de la machine. Toutes les fautes de pages sur ces volumes seront alors traitées localement. D'un autre coté, si une machine puissante est disponible, elle pourra servir de support à des volumes partagés par tous les utilisateurs.

L'avantage de cette architecture est que la charge associée au support de la persistance est partagée entre les différentes machines et n'est pas concentrée sur une seule machine (Cricket et O_2) ou sur un nombre limité (Clouds).

Pour ce qui est des fonctions fournies, la machine à grappes semble très proche d'un serveur de page du type O_2Engine. Cela est particulièrement remarquable lorsque l'on étudie les points suivants :

- La machine à grappes définit et gère une entité non typée, la grappe, composée d'un ensemble de pages. La grappe peut donc être vue comme une partition d'une base de données.
- L'interface entre les clients et la machine à grappes (plus exactement les gérants de mémoire) est basée sur le transfert de pages.
- Les fonctions de sauvegarde sont encapsulées et protégées dans des serveurs externes aux applications.

Les différences sont les suivantes :

- Au contraire des clients O_2, ce ne sont pas les clients des gérants de mémoire qui gèrent leur cache local (représenté par la mémoire centrale de la

machine), mais le noyau Mach. Ainsi, si plusieurs clients partagent une même grappe et accèdent à la même page, le gérant ne recevra qu'une seule faute de page. La réalisation d'un contrôle transactionnel de la concurrence ne peut donc pas être basée sur la seule réception des fautes de pages. Il est nécessaire de gérer le verrouillage des pages dans un serveur ou une couche de verrouillage externe (comme cela est fait dans le système Casper [Vaughan 92]).

- Les systèmes de base de données sont capables de recopier de manière atomique un ensemble de données en mémoire de stockage.

  La nature répartie de la machine à grappes (ensemble de gérants de mémoire) pose en revanche un problème pour la mise en œuvre de la re−copie atomique d'un ensemble de grappes.

  Deux solutions sont possibles : soit se limiter à une seule grappe pour représenter l'ensemble de la base, mais alors l'intérêt du partage de charge entre les différents gérants de mémoire et de la protection entre les utilisateurs est perdu, soit mettre en œuvre un protocole de validation à deux phases [Gray 93] entre les gérants de mémoire ou entre les parties clientes de la machine à grappes et les gérants de mémoire.
- La protection basée sur la notion de propriétaire de grappes est assurée de manière sûre par la machine à grappes.

Dans la réalisation actuelle de la machine à grappes deux points n'ont pas été traités : la primitive `clu_Commit` qui doit permettre aux applications de forcer la copie d'une grappe en mémoire de stockage (cf sous−section IV.1.2.4) est en cours de développement et de mise au point, et la définition de grappes temporaires.

V.1.1 Support des grappes temporaires

Les grappes ont été définies comme étant par défaut persistantes. Or certaines applications peuvent utiliser un grand nombre de données, mais ne vouloir en stocker qu'une faible partie. Ces applications pourraient bénéficier (au niveau des performances) de grappes non persistantes, dites grappes temporaires. Cela permettrait de réduire les échanges entre la mémoire d'exécution et la mémoire de stockage.

On peut distinguer différents types de grappes temporaires :

- les grappes non persistantes et non partageables, qui peuvent être réalisées simplement en gérant un tas dans chaque contexte.
- Les grappes non persistantes et partageables. Cela suppose que les applications sont capables de désigner ce type de grappe afin de pouvoir les partager.

Nous nous intéressons principalement au support des grappes non persistantes et partageables, que nous qualifions de grappes temporaires.

Le principal problème à résoudre est celui de la désignation des grappes temporaires. Il faut être capable d'associer à chaque grappe un identificateur global (afin que le partage soit possible) tout en permettant l'éventuelle promotion d'une grappe temporaire en grappe persistante(1). La manière la plus simple, mais pas la plus efficace, est de faire allouer par le service de stockage un identificateur persistant.

L'allocation de l'identificateur sera ainsi un peu coûteuse mais elle n'hypothéquera pas les performances des applications. De plus, l'allocation d'un identificateur persistant autorise la promotion immédiate de la grappe en une grappe persistante.

Au niveau des gérants de mémoire, les mécanismes de couplage et de traitement des fautes de pages seront les mêmes que ceux utilisés pour les grappes persistantes. En fait, seule l'absence de la phase de stockage (en cas de non promotion) distinguera les grappes temporaires des grappes persistantes pour ce qui concerne les gérants de mémoire.

Les grappes temporaires ne sont pas disponibles dans la version actuelle du système. Leur réalisation ne semblant pas poser de problèmes particulièrement délicats nous envisageons une mise en œuvre dans un futur proche. Il convient cependant de noter que la machine à grappes peut supporter des grappes temporaires, mais que la question de l'utilisation de ces grappes par les applications et les compilateurs est loin d'être résolue en l'état.

V.1.2 Conclusion

Du simple point de vue qualitatif (ou fonctionnel), la machine à grappes fournit tous les mécanismes de bas niveau nécessaires à la mise en œuvre du partage et de la conservation des données. De plus, elle garantit le respect des règles de protection des données.

Les entités gérées par la machine à grappes étant de bas niveau, il appartient aux compilateurs ainsi qu'aux applications de fixer les règles d'utilisation des grappes : par exemple, l'allocation et la désignation des objets utilisés par les applications au travers de langages de programmation ne sont pas du ressort de la machine à grappes.

Ce découpage a l'avantage de rendre le système plus modulaire puisque la machine à grappes définit une sorte de support générique du partage et de la persistance. Mais l'écriture des couches hautes du système est alors plus complexe.

---

(1) Cette fonction semble être recherchée au niveau des applications.

Cependant, le développement des machines à objets et à segments qui ont servies de support aux langages à objets Guide et C++, a montré que la machine à grappes forme une base très solide et suffisamment riche pour bâtir un noyau de système d'exploitation distribué.

Par ailleurs, si les fonctions de la machine à grappes concernant le support des transactions réparties et notamment la re−copie atomique d'un ensemble de grappes sont étendues, au vu des fonctions offertes, la machine à grappes pourrait très bien servir de base à la construction d'une base de données répartie. Ce point qui dépasse largement le cadre de notre étude et qui doit faire l'objet d'études ultérieures nous offre des perspectives de recherche très intéressantes.

V.2 Évaluation quantitative

L'évaluation quantitative de la machine à grappes a été menée en quatre étapes :

- Une instrumentation des grappes permettant de caractériser de manière sommaire l'usage effectif qui en était fait par les quelques applications mises en œuvre.
- La mesure des performances des fonctions de base de la machine à grappes.
- La comparaison entre une architecture de pagination classique utilisant le disque local de la machine et une architecture qui utilise la mémoire centrale d'une machine distante comme espace de pagination.
- L'évaluation du coût de la journalisation sur le stockage des grappes.

Nous pensons que des mesures plus complètes pourrons être menées lorsqu'un plus grand nombre d'applications utilisées par de véritables utilisateurs seront disponibles. Une instrumentation du système devrait alors permettre de le "calibrer" et d'évaluer par la suite le coût de la machine à grappes en utilisation réelle.

Nous allons donc commencer dans la sous−section V.2.1 par donner quelques statistiques d'utilisation obtenues à l'aide des applications de démonstration actuellement disponibles sur notre système. Les mesures sont données à titre purement indicatif et ne permettent pas une généralisation des résultats à d'autres types de comportement d'applications.

Nous donnons ensuite dans la sous−section V.2.2 l'évaluation de la pagination à distance, nous allons comparer l'accès aux pages d'une grappe

Ensuite dans la sous−section V.2.3, nous évaluons les performances d'une pagination basée sur l'utilisation de la mémoire centrale de sites distants. Cette étude ne fait pas, à proprement parlé, partie de nos principaux objectifs mais il nous a semblé

intéressant d'utiliser le prototype à notre disposition afin de mener une première expérimentation.

Enfin, la sous−section V.2.4 concerne la comparaison des différentes versions du service de stockage rapide : avec ou sans journalisation, avec ou sans synchronisation des opérations sur disques. Cette étude est basée sur l'utilisation d'un banc d'essai naïf : la création et la lecture séquentielle de plusieurs grappes.

V.2.1 Instrumentation

Afin de jeter les bases d'une instrumentation du système, la machine à grappes mémorise dans sa version actuelle un certain nombre de paramètres qui caractérisent l'utilisation des grappes. Les principaux paramètres sont :

- Le temps moyen d'utilisation (de couplage) de la grappe.
- Le taux de partage de la grappe, c'est à dire le nombre moyen de contextes qui ont couplé la grappe à un instant donné.
- Le nombre moyen de pages modifiées par utilisation.
- Le nombre moyen de pages recopiées en espace de pagination par utilisation.

Ces informations sont stockées dans le descripteur associé à la grappe.

A titre d'exemple, nous donnons les principales valeurs obtenues avec les trois principales applications réparties de démonstration du système : un tableur multi−utilisateur [Chevalier 94a], les tours de Hanoï et l'éditeur coopératif de documents structurés Griffon[Decouchant 93] .

Utilisation des grappes		**Tableur**		**Hanoï**		**Griffon**	
		Grappes protégées	Grappes partageables	Grappes protégées	Grappes partageables	Grappes protégées	Grappes partageables
Nombre de grappes		7	15	6	5	56	27
Taille (en pages)	Moyenne	21	44	11	28	12	101
	Maximale	57	109	11	35	62	302
Pages modifiées	Moyenne	7	4	1	3	6	10
	Maximale	25	28	1	9	34	79

Nous avons séparé les grappes protégées des grappes partageables car dans la pratique les premières sont utilisées pour stocker des données alors que les secondes sont utilisées pour stocker du code.

Le nombre de grappes partageables créées est important par rapport à ce que nous envisagions (environ trois grappes par application). En fait, cela est dû à la manière dont sont générées les classes des applications. Le programme de génération de l'application que nous avons utilisé, crée une nouvelle grappe par classe. Dans la pratique, nous pensons qu'il est nécessaire de regrouper toutes les classes d'une même application dans une seule grappe (une version particulière du programme de génération permet ce regroupement).

On peut faire la même constatation pour les grappes contenant des données, et notamment dans le cas de l'éditeur coopératif. La création de 5 documents par 3 utilisateurs entraîne la création de 56 grappes, ce qui revient, pour des raisons de protection, à environ 3 grappes par document et par utilisateur. Ce chiffre elevé montre bien la nécessité d'outils facilitant la gestion des grappes par les concepteurs d'applications.

Par ailleurs, le programme de génération demande la création de grappes de taille beaucoup plus grandes que la taille réellement utilisée. Par exemple, pour l'éditeur, seules 79 pages de la grappe de taille maximale (302 pages) sont utilisées. Pour les grappes contenant des données, seule la moitié des pages environ est utilisée. Ce dysfonctionnement est dû à la manière dont est estimée la taille de la grappe par les couches hautes de la machine virtuelle Guide en fonction des informations passées par le programme de génération.

Le nombre moyen de pages modifiées dans les grappes partageables est supposé aller décroissant. En effet, une fois la totalité du code généré, elles ne seront plus modifiées au cours des exécutions.

L'application Hanoï ne modifie, par exécution, qu'une seule page des grappes protégées. Cela est dû au fait qu'il n'y a qu'un ou deux objets par grappe afin de pouvoir répartir l'exécution en plaçant les différents objets sur différentes machines. Comme pour les grappes partageables, la taille estimée à la création est beaucoup plus grande que la taille réellement utilisée. Cela permet la création de nouveaux objets sans provoquer un agrandissement systématique de la grappe.

V.2.2 Fonctions de la machine à grappes

La table suivante donne des mesures préliminaires obtenues sur des machines Zénith i486 possédant une vitesse d'horloge de 33 MHz avec 24 Mo de mémoire centrale (dont 12 à 14 Mo sont disponibles pour les applications) et sur lesquelles s'exécutent un noyau Mach 3.0 (Norma MK14) et le système OSF/1 MK (V4.1). Les disques utilisés sont des disques IDE dont le débit maximal en lecture est de 860 Ko/s.

Les chiffres donnés indiquent le temps, passé vu d'une application, dans les fonctions de la machine à grappes : la création d'une grappe, le couplage d'une grappe et le traitement d'une faute de page en écriture ou en lecture.

Machine à grappes	temps (en ms)
`clu_Create`(2)	186,0
`clu_Map`	8,0
`page_fault`(3)	3,8

La création d'une grappe est une opération coûteuse car elle demande l'allocation d'une référence unique, d'un descripteur persistant de grappe ainsi que la création du fichier Unix qui représente la grappe en mémoire de stockage. Le surcoût dû à la journalisation de l'opération de création est de 21 %.

Le couplage d'une grappe implique la lecture de son descripteur afin de connaître sa taille et son propriétaire.

Les mesures relatives aux défauts de page ainsi qu'au couplage prennent en compte la gestion du cache de page associé au système de fichiers Unix, ainsi que la lecture par anticipation des pages (politique "read−ahead" du système de fichiers).

V.2.3 Pagination locale et pagination à distance

Avant de nous intéresser à l'évaluation de la pagination à distance, nous comparerons l'accès aux pages d'une grappe à l'accès aux pages d'une région de mémoire gérée par le paginateur du noyau Mach appelé "i−node pager". Les mesures suivantes nous permettent de situer notre mise en œuvre relativement à celle du noyau Mach.

Pagination locale

Pour mener à bien cette comparaison, il nous a fallu être capable de mesurer le coût d'une faute de page (du point de vue de l'utilisateur) lorsque la page est placée en espace de pagination. Nous avons employé la technique suivante :

1. Création et initialisation d'une grappe (resp. allocation d'une région mémoire) de taille 100 pages. La grappe (resp. la région) étant petite, 400 Ko, toutes les pages sont en mémoire centrale.
2. Allocation d'une région mémoire de taille 32 Mo (8000 pages) utilisée pour saturer la mémoire centrale. La région utilisée pour la saturation étant plus grande que la mémoire centrale disponible, l'initialisation de ses pages provoque l'élimination des pages de la grappe (resp. région) créée et initialisée en mémoire centrale. Les pages sont alors stockées par le gérant de mémoire (resp. le paginateur noyau) dans l'espace de pagination.

---

(2) Les grappes utilisées font 20 pages de long (soit 81920 octets).

(3) La page est transférée depuis la mémoire de stockage dans la mémoire d'exécution.

3. L'accès séquentiel aux pages de la grappe (resp. région) provoque une série de défauts de page. On mesure donc le temps total d'accès en lecture ou en écriture aux 100 pages de la grappe.

Les mesures, récapitulées dans la table suivante, ont été obtenues selon deux configurations : une où le cache du système de fichiers Unix contenait éventuellement les pages et l'autre où l'on s'assurait que le cache avait été vidé (ceci grâce au chargement d'un gros fichier avant la mesure). Tous les temps sont donnés en millisecondes.

Gestion de la pagination	**Gérant de mémoire**		**i−node pager**	
	avec cache	cache vidé	avec cache	cache vidé
`page_init`	1,3	1,3	0,4	0,4
`page_fault`	6,5	13,6	19,8	14,0

On constate que le i−node pager est plus efficace en ce qui concerne l'initialisation des pages, car la région de mémoire n'existant pas il suffit de laisser le noyau allouer des pages vides pour la région. Le gérant de mémoire doit lui tester s'il s'agit d'une nouvelle grappe ou d'une grappe déjà présente sur le disque.

On constate que lorsque le cache est activé, le gérant de mémoire est beaucoup plus efficace que le paginateur noyau. Le gérant bénéficie de la gestion évoluée des blocs du disque mise en œuvre par Unix (et notamment de la lecture des pages par anticipation). Lorsque le cache n'est pas vidé, le paginateur noyau est handicapé par le fait que les pages sont considérées comme présentes dans le cache alors que le cache est lui même paginé, il se produit donc une double faute de page.

Lorsque le cache a été vidé avant la mesure, les temps obtenus sont similaires. Ces chiffres sont d'ailleurs plus significatifs que ceux obtenus avec le cache activé dans la mesure où dans le cas d'applications réelles il n'y aura pas une aussi importante localité dans les fautes de pages (reçues par le gérant de mémoire). Dans la pratique, le cache Unix associé à la politique de lecture des pages par anticipation devrait améliorer les performances mais pas de manière aussi importante que celle illustrée par les mesures que nous avons faites.

Nous sommes très satisfaits de ces mesures car elles montrent que la gestion de la pagination en espace utilisateur (par le gérant de mémoire) ne pénalise pas les performances. Ceci nous permet de mettre en œuvre des grappes persistantes sans ralentir l'exécution. On trouve d'ailleurs dans [Golub 91] une proposition qui vise à sortir le paginateur système du noyau afin de rendre sa réalisation plus simple (car pouvant bénéficier des services du système Unix) et plus efficace.

Pagination à distance

Nous avons conduit une étude préliminaire visant à évaluer l'intérêt d'une pagination à distance et notamment de l'utilisation de serveurs de pagination utilisant la mémoire centrale de leur site comme espace de pagination.

L'architecture répartie de la machine à grappes nous a permis de mener à bien une première étude sur ce sujet. Nous avons utilisé le même principe que précédemment en plaçant l'application cliente qui couple la grappe sur une station de travail et le gérant de mémoire sur une autre. Notons que seule la mémoire centrale de la station cliente se trouvait saturée (afin de forcer le renvoi des pages au gérant).

Nous cherchions essentiellement à définir une base de comparaison entre la pagination classique (sur disque local) et la pagination en mémoire centrale (dans l'espace d'adressage du gérant) au travers du réseau. Nous avons donc mis en œuvre un gérant de mémoire particulier qui n'utilise pas un fichier comme espace de pagination mais une zone de mémoire pré−allouée.

Les mesures obtenues (en millisecondes) pour le parcours de 100 pages sont les suivantes (dans les deux premières configurations, le cache Unix était vidé avant les mesures) :

Pagination distante	**Pagination disque locale**	**Pagination disque distante**	**Pagination mémoire à distance**
`page_fault`	13,6	18,5	15,8

Le coût d'une faute de page traitée à distance et en mémoire centrale par un gérant mémoire peut se décomposer comme suit : environ 10 ms pour l'envoi de la demande et le renvoi de la page au niveau des primitives de communication Mach, auxquels on ajoute 5,8 ms pour le traitement de la demande par le gérant de mémoire. Ces 6 ms sont à rapprocher du coût d'une faute de page traitée localement lorsque le cache Unix est actif (6,5 ms). Le coût du traitement sur le serveur est légèrement moindre car la mémoire centrale du serveur n'est pas saturée.

Le coût de la pagination disque à distance peut sembler proche de celui de la pagination mémoire. Il ne faut pas oublier que l'on bénéficie au maximum de la gestion du système de fichiers Unix et notamment de la politique de lecture par anticipation. Dans la pratique, le caractère séquentiel des fautes de pages sera moins important, ce qui diminuera les effets bénéfiques du cache et de la lecture anticipée.

Si on compare les débits entre la pagination disque locale et la pagination mémoire distante, on obtient 278 Ko/s en local contre 262 Ko/s à distance (le débit des primitives de communication Mach étant de 383 Ko/s). La pagination en mémoire centrale semble donc offrir une alternative réaliste à la pagination disque. De plus, la version du gérant de mémoire utilisée pour la pagination en mémoire centrale ne tire pas pleinement partie de cette stratégie notamment dans la gestion de l'espace de

pagination et des tampons de transfert. Nous pensons qu'il est possible d'améliorer encore un peu les performances.

Les mesures obtenues confirment qu'il est possible d'avoir de bonnes performances en paginant à distance, notamment lorsque la mémoire centrale d'un site est limitée. En fonction des configurations matérielles disponibles, l'administrateur d'un système Guide aura donc intérêt à ne pas placer de gérant de mémoire sur certains sites et à affecter d'autres sites comme serveurs de pagination et de stockage des grappes.

V.2.4 Impact de la journalisation sur le service de stockage

Nous avons cherché à évaluer le coût de la journalisation sur les performances du service de stockage. Nous avons donc essayé de mesurer le coût de la primitive de re−copie des modifications en mémoire de stockage (`vol_CluWrite`) avec ou sans utilisation du journal. A cet effet, nous avons conduit deux séries de mesures :

- La première série consiste à mesurer le temps passé dans l'écriture page par page de 100 (jusqu'à 1000) pages. Il s'agit du cas le plus défavorable car en temps normal le système tire parti de la contiguïté physique des pages pour diminuer le nombre des entrées/sorties.
- La deuxième série consiste à mesurer le temps passé dans l'écriture de 100 (jusqu'à 1000) pages en une seule opération. Il s'agit d'un cas très favorable puisqu'il n'y a qu'une seule entrée/sortie lors de la re−copie en mémoire de stockage. C'est aussi un cas de figure plus réaliste car les gérants de mémoire tentent de regrouper au maximum les pages à recopier.

Pour chacune des deux séries de mesures, trois configurations ont été évaluées à l'aide d'un programme simulant le fonctionnement d'un gérant de mémoire en appelant directement le service de stockage :

- Une configuration standard utilisant le journal et "synchronisant" le système de fichiers Unix (on force UFS à vider son cache afin de garantir que les pages modifiées sont bien recopiées sur le disque).
- Une configuration avec journal mais sans synchronisation (ce qui n'est pas une configuration réaliste mais qui permet d'évaluer le coût de la synchronisation des opérations de re−copie sur le disque).
- Une configuration sans journal et sans synchronisation dont les performances sont très proches (3 %) de celles du système de fichiers Unix.

La figure Fig. 5.1 présente les mesures obtenues lors de l'écriture page par page. Les temps donnés mesurent, vu du programme de simulation, le temps total écoulé pour l'écriture de l'ensemble des pages.

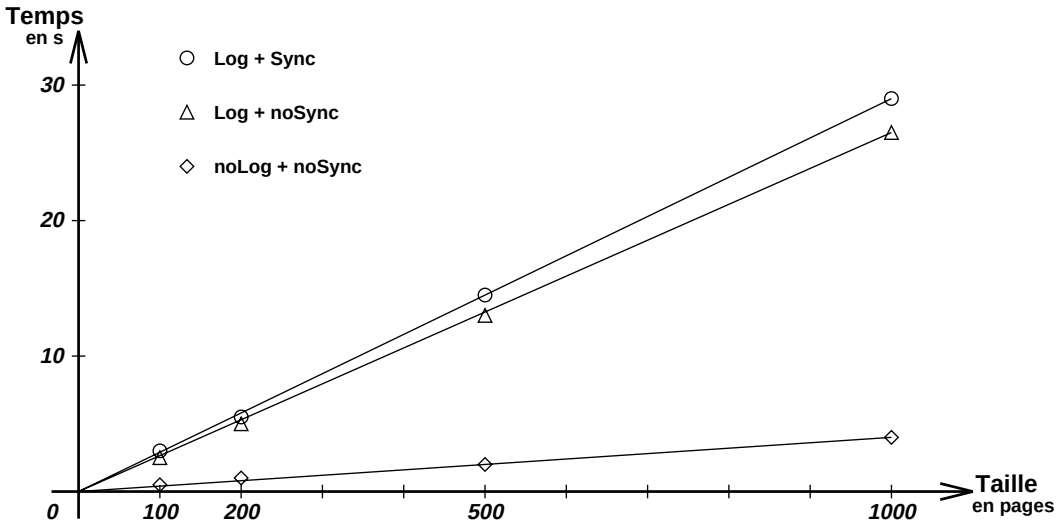

Fig. 5.1 : SSR, écriture page par page

On constate de manière frappante que l'utilisation du journal grève lourdement les performances (diminution d'un facteur 7 par rapport à la configuration sans journal). Ce coût s'explique par le doublement des entrées/sorties (au moins une pour le journal et une en mémoire de stockage) au niveau des pages auquel il faut ajouter le coût du protocole de validation du journal et de l'écriture du fichier ancre (trois entrées/sorties supplémentaires par opération). De plus, chaque écriture d'une page se traduit par une validation du journal et donc par un vidage du cache du système de fichiers.

La différence entre les deux premières configurations est d'environ 10 %. Il s'agit du coût induit par la synchronisation du cache UFS à la fin de chaque opération. Par rapport à une première réalisation du service de stockage, ce coût a largement été diminué grâce à une gestion plus fine des points de synchronisation.

La figure Fig. 5.2 présente les mesures obtenues lors de l'écriture d'un ensemble de pages (en une seule opération `vol_CluWrite`). Par rapport aux mesures précédentes, nous avons rajouté le temps passé vu d'un gérant de mémoire entre la réception d'une requête de découplage (`clu_Unmap`) et la fin de la re−copie des pages en mémoire de stockage (`memory_object_terminate`).

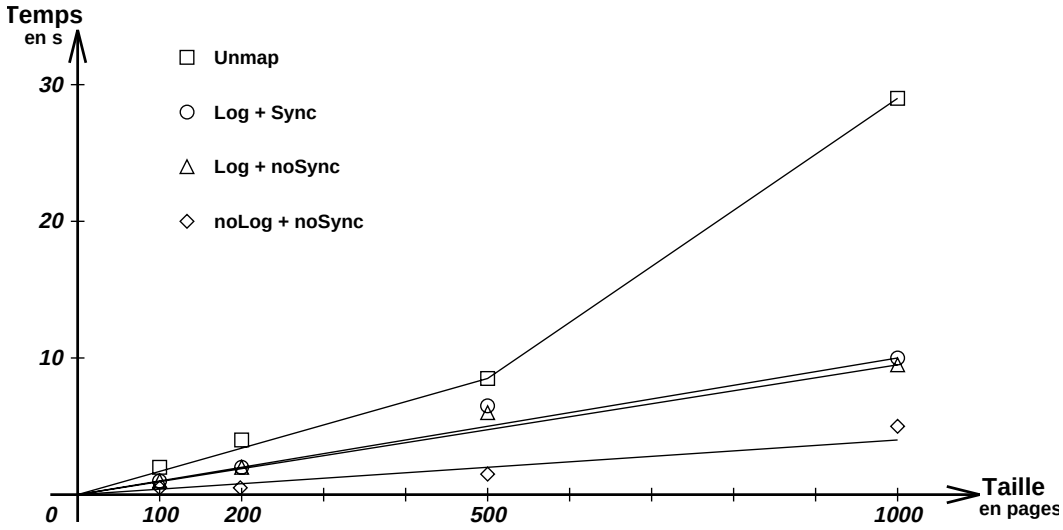

Fig. 5.2 : SSR, écriture de toutes les pages

La diminution du nombre des entrées/sorties réduit de manière notable le rapport entre les configuration avec ou sans journal. Il passe de 7 à 2. Il ne faut tout de même pas oublier que le cas est très favorable puisqu'on effectue qu'une seule opération.

La comparaison entre les performances lors de la recopie d'une grappe (vu du gérant de mémoire) avec les performances obtenues par le programme de simulation de l'activité d'un gérant, nous donne un surcoût de 37 % (pour 500 pages). Ce surcoût comprend le renvoi des pages modifiées par le noyau, le stockage de ces pages dans la partition de pagination(4) ainsi que la libération des structures de données associées à la grappe une fois que celle−ci est recopiée en mémoire de stockage. Il faut noter que les mesures obtenues sont assez fluctuantes, l'écart type étant de ± 1 seconde.

Par ailleurs, on constate une brusque dégradation lorsque la grappe à une taille proche de 1000 pages (soit 4 Mo octets). Cette dégradation est principalement due à l'activité de pagination qui se produit lorsque l'occupation mémoire est trop importante. L'intérêt de cette mesure est qu'elle donne une idée des temps que les applications peuvent s'attendre à trouver lorsqu'une activité de stockage importante est demandée aux gérants de mémoire.

---

(4) Une optimisation possible serait de ne pas recopier les pages dans la partition de pagination lorsque le gérant de mémoire "sait" qu'il va recevoir le `memory_object_terminate`. Mais cela aurait comme conséquence d'augmenter l'occupation de la mémoire centrale pendant le renvoi des pages, tout en introduisant d'autres problèmes supplémentaires liés à la gestion des pages. Cette solution n'a donc pas été mise en œuvre dans la version actuelle des gérants de mémoire.

V.2.5 Conclusion

Les performances obtenues sont très encourageantes surtout lorsque l'on tient compte du faible coût de la plate−forme matérielle et donc des relativement et intrinsèquement basses performances des machines utilisées. On ne peut en effet s'attendre de la part d'une machine compatible IBM/PC à des performances de même niveau que celle d'une station de travail coûtant quatre à cinq fois plus cher à l'achat.

On peut toutefois regretter qu'il n'existe pas ou peu de mesures comparables permettant de situer la machine à grappes par rapport à des systèmes existants. Soit les configurations matérielles sont trop différentes pour permettre la comparaison (c'est le cas pour le système Clouds [Dasgupta 90], par exemple), soit les mesures données concernent des fonctions trop différentes ou de trop haut niveau par rapport aux nôtres (c'est le cas avec le système Cricket [Shekita 90] qui bien qu'également mis en œuvre au dessus du micro−noyau Mach n'offre pas de performances directement comparables).

V.3 Support du micro−noyau

La construction du système Guide−2 a été menée en tirant partie du support du micro−noyau Mach 3.0. Nous estimons que la technologie des micro−noyaux offre un support adéquat pour l'écriture d'un système distribué tel que Guide. On se reportera à [Balter 93] pour une étude plus complète du support fourni par le micro−noyau.

En ce qui concerne le support de la persistance, la possibilité de concevoir notre propre gérant de mémoire est l'un des bénéfices majeurs apporté par l'utilisation du micro−noyau :

- Une mise en œuvre simple et efficace du partage de grappes notamment grâce à la séparation claire entre la grappe, qui définit l'unité de couplage et la page utilisée comme unité de transfert pour les entrées/sorties.
- Une grande souplesse dans la gestion de la mémoire, un gérant de mémoire peut par exemple proposer différentes politiques de gestion des grappes en fonction des demandes effectuées par les applications .
- Une protection des grappes renforcée par l'utilisation de gérants de mémoire. Le contrôle des droits associés aux utilisateurs est réalisé dans un serveur protégé du système. De plus, les grappes ne sont pas manipulables directement par les applications, ce qui accroît leur protection.

Au niveau des fonctions offertes par le noyau Mach pour la gestion de la mémoire, nous avons tiré les conclusions suivantes :

- La mise en œuvre d'une mémoire partagée distribuée (la XMM) directement dans le noyau Mach a simplifié le développement du gérant de mémoire. Nous n'avons ainsi pas eu à nous préoccuper des problèmes de cohérence mémoire entre les différents sites(5) puisque ceux−ci étaient pris en charge par la XMM.
- Le problème qui surgit avec la XMM est que le gérant de mémoire ne peut pas la court−circuiter et qu'elle lui dissimule complètement l'existence des différentes machines. Le gérant de mémoire ne peut alors pas mettre en œuvre de protocole de cohérence mémoire ad hoc (spécifique au besoin de certaines applications) comme celui défini dans [Boyer 91b] et [Perret 93].

- La plus grande lacune concerne la politique de remplacement des pages de la mémoire centrale. Celle−ci est gérée entièrement par le noyau sans le secours des gérants de mémoire. Or, comme il est proposé dans [McNamee 90], il serait très intéressant que le remplacement des pages soit géré par les gérants de mémoire car ceux−ci possèdent une connaissance de l'usage des grappes que le noyau n'a pas. Dans leur version actuelle, les gérants de mémoire collectent et stockent des informations sur l'accès aux pages des grappes qui pourraient rendre la pagination plus efficace.

V.4 Conclusion

Au delà de la présente évaluation et des chiffres que nous venons de donner, il est important de préciser que la machine à grappes définit et fournit essentiellement des mécanismes et que l'utilisation de ces mécanismes est laissée à l'entière liberté des applications (ou des compilateurs). Or la présente évaluation est basée sur une utilisation arbitraire de ces mécanismes, elle ne peut donc prétendre à être exhaustive.

Le prolongement de notre travail serait notamment d'étudier plus spécifiquement les politiques de regroupement des objets dans les grappes et d'évaluer leur influence sur les performances mais aussi de définir des outils de gestion des grappes qui permettent aux utilisateurs d'aider le système à mieux se comporter.

---

(5) Le mécanisme d'extension nous offre aussi une alternative au développement d'un protocole de cohérence mémoire puisque les grappes ne sont couplées que sur un seul site.

Chapitre VI

Vers une fiabilité accrue

La caractéristique principale du support de la persistance dans Guide, comme de tout environnement réparti, est sa souplesse, en grande partie due à l'indépendance physique qui existe entre ses divers composants. Cependant, un tel environnement est intrinsèquement fragile ; la probabilité qu'un au moins des composants (site ou disque) soit en panne à un instant donné augmente avec le nombre des composants.

Lors de la conception du système Guide, nous avons donc été amené à étudier les problèmes de tolérance aux pannes dans les systèmes répartis.

En général, il existe deux niveaux de prise en compte de la fiabilité des systèmes :

- Le niveau des calculs, c'est à dire tout ce qui concerne l'exécution des applications et qui permet de continuer ou de reprendre une exécution défaillante afin de ne pas perdre le travail déjà effectué. A ce niveau, on est principalement sensible aux pannes des machines.
- Le niveau des données, c'est à dire principalement la conservation des données et leur disponibilité. Le principal problème est alors de résister aux pannes des disques et dans une moindre mesure aux pannes des machines qui supportent des disques.

La souplesse de notre système, nous permet de répondre assez naturellement à la panne d'un site en autorisant un fonctionnement dégradé (au niveau des performances) du système : on peut par exemple regrouper ou réexécuter sur un autre site les applications qui s'exécutaient sur le site défaillant. Les techniques employées pour offrir une telle qualité de service sont généralement basées sur les mécanismes de transactions et de points de reprises [Cristian 91].

On peut remarquer que pour que les calculs soient robustes, il est nécessaire que les données soient disponibles. C'est pourquoi, comme point de départ à notre étude sur les problèmes de tolérance aux pannes, nous nous sommes principalement intéressé aux aspects relatifs au support du stockage fiable des données. L'étude et la mise en œuvre des mécanismes fiabilisant les calculs dépassent donc le cadre de ce mémoire.

Pour résoudre le problème de la perte d'un disque et donc des données contenues sur ce disque, on chercher généralement à augmenter la disponibilité des données en les dupliquant sur plusieurs disques (ou sites). On espère ainsi, en cas de panne d'un site, pouvoir accéder à l'un des autres sites possédant une copie de la donnée.

La duplication est une technique générale qui peut aussi bien s'appliquer aux composants matériels que logiciels d'un système informatique. Depuis de nombreuses années, son étude est à l'ordre du jour notamment dans le domaine des bases de données. L'objectif généralement visé par les concepteurs d'un système utilisant la duplication est double :

- augmenter la résistance aux pannes qui perturbent le fonctionnement du système : panne d'une machine, panne d'un disque, etc ... (ainsi, en dupliquant les composants, le système sera capable de continuer à rendre un service malgré la panne d'un ou de plusieurs de ses composants),
- augmenter les performances d'accès aux services fournis par le système. En effet, si une donnée se trouve localisée sur plusieurs sites, il n'est pas forcément nécessaire de s'adresser à un site distant afin de la consulter.

Le premier de ces deux buts s'applique aux systèmes informatiques centralisés ou répartis. Le second est quant à lui d'un intérêt essentiel dans les systèmes répartis dans la mesure où l'on essaie de minimiser l'utilisation des canaux de communication et que l'on privilégie le travail que l'on peut réaliser localement.

On peut remarquer qu'un système réparti fournit un support idéal à la duplication dans la mesure où la redondance matérielle des composants physiques du système autorise la mise en œuvre de cette technique sans bouleverser l'architecture du système.

La section VI.1 décrit les concepts de base relatifs à l'application de la duplication pour la tolérance aux pannes. La section VI.2 présente les principaux algorithmes de gestion de la duplication des données. La section VI.3 fournit une description plus précise de deux classes d'algorithmes : les algorithmes pessimistes et les algorithmes optimistes. La section VI.4 présente, en s'appuyant sur les système de fichiers distribués Coda et Echo, l'application des techniques de gestion de la duplication au stockage fiable des données.

VI.1 Concepts de base et terminologie

L'utilisation de la duplication pour améliorer la tolérance aux pannes d'un système informatique revêt deux aspects :

- Elle permet d'accroître ***la disponibilité du système***, c'est à dire sa capacité à être prêt à délivrer un service (***availability*** en anglais).

- Elle augmente ***sa fiabilité***, c'est à dire sa capacité à assurer, en présence de panne, la continuité du service (***reliability*** en anglais).

On trouvera dans [Laprie 89] une présentation générale des concepts de base et des techniques utilisées dans le domaine de la tolérance aux fautes auquels se rapporte la duplication.

La duplication est mise en œuvre de manière à pouvoir compenser ou résister à un certain nombre de pannes des composants matériels ou logiciels d'un système informatique. Les principales défaillances auxquelles on essaye de résister sont :

- La panne d'un site, c'est à dire l'arrêt d'une machine suite à une erreur matérielle ou logicielle.

  On suppose en général que les machines sont de type "silence sur défaillance" ("Fail−stop") ; soit elles fonctionnent correctement (en rendant des réponses correctes), soit elles s'arrêtent (ou retournent une erreur).
- La panne d'un disque ; le disque n'est alors plus accessible en lecture et toute tentative d'écriture retournera une erreur.
- Le partitionnement du réseau ; le réseau se subdivise en plusieurs parties indépendantes appelées partitions ; les machines situées dans une partition ne peuvent pas communiquer avec les machines situées dans les autres.

  Un cas particulier de partitionnement est celui d'une machine trop lente qui ne peut plus recevoir de messages faute de ressources en quantité sufisante. Elle sera alors considérée comme isolée par le reste du système.

Pour ce qui est du stockage des données, il est important de bien distinguer les deux concepts de disponibilité et de fiabilité. Fournir un système fiable, c'est garantir que l'utilisateur ne peut pas perdre ou corrompre ses données. Fournir un système disponible, c'est assurer que l'utilisateur pourra accéder à ses données.

La technique de base pour augmenter la fiabilité d'un système est de dupliquer des données en espace de stockage : le système conserve plus d'un exemplaire des données. Le nombre d'exemplaires est fonction de l'importance de la donnée, de la probabilité d'une erreur du disque et du coût des disques.

La technique de base pour fournir de la disponibilité est de dupliquer les chemins d'accès à l'espace de stockage. Pour ce faire, on peut dupliquer les serveurs de stockage mais aussi les contrôleurs de disque dans le cas des disques miroirs ("dual−ported"). Le système Voltan [Shrivastava 92] et la FTM [Banâtre 91] sont des exemples d'architectures dans lesquelles les chemins d'accès sont doublés.

Un autre intérêt de la duplication des données, appliquée dans le domaine des bases de données, est l'amélioration des performances. L'accès en lecture à une donnée, dont une copie est présente localement, est en effet beaucoup plus rapide que

l'accès à disance (si le réseau de communication est étendu). On cherche donc à minimiser les communications au profit du travail local. Or dupliquer une donnée, signifie la conserver en plus d'un endroit, on va tenter de dupliquer la donnée sur tous les sites qui risquent de l'utiliser. On espère ainsi augmenter la probabilité d'accès local à la donnée.

Par exemple, le catalogue d'un système informatique (c'est à dire, la liste des objets accessibles par un nom symbolique) contient des informations auxquelles on accède fréquemment ; en stockant une copie du catalogue sur chaque site, sa consultation est toujours exécutée localement. La mise à jour du catalogue sera coûteuse dans la mesure où elle doit être réalisée sur chaque copie. Le catalogue appartient cependant à la classe des informations qui sont plus lues que modifiées et donc un temps de mise à jour important peut être toléré.

Certains auteurs comme Mutchler et Jajodia [Jajodia 89] n'hésitent d'ailleurs pas à considérer que le développement d'un système gérant la duplication doit impérativement viser à diminuer le temps d'accès aux données. On doit cependant noter que certains problèmes de maintien de la cohérence des copies induits par la duplication rendent parfois caduc un objectif aussi ambitieux.

VI.2 De la duplication appliquée au stockage des données

Le stockage de plusieurs copies d'une même donnée, avec une politique de maintien de la cohérence entre les copies plus ou moins stricte, vise à augmenter la disponibilité de cette donnée.

Cependant, cohérence et disponibilité ne sont pas deux buts indépendants. La cohérence peut, par exemple, être assurée en suspendant les modifications dans toutes les partitions sauf dans une qui sera chargée de diffuser les modifications lorsque les partitions se regrouperont. Mais cette solution compromet sérieusement la disponibilité. Pour certaines applications comme la réservation aérienne cette caractéristique est inacceptable dans la mesure où refuser une réservation, sous le prétexte que le réseau est partitionné, entraîne la perte du client. D'un autre coté, permettre à toutes les partitions de modifier les données assure une grande disponibilité mais ne permet pas de garantir la cohérence, ce qui pour certains types de données (les comptes bancaires, par exemple) peut avoir des conséquences fâcheuses.

Davidson a abordé de façon très complète ces problèmes dans le cadre de transactions accédant à une base de données dupliquées [Davidson 89]. Le cadre de son travail est quelque peu différent du notre, cependant les problèmes à résoudre et les solutions employées sont généralement semblables.

VI.2.1 La cohérence des copies

Lorsque qu'un serveur de stockage stocke plusieurs copies des objets[(1)] dont il a la charge, il cherche en général à masquer la duplication aux clients. Cette stratégie est mise en œuvre en distinguant ***l'objet logique*** auquel les clients accèdent, ***des objets physiques*** qui sont réellement stockés sur les différents supports. La figure Fig. 6.1 illustre la transparence de la mise en œuvre.

Dans un système sans duplication, lorsqu'une application veut accéder à un objet, elle essaie de verrouiller l'objet physique, ce qui revient à poser un verrou sur l'objet logique. Mais si l'objet logique est représenté par plusieurs copies physiques, deux applications situées sur deux sites différents peuvent alors verrouiller des copies physiques différentes. La cohérence de l'objet logique ne sera alors plus garantie, puisque les deux copies peuvent diverger. La cohérence n'étant plus assurée, l'état logique de l'objet devient indéterminé. C'est pourquoi les algorithmes de gestion de la duplication sont conduits à arbitrer entre les accès concurrents (en verrouillant les objets logiques) afin de garantir que la cohérence logique de l'objet soit maintenue.

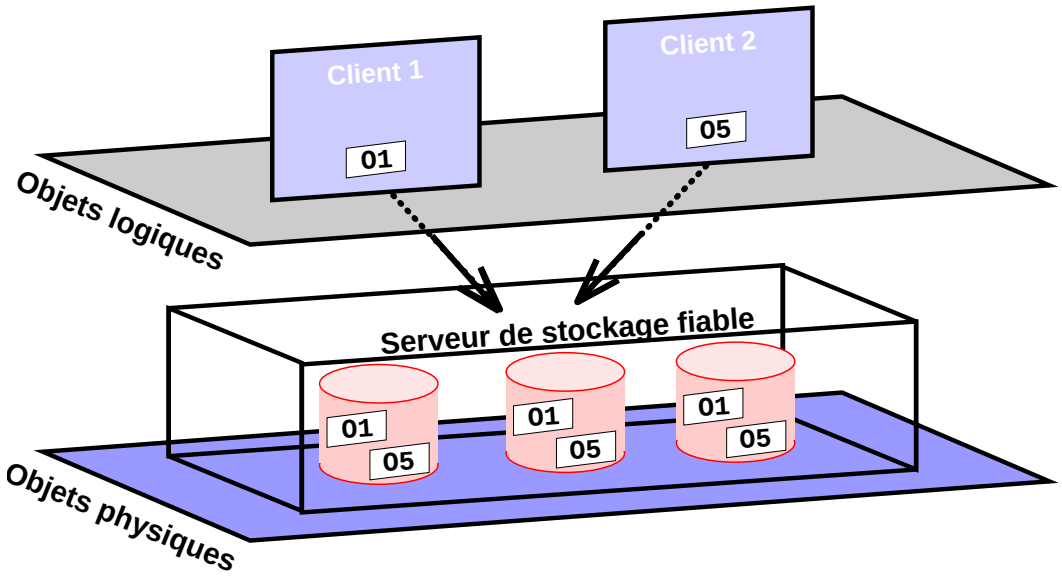

Fig. 6.1 : Transparence de la duplication

(1) Le terme objet est ici utilisé comme définissant une donnée plus ou moins structurée. Il en sera de même dans toute la suite de ce chapitre.

> Les deux clients Client1 et Client2 accèdent respectivement aux objets O1 et O5. La vision que les clients ont des objets est qualifiée de logique car ils se contentent de charger une image des objets dans leur espace d'adressage sans connaître les détails de leur réalisation physique. Le stockage et la gestion de la cohérence des différentes copies physiques des objets sont à la charge du serveur de stockage.

De plus, un système réparti peut être soumis à un partitionnement du réseau en des sous−réseaux indépendants : dans chacun de ces sous−réseaux, les processus qui réalisent le serveur de stockage coopèrent entre eux mais ne peuvent pas étendre cette coopération aux processus des autres sous−réseaux.

En conséquence, la gestion des accès concurrents (c'est à dire le verrouillage d'un objet par les processus d'une même partition) ne suffit pas à elle seule. Une application située dans une partition peut accéder à une copie (après l'avoir verrouillée), pendant qu'une autre application située dans une autre partition accède à une autre copie du même objet. Or, puisque les partitions sont isolées, le contrôle simpliste des accès concurrents ne peut pas interdire ces accès et la cohérence des copies n'est alors plus assurée. Il est donc aussi nécessaire de contrôler la cohérence en cas de panne.

Les algorithmes de gestion de la cohérence des données dupliquées peuvent être classés en trois familles, selon la cohérence qu'ils assurent :

Les protocoles cohérents

Ces protocoles rendent visibles les modifications des données à tous les clients qui y accèdent "*au même moment*". La propriété "*au même moment*" peut être prise au sens d'un temps global ou d'un temps virtuel [Lamport 78]. On trouve dans cette famille les protocoles du type *Copies Disponibles* [Long 90] ou *Quorum* [Golding 91].

Les protocoles à incohérence bornée

Ces protocoles garantissent que les copies seront toutes mises à jour "*au bout d'un certain temps*". Les modifications n'ont donc pas à être propagées de manière synchrone, ce qui permet de gagner en performance, puisque les propagations pourront avoir lieu de manière asynchrone et ce sans trop charger le système de communication. Les protocoles appartenant à cette famille sont par exemple : *"Epsilon−serializability"* [Pu 91], *l'anti−entropie estampillée* ("Timestamped anti−entropy") [Golding 92] et la *duplication paresseuse* ("Lazy Replication") [Ladin 91].

Les protocoles incohérents

Ces protocoles se contentent de propager à faible coût les mises à jour, mais aucune garantie n'est fournie quant à la cohérence des données. Ces protocoles sont généralement utilisés par les applications du type serveur de

noms. En effet, les informations fournies par un serveur de noms sont utilisées comme des indicateurs de localisation éventuellement erronés mais offrant de bonnes performances et d'un coût de remise à jour faible.

Dans le cadre du stockage des données, nous nous intéresserons principalement aux deux premiers types de protocoles.

VI.2.2 Gestion de la cohérence

Un système réparti qui gère la duplication doit donc contrôler les accès concurrents à un objet logique ainsi que la cohérence des copies de cet objet.

Par la suite, on s'intéresse principalement au contrôle de la cohérence et non pas au contrôle des accès concurrents, dans la mesure où ce contrôle peut être effectué au niveau supérieur à la gestion du stockage (idéalement par un gestionnaire de transaction ou par des mécanismes de synchronisation appliqués aux objets logiques). Ainsi, on suppose qu'au sein d'une partition, les accès concurrents sont correctement gérés.

Afin qu'il n'existe aucune ambiguïté sur la "valeur" de l'objet logique, le service de gestion de la duplication doit garantir le maintien de la cohérence des copies physiques. En effet, si deux copies physiques ont des valeurs différentes, il n'est pas aisé de déterminer quelle est la valeur de l'objet logique. C'est pourquoi on cherche à assurer que toutes les copies ont la même valeur.

Cependant un système réparti repose sur un système de communication de fiabilité inférieure à celle des processeurs qu'il connecte. Propager les mises à jour ne garantit alors pas qu'elles seront toutes reçues par les copies, car des pertes de messages et des partitionnements du réseau sont toujours possibles.

Pour garantir la cohérence face à ces différentes pannes, deux approches peuvent alors être utilisées :

a) **rendre le réseau de communication du système fiable**. C'est à dire bâtir un système de communication qui garantisse la réception d'un message par son destinataire en respectant des contraintes temporelles plus ou moins fortes. Les mises à jour pourront alors être propagées sur tous les sites de manière fiable en respectant un certain ordre. Le service de gestion de la duplication est ainsi déchargé de la prise en compte des pannes du système de communication.

   Le système ISIS [Joseph 89] assure par exemple une diffusion fiable et ordonnée des messages d'un site vers d'autres sites. Une approche légèrement différente est celle mise en œuvre pour le système Arjuna [Little 90].

Arjuna n'utilise pas de diffusion fiable et ordonnée au niveau du système de communication, mais impose par contre un ordonnancement au niveau des applications, en se basant sur la propriété de *sérialisabilité* des transactions.

b) **prendre en compte les pannes au niveau du système de gestion de la duplication**, en ne faisant aucune supposition sur la fiabilité du réseau de communication. Cette approche a été utilisée dans Coda [Satyanarayanan 90] ou Echo [Mann 89]. On trouve dans [Davidson 89] une synthèse sur ce sujet.

VI.3 Mise en œuvre de la duplication des données

Le maintien de la cohérence des copies doit principalement résister à deux types de pannes :

- les pannes de site : un site qui tombe en panne, puis qui réintègre le système, doit remettre à jour les copies qu'il gère,
- les partitionnements du réseau : des mises à jour d'un objet dans différentes partitions peuvent en faire diverger les copies.

Lorsqu'un site tombe en panne, les copies qu'il contient ne peuvent plus être mises à jour. Lorsque le site sera réparé, il conviendra de remettre à jour l'ensemble des copies qu'il détient. Cela peut être réalisé par copie de versions valides situées sur un autre site, ou par utilisation d'un journal permettant d'exécuter les opérations que le site a "manquées".

Le partitionnement du réseau est plus délicat à traiter. Les communications entre partitions étant impossibles, les accès concurrents au sein de partitions différentes ne peuvent plus être détectés.

Selon que l'algorithme de maintien de la cohérence résiste ou non au partitionnement du réseau, la cohérence est dite stricte ou lâche. La ***cohérence stricte*** impose que l'ensemble des copies physiques apparaisse comme une seule entité logique. Ce qui signifie que l'ensemble des copies doit avoir la même valeur à un instant donné, et que seule cette valeur est accessible ou modifiable. Pour assurer la cohérence stricte lors d'un partitionnement, les modifications des objets ne seront permises qu'au sein d'une seule partition dans laquelle on suppose la cohérence stricte automatiquement assurée.

Dans certains cas :

- lorsque l'efficacité est un critère prépondérant,
- lorsque la disponibilité des objets doit être très grande,
- lorsque les temps de communication sont importants [Davidson 89] , ou
- lorsque la sémantique des objets n'est pas forcément très stricte et que l'on accepte de laisser des copies diverger comme dans Coda [Satyanarayanan 90],

on préférera assurer une ***cohérence lâche*** (les copies ne reflètent pas forcément la même valeur) qui sera jugée beaucoup moins contraignante et plus efficace que la cohérence stricte.

Deux applications, situées dans des partitions différentes, peuvent alors modifier de manière concurrente un même objet, les mises à jour étant propagées de manière différée aux autres sites. Le gestionnaire de cohérence lâche fournit un mécanisme de détection (et éventuellement de réparation) des conflits lorsque des partitions qui contiennent des copies ayant divergées se regroupent.

On trouve dans la littérature deux grandes classes d'algorithmes selon le type de cohérence mis en œuvre :

- la cohérence lâche est réalisée par les ***algorithmes optimistes***. L'efficacité et surtout la disponibilité sont jugées beaucoup plus importantes qu'une cohérence stricte ressentie comme trop pénalisante.

  Ces algorithmes supposent que les partitionnements du réseau sont rares et peu importants du point de vue des applications et laissent donc s'exécuter librement les accès et les mises à jour.

  Néanmoins un indispensable contrôle des copies a lieu pour déterminer si un conflit s'est produit ou non ; dans ce dernier cas, on déclenche une phase de correction.

- la cohérence stricte est réalisée par les ***algorithmes pessimistes***, qui interdisent les modifications d'objets dans deux partitions au même moment. Ils sont basés sur la supposition que les partitionnements sont dramatiques pour les applications et qu'il convient de les traiter de manière rigoureuse. Ces algorithmes préfèrent réduire la disponibilité au profit d'une cohérence stricte jugée indispensable.

En résumé, les algorithmes pessimistes se caractérisent par une synchronisation des accès aux copies avant les mises à jour, alors que les algorithmes optimistes réalisent cette synchronisation après les mises à jour.

Cette classification est cependant un peu simpliste dans la mesure où il existe des algorithmes qui essayent de tirer partie des avantages d'une approche optimiste,

surtout en ce qui concerne les accès, et de ceux d'une approche pessimiste, pour tout ce qui touche aux modifications.

VI.3.1 Algorithmes pessimistes

Il y a deux approches dans le développement d'un protocole pessimiste de contrôle de la duplication :

- l'approche ***indépendante du contexte*** ("status−oblivious approach"), qui ne tient pas compte de l'état courant du système de communication (i.e. l'état des communications entre processeurs), généralement représentée par les algorithmes de votes [Jajodia 90] ou de quorum [Davidson 89][Golding 91][Long 90], et
- l'approche ***dépendante du contexte*** ("status−dependent approach") qui est basée sur la connaissance qu'ont les processeurs de la topologie courante du réseau. Ainsi, chaque processeur peut décider avec qui il doit communiquer lorsqu'il exécute une opération. La mise en œuvre de cette approche est décrite dans [El Abbadi 85], [Jajodia 89] et [Mann 89].

VI.3.1.1 Le quorum

L'algorithme du quorum (aussi appelé vote majoritaire) repose sur un principe très simple : toute opération de lecture ou d'écriture ne peut être effectuée que lorsqu'un certain nombre de sites (le quorum) est prêt à participer. Le protocole déroule une phase de vote, auprès de l'ensemble des sites qu'il peut contacter, afin de constituer un quorum. Si le quorum de voix est atteint, l'opération est exécutée, sinon elle est abandonnée.

Le protocole résiste aux pannes de sites ainsi qu'aux partitionnements du réseau en ignorant, lors de la phase de vote, les sites en panne ou isolés. C'est un algorithme pessimiste qui fait partie de la classe des algorithmes indépendants du contexte.

Nous allons décrire le fonctionnement de cet algorithme en faisant, à des fins de simplification, abstraction du contrôle des accès concurrents. On considère qu'un client a acquis auparavant le droit de consulter et de modifier un objet.

Tout d'abord examinons le déroulement d'une opération de lecture :

- ♦ Le client envoie à toutes les copies sa demande de lecture, puis il attend un quorum de réponses,
- ♦ Si un quorum de réponses identiques, et à jour(2), a pu être reçu, la lecture a réussi, sinon elle est abandonnée.

(2) on suppose qu'un mécanisme "ad hoc" permet de définir la notion de copie à jour (à l'aide d'un numéro de version, par exemple).

Le déroulement d'une opération d'écriture est sensiblement identique :

- Le client envoie à toutes les copies la nouvelle valeur, puis il attend un quorum d'acquittements,
- Si le quorum est reçu, l'écriture est considérée comme réussie. Un nouveau numéro de version lui est associé et un message de validation de l'écriture est alors envoyé à tous les sites. Si le quorum n'est pas reçu, l'écriture échoue.

La taille du quorum en lecture (notée r) et celle du quorum en écriture (notée w) sont fixées par le protocole en fonction du nombre maximum de votes (noté v) qu'un client peut recevoir. Afin de garantir la cohérence stricte, elles doivent satisfaire deux contraintes :

- $r + w > v$
- $w > v / 2$

La première contrainte assure que les quorums en lecture et en écriture ont toujours une intersection non vide. Ce qui garantit que la version lue sera toujours la dernière écrite et que l'on ne pourra pas lire dans une partition pendant que l'on écrit dans l'autre. La deuxième quant à elle, assure qu'un même objet ne pourra pas être écrit en même temps dans deux partitions, seule la partition qui contient une majorité des copies (si elle existe) peut modifier l'objet (et ce quel que soit le nombre de partitions).

Une propriété intéressante de ce protocole est que l'on peut fixer "l'importance" des différentes copies en affectant à chaque copie un certain nombre de votes. Ainsi, si on accède à la copie 1 plus souvent qu'à la copie 2 ou si le site où elle se trouve est considéré comme très fiable par rapport au site de la copie 2, on pourra lui affecter un plus grand nombre de votes qu'à la copie 2.

Nous allons illustrer le fonctionnement de ce protocole à l'aide d'un exemple. Soit un client qui désire écrire la valeur Y dans un objet dupliqué en cinq copies. Le réseau est supposé non−fiable. L'initiateur est un processus agissant pour le compte du client et qui diffuse tout d'abord une demande d'écriture à l'ensemble des copies, puis qui attend un quorum d'acquittements.

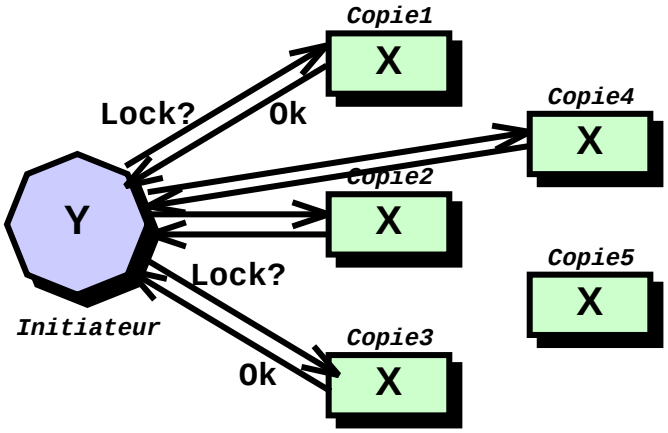

Fig. 6.2 : Phase 1− Demande d'écriture et attente du quorum

On remarque que la copie 5 n'a pas reçu l'opération (soit à cause d'une perte de message, de la panne du site sur lequel elle se trouve ou d'un partitionnement du réseau). Si l'opération réussit, il conviendra donc par la suite de détecter que cette copie est obsolète et de la mettre à jour (à l'aide d'un protocole de restauration des sites tombés en panne, par exemple).

Une fois un quorum de réponses reçu des copies 1, 2, 3 et 4, l'initiateur valide la modification. Les copies qui reçoivent cette validation affectent la valeur Y à l'objet.

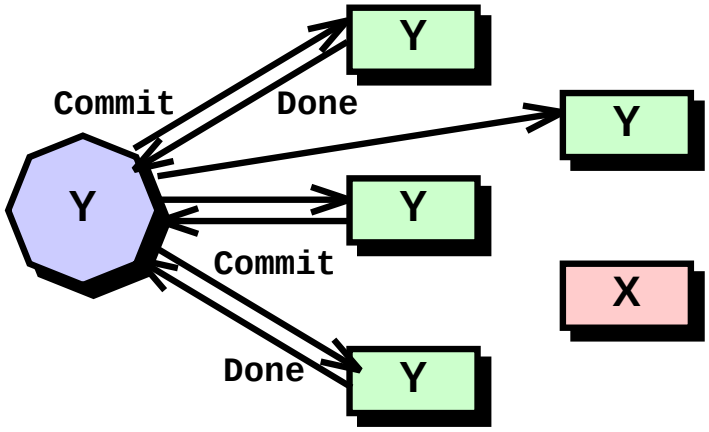

Fig. 6.3 : Phase 2 − Validation de l'écriture

On constate que la copie 4 a été mise à jour, mais qu'elle n'a pas répondu à l'initiateur. Cependant puisque celui−ci a reçu son quorum d'acquittements, il ne s'en préoccupe pas. Il peut alors indiquer au client que la modification a bien été effectuée.

Il existe deux variantes permettant d'améliorer les performances du protocole :

- L'initiateur se contente, en premier lieu, de diffuser les messages aux (n+1)/2 premiers sites (avec n = nombre de sites). En cas de panne, il diffuse la demande à un nombre de sites égal au nombre de votes manquants. Cela permet de diminuer le trafic réseau lorsqu'il n'y a pas de panne en diffusant un nombre minimal de messages(3).
- Si le trafic réseau est important, la phase de validation du protocole peut être intégrée aux messages suivants.

Les avantages du quorum sont :

+ Un protocole simple à mettre en œuvre et qui s'ajuste dynamiquement aux pannes de machines ; le traitement de la panne d'une machine fait implicitement partie de la mécanique associée au vote.
+ La cohérence stricte des données est garantie même en cas de partitionnement du réseau ; seule la partition qui contient la majorité des sites peut modifier les données.

Les principaux inconvénients sont :

– La taille du quorum est importante (pour n sites, il faut une majorité de (n+1)/2 sites) ce qui réduit rapidement la disponibilité en cas de panne.
– Un nombre important de copies si l'on veut fournir une bonne disponibilité.
– La recherche de la cohérence stricte pénalise les performances ; on utilise un coûteux protocole à deux phases pour valider toutes les copies de manière atomique.

Afin d'améliorer les performances, on cherche à réduire la taille du quorum tout en conservant la cohérence stricte. Le protocole en grille ("Grid protocol") de Cheung, Ammar et Ahamad [Cheung 92] et le protocole du treillis ("Triangular Lattice protocol") de Wu et Belford [Wu 92] sont des exemples de propositions où l'on tire partie de la structure logique induite par le réseau pour réduire la taille du quorum. On trouve dans [Rabinovitch 92] une excellente présentation des motivations et des solutions apportées pour mettre en œuvre de tels protocoles.

Pour illustrer le fonctionnement de cette classe de protocoles, nous allons présenter brièvement le protocole en grille.

Le protocole est fondé sur l'organisation logique des sites d'un réseau en grille. Un quorum en lecture sera composé d'un site de chaque colonne, alors qu'un quorum en écriture sera composé de tous les sites d'une colonne et d'un site pour

(3) Cette variante n'est intéressante que si le taux de panne reste faible. En effet, le temps d'opération augmente en fonction du nombre de pannes.

chacune des autres colonnes. Cette définition des quorums garantit que tout quorum en lecture à au moins une intersection avec un quorum en écriture.

Dans la figure Fig. 6.4, la taille du quorum en lecture, pour une grille composée de 3 lignes de 4 colonnes, est ramené de 7 (pour le vote majoritaire) à 4 sites, alors que celle du quorum en écriture est ramené de 7 à 6. Les tailles respectives des quorums sont fonction de la configuration de la grille : moins il y a de colonnes, plus le quorum en lecture est petit et le quorum en écriture grand.

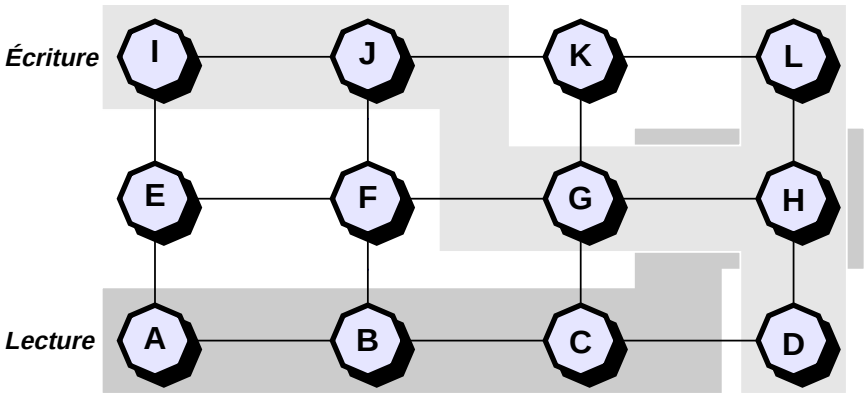

Fig. 6.4 : Un réseau organisé en une grille 3x4

> Les sites {A, B, C, H} définissent un quorum en lecture. Ce quorum a une intersection commune {H} avec le quorum en écriture {I, J, G, H, L, D}.

En cas de panne d'un ou plusieurs sites, le protocole procède par étapes successives (ligne par ligne, puis colonne par colonne) pour définir un quorum. Ainsi dans le cas de la figure Fig. 6.5, si les sites B, F, G, K et L sont en panne, {A, J, C, D} définit un quorum en lecture alors que {A, J, C, D, E, I} définit un quorum en écriture.

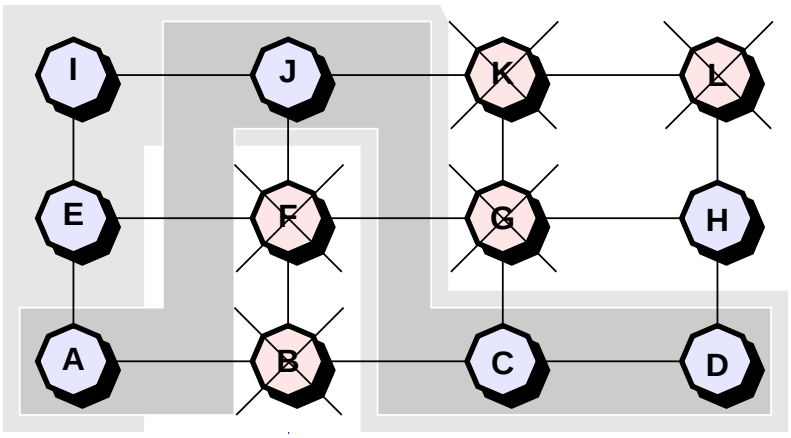

Fig. 6.5 : Réseau avec pannes

La constitution des quorums est progressive. Le protocole essaie de verrouiller tous les sites d'une ligne (resp. colonne). En cas de panne d'un site (détectée par un délai de garde), le protocole contacte le site de la ligne (resp. colonne) suivante dans la même colonne (resp. ligne).

L'avantage de cette technique est qu'elle permet (en changeant de ligne de départ) de répartir la charge entre les différents sites. On retrouve d'ailleurs un fonctionnement similaire dans le protocole du treillis.

Son inconvénient majeur est le coût de la constitution progressive du quorum en cas de panne d'une machine : il faut attendre la fin du délai de garde pour détecter la panne et contacter un nouveau site. Dans la figure Fig. 6.5, le protocole a dû attendre la détection de la panne du site B, puis celle du site F avant de pouvoir contacter le site J.

Remarques

Un effet de bord de la réduction de la taille du quorum est qu'elle réduit la disponibilité des données. En effet, puisque les écritures sont effectuées sur un plus petit nombre de sites (relativement au classique vote majoritaire), la disponibilité des données est réduite à celle des sites à jour.

De plus, comme le proposent Rabinovitch et Lazowska [Rabinovitch 92] ainsi que Pâris et Sloope [Pâris 92], il est nécessaire de fournir des mécanismes permettant la reconfiguration de la grille en cas de panne. En effet, si la configuration est statique, la panne de tous les sites d'une colonne, par exemple, bloque le fonctionnement de tous les sites.

Cette classe de protocoles a quand même l'énorme avantage de concilier performance et cohérence en l'absence de pannes, ce qui est bien sûr le cas le plus favorable mais aussi le plus fréquent. De plus, il est difficile de faire admettre aux utilisateurs que la tolérance aux pannes entraîne un surcoût important lorsqu'il n'y a pas de panne. En privilégiant le cas sans panne, on a de bonnes chances de satisfaire les utilisateurs.

VI.3.2 Algorithmes optimistes

Les algorithmes optimistes privilégient la disponibilité des objets par rapport à leur cohérence. Ils sont basés sur l'hypothèse que le partitionnement du réseau est une chose rare et considèrent qu'il n'y a pas lieu de mettre en œuvre un protocole trop strict, qui de plus diminue fortement les performances d'accès et la disponibilité des objets. Certains algorithmes essayent de profiter de la sémantique des objets manipulés afin de garantir une cohérence plus souple que la cohérence stricte.

L'algorithme que nous avons choisi pour illustrer la mise en œuvre d'un algorithme optimiste est celui de l'anti−entropie estampillée ("Timestamped anti−entropy") dû à Golding et Long [Golding 92].

L'algorithme optimiste utilisé par le système Coda est présenté dans la sous−section VI.4.1.

VI.3.2.1 L'anti−entropie estampillée

Le travail mené par Golding et Long se place dans le cadre de la construction d'une architecture supportant des services répartis avec comme objectif principal le support de services fiables à grande échelle (celle de l'environnement Internet).

La duplication des données est utilisée pour atteindre l'objectif de disponibilité fixé et permettre la croissance du système. La duplication doit être dynamique dans le sens où des serveurs peuvent être ajoutés ou supprimés. Le système doit être asynchrone (il ne doit jamais entraîner l'exécution d'un protocole de synchronisation des copies impliquant un grand nombre de machines) et les serveurs doivent être aussi indépendants que possible.

Les protocoles à incohérence lâche ou bornée ne font justement pas de mises à jour synchrones. Une mise à jour est effectuée sur un site, puis elle est propagée de manière différée aux autres sites. La valeur retournée par un serveur à un client dépend de ce que le serveur aura observé (ou reçu) au moment de la demande.

Les avantages liés à la propagation différée des mises à jour sont :

- Les clients n'ont pas à attendre que les modifications atteignent des sites distants. L'éventuelle panne du client pendant la propagation ne gêne en rien le fonctionnement du protocole.
- Les mises à jour peuvent être regroupées dans un seul message ce qui permet d'utiliser toute la puissance des réseaux à haut débit.
- Les transferts peuvent avoir lieu en dehors des heures de forte occupation du réseau.

Cependant, les applications doivent être capables de tolérer les incohérences potentielles des données. En effet, puisque les mises à jour ne sont pas propagées de manière synchrone à l'ensemble des copies, deux applications peuvent modifier de manière concurrente une même donnée.

Golding et Long modélisent leur système sous la forme d'un groupe de serveurs de copies qui communiquent au moyen d'un protocole de communication de groupe afin de gérer la duplication d'une donnée. Le protocole de communication fournit un service de diffusion d'un message depuis un serveur vers tous les autre serveurs du groupe. Le protocole détermine la cohérence des copies en contrôlant l'ordre dans lequel les messages sont envoyés entre les serveurs.

Le protocole de l'anti−entropie estampillée est un protocole de communication de groupe qui offre une garantie de fiabilité (le message sera reçu par tous les serveurs du groupe) en un temps fini mais non borné[(4)].

Chaque serveur maintient (sur disque) trois structures de données : un journal des messages, et deux vecteurs d'estampilles. Le journal des messages contient tous les messages ayant été reçus par le serveur. Les messages sont marqués de l'identité de leur émetteur et d'une estampille (mise à jour à chaque envoi de message). Les messages sont effacés du journal lorsque tous les autres serveurs les ont reçus. Le premier vecteur d'estampille mémorise la dernière estampille de mise à jour reçue par le serveur en provenance de chacun des autres serveurs. Le deuxième vecteur sert à mémoriser les acquittements de réception de message reçus des autres sites. Un serveur utilise ce vecteur pour détecter la réception par tous les autres serveurs d'un message qu'il a émis.

A intervalles réguliers, un serveur A sélectionne un partenaire B et démarre une session d'anti−entropie. Les deux serveurs s'échangent leurs vecteurs d'estampilles, et détectent, en comparant les vecteurs, la non réception éventuelle de certains messages. Les messages manquants sont alors récupérés dans le journal de celui des deux serveurs qui est à jour et sont envoyés à l'autre serveur.

De session en session, les messages vont ainsi être propagés à l'ensemble des serveurs. La figure Fig. 6.6 présente un exemple de diffusion de messages entre 4 serveurs.

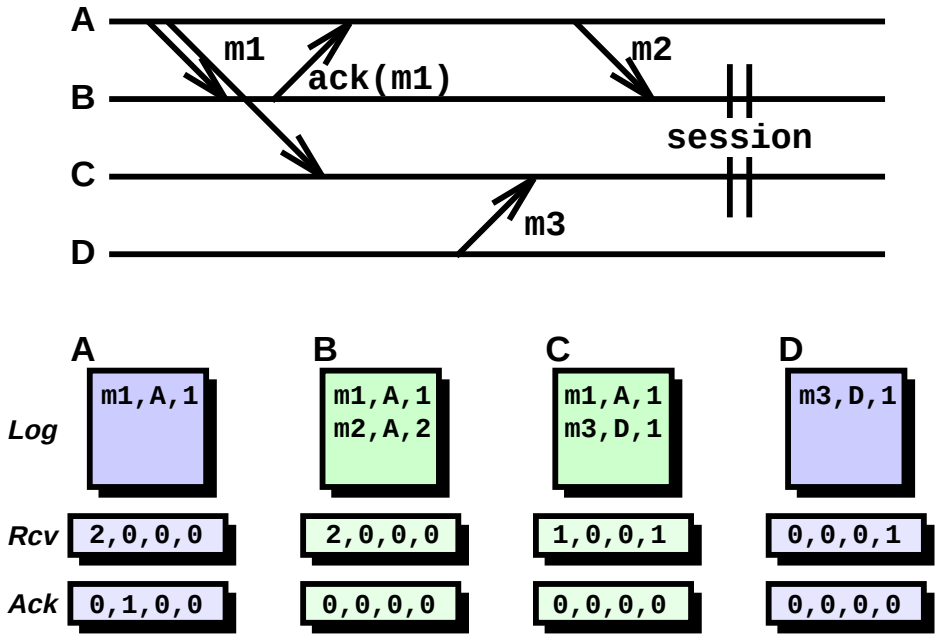

Fig. 6.6 : Diffusion de message par anti−entropie

(4) Les horloges des différents sites doivent être faiblement synchronisées.

> Le serveur A a émis 2 messages : m1 et m2. m1 a été reçu par les serveurs B et C. B a acquitté la réception de m1. Le message m2 a été reçu uniquement par B. Pour sa part le serveur D a envoyé le message m3 au seul serveur C.
>
> Le déclenchement d'une session entre B et C se traduit par la comparaison des deux vecteurs Rcv(B) = {2,0,0,0} et Rcv(C) = {1,0,0,1} par les deux serveurs. B constate que C n'a pas reçu m2 alors que C constate que B n'a pas reçu m3. Chaque serveur utilise alors son journal local pour transmettre le message manquant.

De nombreuses variations sont possibles : le protocole peut garantir la diffusion des messages selon un ordre total ou causal en retardant la diffusion des messages trop jeunes lors de la comparaison des vecteurs d'estampilles. On se reportera à la description et à l'évaluation théorique du protocole donnée dans [Golding 92].

VI.4 Application au stockage fiable

Nous allons maintenant étudier la mise en œuvre de la duplication dans les systèmes de fichiers distribués Coda [Satyanarayanan 90] et Echo [Birrell 93]. Nous avons choisi ces deux systèmes car ils ont l'énorme avantage d'être (ou d'avoir été) utilisés réellement et quotidiennement comme support de stockage par un nombre important d'utilisateurs : une trentaine pour Coda et une cinquantaine pour Echo.

L'expérience acquise par les concepteurs de ces deux systèmes a donc valeur d'exemple car elle est représentative d'une utilisation en vraie grandeur.

Par ailleurs, ces deux systèmes illustrent la mise en œuvre des deux principales classes d'algorithmes : Coda emploie une approche optimiste alors que Echo est fondé sur un algorithme pessimiste.

VI.4.1 Coda

Coda est un système de fichiers distribués, descendant du système de fichiers Andrew (AFS), développé à l'Université Carnegie−Mellon par l'équipe de Mahadev Satyanarayanan [Satyanarayanan 90]. Il offre un accès continu aux données en cas de panne de serveur ou du réseau. Un des principaux traits du système est l'attention portée au support du travail en mode déconnecté (qualifié aussi de travail nomade) [Kistler 92][Satyanarayanan 93].

Coda a été conçu pour un environnement Unix constitué d'un ensemble de clients non sûrs et d'un petit nombre de serveurs de stockage sûrs. La conception a été optimisée pour supporter les types d'accès et de partage rencontrés dans le milieu académique et celui de la recherche. Le système ne supporte pas les applications telles que les applications transactionnelles qui modifient de manière concurrente un grand nombre de données de granularité fine.

Coda utilise un algorithme optimiste qui est une variante de l'algorithme des Copies Disponibles et autorise les lectures et les écritures dans toutes les partitions. Ce choix a été fait pour trois raisons :

- cette approche permet d'offrir une grande disponibilité,
- les utilisateurs doivent pouvoir travailler à partir de portables connectés au système ou isolés, une approche pessimiste interdit le travail en mode isolé,
- les concepteurs estiment que le partage en écriture entre des utilisateurs est relativement peu fréquent dans les environnements universitaires qui sont visés.

Un mécanisme de détection des conflits est mis en œuvre par le système. Un mécanisme de réparation des conflits activé de manière semi−automatique est fourni. Il tient compte de la sémantique attachée à certains objets tel que les répertoires du système de fichier. En dernier recours, l'utilisateur est prévenu. Il peut alors corriger manuellement le conflit à l'aide d'un outil de type éditeur de fichier, fourni par le système.

Coda augmente la disponibilité des données par la duplication des fichiers sur un ensemble de serveurs et par l'utilisation d'un mécanisme de cache des fichiers sur les machines des clients. Ce mécanisme permet aux clients de travailler entièrement en local sans avoir à contacter les serveurs pour peu que l'ensemble des fichiers auxquels ils accèdent soient dans le cache. L'accès aux fichiers est réalisé de façon complètement transparente à l'utilisateur. L'utilisateur accède en effet à ses fichiers à travers une interface du type *open, read, write, close.* Un processus système, appelé Venus, est chargé de transmettre la requête utilisateur à l'ensemble des serveurs.

L'unité de duplication choisie est le volume, c'est à dire un ensemble de fichiers. L'ensemble des serveurs contenant une copie d'un volume constitue le groupe de stockage du volume (VSG). L'ensemble des serveurs d'un VSG accessible à un instant donné par Venus est appelé le VSG accessible (AVSG). Différents clients (processus Venus) peuvent avoir différents AVSG pour le même volume. Venus teste de façon périodique les membres des VSG dont il a des données dans son cache. Ce test est relativement peu fréquent dans la mesure où il a lieu toutes les dix minutes.

Nous allons étudier le fonctionnement du protocole optimiste de Coda au travers d'un exemple relativement simple. La variante de l'algorithme utilisée par Coda peut être qualifiée de variante "lit−une−donnée, lit−tous−les−états, écrit−toutes". Pour lire une donnée qui ne se trouve pas dans le cache, Venus s'adresse à un serveur

privilégié (PS) de l'AVSG afin qu'il lui retourne cette donnée ainsi qu'une information sur l'état de cette donnée. Ce PS peut être choisi en fonction du temps de réponse qu'il fournit. Néanmoins, Venus contacte aussi en parallèle les autres serveurs afin de déterminer si le PS lui a retourné une donnée à jour ou non. La figure suivante schématise le déroulement de ce protocole dans le cas d'une faute de cache.

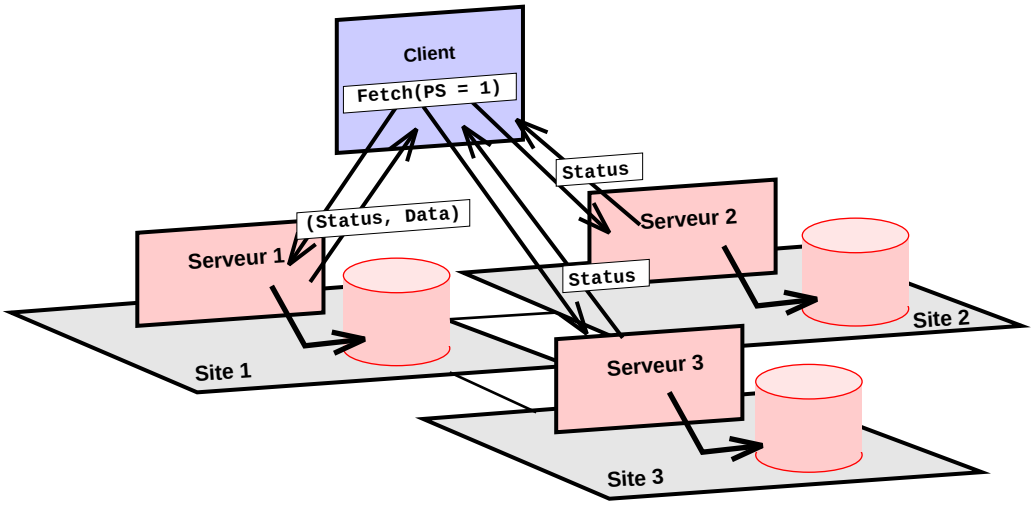

Fig. 6.7 : Déroulement d'une faute de cache

Venus compare les états reçus pour tester si les copies sont toutes équivalentes. Si un conflit est détecté, l'appel système qui a provoqué la faute de cache est abandonné et la procédure de correction est appelée. En revanche, si aucun conflit n'est décelé, mais qu'une des copies n'est pas à jour, Venus informe de manière asynchrone l'AVSG qu'une remise à jour de certaines copies est nécessaire. Le contrôle des conflits écriture−écriture est réalisé grâce à des vecteurs de versions (CVV) mis à jour et testés lors des fermetures de fichiers ou régulièrement par Venus.

Lorsqu'un fichier est fermé après modification, Venus le transfère en parallèle à tous les membres de l'AVSG. Cette approche a été préférée à l'approche qui consiste à transférer le fichier au PS, puis à faire réaliser la propagation de la nouvelle copie par le PS en tâche de fond. Elle permet de ne pas utiliser, pour la propagation, le temps CPU des serveurs qui constituent le goulot d'étranglement du système. Cette technique doit permettre la construction d'un système de grande taille (c'est à dire, plusieurs centaines de sites).

De plus, un serveur n'exécute aucune opération de remise à jour après une panne, ce sont les clients qui l'informeront que ses données ne sont plus à jour ou qu'elles sont en conflit avec d'autres. Bien que cette stratégie ne viole pas les règles de cohérence garanties par le système, elle augmente les chances d'un futur conflit. C'est pourquoi, les concepteurs du système envisagent de remettre automatiquement à jour un serveur par collaboration avec les autres.

Les performances observées dans Coda montrent que le surcoût de la duplication des fichiers est acceptable. Cela est principalement dû à l'utilisation d'un protocole optimiste qui s'adapte très bien à la sémantique des objets manipulés, les fichiers, ainsi qu'à un degré de partage relativement faible. Un tel système n'est absolument pas apte à supporter une base de données dans laquelle les objets sont petits, partagés et souvent modifiés.

VI.4.2 Echo

Echo est un système de fichiers distribués développé au centre de recherche DEC SRC [Birrell 93]. Il fut conçu dans l'intention d'être aussi performant qu'un système de fichiers local, d'être capable de supporter un grand nombre de clients par serveur et surtout d'offrir une fiabilité élevée par rapport aux systèmes de fichiers distribués classiques (NFS, AFS) et meilleure que celle d'un système centralisé.

Ses caractéristiques principales sont :

- Une désignation globale des fichiers ; un utilisateur ou un programme utilisera le même nom pour désigner un fichier quelle que soit sa localisation au sein du système.
- Un accès global ; les fichiers et les répertoires sont accessibles et manipulés depuis l'ensemble des sites en accord avec la politique de gestion locale des caches.
- Un contrôle d'accès garantissant le respect des règles de sécurité communément admises dans les systèmes à temps partagé.
- Une tolérance aux pannes de sites et aux partitions réseaux en accord avec la gestion de caches de fichiers utilisés pour les performances.

Echo est organisé selon l'architecture client/serveur que l'on retrouve dans la plupart des systèmes de fichiers distribués. Sur chaque station de travail cliente est placé un client Echo appelé "clerk". L'accès à un fichier Echo est redirigé de manière transparente aux applications vers le clerk. Le clerk est alors responsable de la connexion aux différents serveurs de stockage Echo. C'est au niveau du clerk que sont pris en compte les aspects liés à la désignation et à l'accès global des fichiers. La figure Fig. 6.8 schématise l'architecture du système Echo.

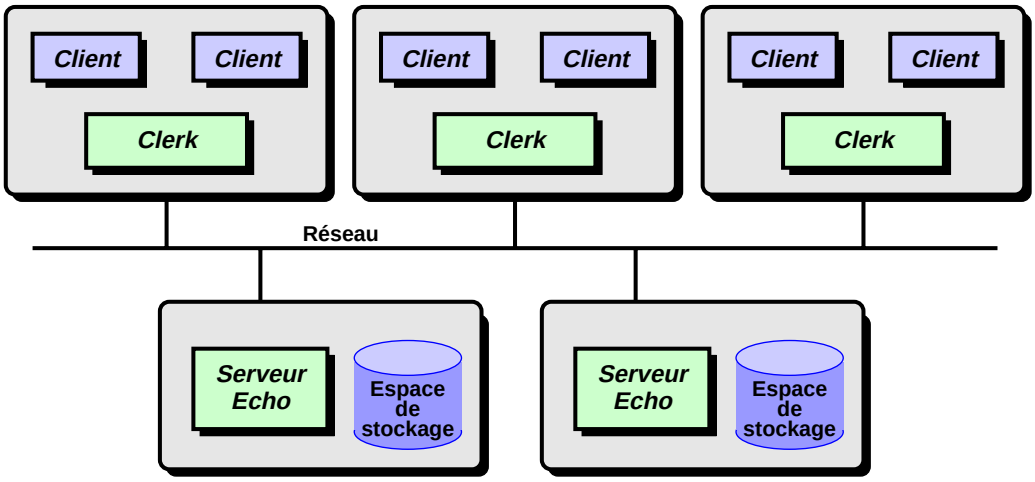

Fig. 6.8 : L'architecture client/serveur du système Echo

Afin d'offrir les meilleures performances possibles tout en supportant un grand nombre de clients, Echo fait un usage intensif des possibilités de cache des fichiers par les clerks sur les stations clientes.

VI.4.2.1 La gestion des caches dans Echo

A la différence de Coda, Echo met en œuvre une gestion cohérente des caches qui permet de supporter des applications partageant des données. Les caches sont gérés selon la politique suivante : lorsqu'une application située sur une station désire accéder à un fichier dont il existe des pages modifiées dans le cache d'une autre station, ces pages sont tout d'abord stockées par le serveur avant que l'application ne puisse y accéder.

Cette gestion de la cohérence des caches utilise un jeton associé à chaque fichier. Un clerk peut posséder une copie d'un fichier lorsqu'il possède un jeton en lecture. Pour être capable de la modifier, il doit posséder un jeton en écriture. Lorsqu'un clerk possède un jeton en écriture, aucun autre clerk ne peut posséder de jeton (en lecture ou en écriture) sur le fichier. Chaque serveur de stockage gère la révocation des jetons, l'invalidation des caches et le transfert des pages en fonction des demandes qu'il reçoit sur les fichiers dont il a la charge.

Ce fonctionnement est très efficace en environnement industriel (sous Unix) où le partage intensif de fichiers n'est pas fréquent et où le mode d'accès courant est la lecture ou l'écriture intégrale d'un fichier. Par contre, les performances se dégradent notablement lorsqu'un fichier est mis à jour de manière concurrente sur plusieurs stations.

En cas de panne d'une station cliente (ou de partitionnement du réseau) la rendant inaccessible, le serveur ne peut plus invalider les jetons que possède le clerk de cette station.

Pour remédier à ce problème, les jetons ne sont valides que pour une courte période de quelques secondes. Le clerk peut conserver un jeton plus longtemps en envoyant à intervalle régulier une demande de prolongation du jeton.

En cas de partition, la période expirant au bout de quelques secondes(5), le serveur et le clerk révoquent les jetons du clerk. Cependant afin de résister aux pannes de réseau temporaires, l'invalidation n'est pas effective (du coté du serveur) tant qu'il n'y a pas de conflit avec un autre clerk.

La gestion proprement dite des caches utilise la politique des ***écritures masquées*** : lorsqu'une application crée, écrit ou détruit un fichier, ou modifie un répertoire, l'appel système retourne dans le client avant que l'opération ait été transmise au serveur ou validée en espace de conservation. Cette politique est similaire à celle employée dans Sprite, NFS ou AFS, mais les concepteurs d'Echo ont eux cherché à l'utiliser de manière systématique. Une application peut ainsi créer un fichier, l'écrire, le re−lire et le détruire sans même que le serveur ne s'en rende compte.

Afin qu'une telle politique soit utilisable par les programmeurs, le système doit offrir des garanties sur les opérations qui sont effectivement envoyées au serveur et sur leur ordre. Car en cas de panne, les applications doivent être capable de prendre en compte le fait que certaines opérations n'ont pas encore été effectivement reportées.

Echo utilise la sémantique des objets qu'il manipule, les fichiers et les répertoires, pour mettre en œuvre une politique de propagation efficace, automatique et cohérente. Il fournit aussi deux primitives qui permettent de contrôler cette recopie depuis les applications. L'appel système `fsync` bloque l'application jusqu'à ce que toutes les modifications sur un fichier aient été effectuées. L'appel système `forder`, qui n'est pas bloquant, permet de spécifier un ordre partiel entre les opérations d'écriture (au niveau des blocs du disque). Les applications ont ainsi la garantie de retrouver des états de fichier qui respectent les invariants qu'elles spécifient par `forder`.

Cette gestion des caches est très efficace mais elle a deux inconvénients : la liste des mises à jour à reporter peut devenir très longue et les applications peuvent perdre un grand nombre de données en cas de partition ou de panne du serveur.

---

(5) Les horloges des clerks et des serveurs sont supposées ne pas différer de plus d'une certaine constante.

Le premier point est traité en ajoutant un processus chien de garde au clerk qui est chargé de forcer la copie à des intervalles réguliers ou lorsque des pages modifiées sont depuis trop longtemps dans le cache.

Le deuxième point est plus délicat à traiter. Lorsqu'un clerk se rend compte qu'il y a eu panne (en détectant la fin de période de validité des jetons), il est dans une situation où il a accepté un ensemble d'opérations de modifications mais où il ne peut pas les valider en espace de conservation. La solution employée dans Echo n'est que partiellement satisfaisante aux dires même de ses concepteurs. Elle consiste à prévenir les applications qu'une panne s'est produite et à laisser celles−ci se débrouiller (en recopiant éventuellement les données dans des fichiers locaux). Dans la pratique, les applications ne savent pas bien gérer ce cas de figure.

A la différence de Coda, une machine isolée se verra donc interdire toute modification sur les fichiers stockés sur les serveurs.

VI.4.2.2 La tolérance aux pannes dans Echo

L'architecture d'Echo est particulièrement souple quant à ses aspects tolérance aux pannes : l'espace de conservation peut être configuré pour offrir plus ou moins de fiabilité (en dupliquant les disques de stockage) et plus ou moins de disponibilité (en dupliquant les serveurs).

Le problème est de réaliser des mécanismes de tolérance aux pannes qui ne brisent pas la garantie de cohérence stricte du système : une mise à jour réalisée par un client est immédiatement visible par les autres clients, de plus, une fois validée, la mise à jour n'est jamais abandonnée (comme cela peut−être le cas dans Coda).

Une configuration Echo classique se compose de deux serveurs ayant chacun un disque de stockage. L'hypothèse de base étant que deux disques fournissent suffisamment de fiabilité dans la pratique pour résister à un grand nombre de pannes . Echo supporte aussi des configurations dans lesquelles un serveur peut accéder à plusieurs disques de type miroir ("dual−ported").

Echo utilise l'algorithme du site primaire pour gérer la duplication des données. Les clerks contactent uniquement le serveur primaire qui se charge de propager les mises à jour aux serveurs secondaires. En cas de panne du serveur primaire, un nouveau serveur est élu. L'élection se déroule comme suit :

- Un vote est attribuée à chaque disque.
- Chaque serveur essaie de collecter le plus grand nombre de votes (un serveur qui a accès à des disques miroirs pourra ainsi récolter deux votes).
- Le serveur qui possède une majorité des votes est élu serveur primaire.

Dans une configuration classique, un processus témoin situé sur une troisième machine est utilisé pour faire pencher la balance vers l'un ou l'autre des serveurs. De

plus, en cas de partitionnement, seul le serveur qui pourra communiquer avec le témoin possédera une majorité des votes.

L'élection tient aussi compte du fait qu'un des serveurs peut posséder une version à jour des jetons et des données. On se reportera à [Mann 89] pour une description complète du mécanisme d'élection utilisé dans Echo.

Les opérations de stockage sont mises en œuvre à l'aide d'une technique de journalisation. Lorsqu'un clerk envoie une opération de mise à jour, le serveur contrôle la validité de la demande, met à jour les informations qu'il possède sur les jetons, puis il écrit la modification dans un enregistrement du journal (sur disque). L'enregistrement est alors envoyé à tous les serveurs secondaires. Le serveur signale alors au clerk que l'opération a été validée. De manière asynchrone, les serveurs propagent alors les modifications enregistrées dans le journal sur les disques de conservation.

L'intérêt de cette technique est multiple :

- Le débit du serveur est augmenté (vu des clerks) car le journal est écrit de manière séquentielle. La recopie n'affecte que partiellement les clerks puisqu'elle est réalisée de manière asynchrone.
- La disponibilité du serveur est augmentée dans la mesure où le redémarrage depuis un journal est plus rapide ; on rejoue uniquement les opérations qui n'ont pas été reportées.
- Les performances du serveur peuvent être augmentées en regroupant plusieurs opérations dans un seul enregistrement du journal.
- L'intégrité des données peut être facilement préservée ; un seul enregistrement peut décrire une modification qui affecte différentes parties de l'espace de conservation (par exemple, lors du déplacement d'un fichier).

Echo utilise cette technique pour optimiser et simplifier son algorithme de duplication sur deux points particuliers :

- Dans une configuration où tous les serveurs ont accès à tous les disques, le site primaire met à jour tous les disques et il se contente d'envoyer de manière asynchrone les enregistrements du journal au secondaire. Le secondaire applique ces enregistrements en mémoire centrale. En cas de panne, il n'aura besoin de récupérer dans le journal (stocké sur disque) que les quelques enregistrements qui ne lui auront pas été transmis.
- L'algorithme de restauration des serveurs et des disques utilise les informations stockées dans les journaux pour décider quels sont les enregistrements à propager à toutes les copies et quels sont ceux qu'il faut supprimer car ils n'ont pu être validés à temps.

VI.4.3 Comparaison

Les deux systèmes de fichiers distribués que nous avons présentés dans les sous−sections précédentes ont le même objectif final : fournir un accès global à une base de fichiers répartie sur un ensemble de sites.

Ils font d'ailleurs tout les deux un usage intensif des possibilités offertes par les caches des fichiers sur les stations clientes afin de bénéficier de bonnes performances. Par ailleurs, ni l'un ni l'autre ne font confiance, pour des raisons de sécurité, aux stations clientes.

Cependant, leurs environnements respectifs étant différents, académique avec très peu de partage pour Coda, industriel avec développement de programmes en équipe pour Echo, leurs orientations sont diamétralement opposées :

- Coda privilégie au maximum la disponibilité des données en autorisant le travail en mode déconnecté sans garantie de cohérence.
- Echo assure que la cohérence sera préservée en toutes circonstances au prix d'une réduction de la disponibilité.

L'expérience acquise par l'utilisation quotidienne de chacun des systèmes par un nombre significatif d'utilisateurs a d'ailleurs montré que les fonctions et la qualité de service offertes repondaient parfaitement aux besoins réels des utilisateurs.

On retrouve les différences d'objectifs dans la réalisation :

- Coda utilise un algorithme optimiste avec une phase de réconciliation en cas de conflit d'écriture.
- Echo utilise un algorithme pessimiste pour garantir la cohérence des données face à des partitionnements du réseau.

Les deux systèmes utilisent des caches de fichiers sur les stations clientes, mais Coda profite de cette gestion pour supporter le travail en mode déconnecté. Un utilisateur peut continuer à travailler et à modifier ses données même après avoir été isolé par une panne du réseau ou par une déconnexion volontaire du réseau. Les modifications seront automatiquement reportées lorsque la connexion sera rétablie.

Dans la même situation et pour préserver la cohérence, Echo invalide toutes les données (et les éventuelles modifications) présentes dans le cache de la station locale.

Pour résumer, les points forts et faibles de Coda sont :

+ Support du travail en mode déconnecté.
+ Très bonnes performances.

– Pas de cohérence même en l'absence de panne.
– Phase de restauration parfois délicate à gérer pour l'utilisateur.
– Pas de support pour le partage intensif de fichier.

Les points forts et faibles de Echo sont :

+ Excellentes performances, les mêmes qu'un système de fichier local.
+ Conservation de la cohérence permettant le partage de fichiers.
– Perte de toutes les modifications non validées en cas de panne du réseau.

VI.5 Conclusion

Nous avons vu qu'il existe deux grandes classes d'algorithmes de maintien de la cohérence de données dupliquées : les algorithmes pessimistes et les algorithmes optimistes. Les algorithmes pessimistes assurent la cohérence stricte des copies au détriment des performances d'accès, alors que les algorithmes optimistes acceptent de laisser les copies diverger afin de ne pas trop ralentir les accès. Un protocole optimiste a besoin d'outils de détection et de réparation pour fonctionner correctement.

Le choix d'un type d'algorithme par rapport à l'autre dépend de la qualité de service que le système cherche à fournir à ses utilisateurs. Ainsi le système Coda qui privilégie la disponibilité dans un environnement académique où le partage de fichiers est réduit et donc la probabilité d'accès concurrents faible, utilise un algorithme optimiste. Le système Echo vise lui à supporter le travail d'équipes de développement dans un milieu industriel : la cohérence des données est alors essentielle ; Echo utilise donc un algorithme de type pessimiste.

On peut aussi remarquer que les deux systèmes utilisent des algorithmes de gestion de la duplication qui sont relativement simples. La raison principale est que le taux de duplication des données est faible (de l'ordre de deux à trois copies). Ce faible taux est suffisant pour garantir la conservation fiable des données ainsi que dans le cas de Coda leur disponibilité.

Dans le chapitre suivant, nous décrivons la conception du serveur de stockage fiable Goofy. La caractéristique principale de ce serveur est qu'il tire partie de l'architecture de la machine à grappes afin de concilier au mieux cohérence, performance et disponibilité, trois caractéristiques généralement jugées incompatibles. Le compromis que nous avons défini nous permet de fournir un service qui ne privilégie pas l'un de ces aspects au dépend des deux autres.

Chapitre VII

Pour un stockage fiable dans le système Guide−2

Lors des chapitres III et IV, nous nous sommes attaché à présenter les aspects du système réparti Guide ayant trait au support de la persistance. Plusieurs applications complexes telles qu'un éditeur coopératif de documents, un gestionnaire de circulation intelligente de formulaires ainsi qu'un tableur multi−utilisateurs exploitent la gestion évoluée de la persistance et du partage des données fournie par notre système. Or le système ne fournit actuellement pas de mécanisme de tolérance aux pannes. Un tel service est cependant vital pour ces applications car elles accèdent à un grand nombre d'objets disséminés sur plusieurs machines et sont donc particulièrement sensibles aux pannes de l'environnement d'exécution.

Cette constatation nous a amené à lancer l'étude, décrite dans le chapitre VI, sur les aspects de la tolérance aux pannes qui sont liées au stockage des données. Encore une fois, il n'était pas question de bâtir un système distribué tolérant les pannes, mais plutôt de définir les mécanismes de base qui nous permettent d'augmenter la fiabilité des données et d'améliorer ainsi la qualité de service fournie par le système en terme de support de la persistance. Cette première étape, rendre les données fiables et disponibles, nous semble le point de départ obligé à tout travail sur la tolérance aux pannes du système et des applications.

Cette étude s'est déroulée en deux étapes. Une première expérimentation a été conduite sur la plate−forme Guide−1/Mach 2.5 [Boyer 91a] durant l'année 1991. Les résultats de cette expérimentation, qui sont décrits dans [Chevalier 92b], nous ont permis de valider certaines hypothèses et de nous construire une base de travail : le serveur de stockage fiable Goofy.

Parallèlement à cette évaluation, la conception et la réalisation d'une nouvelle version du système Guide, appelée Guide−2, étaient en cours au dessus du micro−noyau Mach 3.0. Cette nouvelle architecture du système, nous a conduit à repenser la conception du serveur de stockage Goofy [Chevalier 93b] afin de l'adapter à la nouvelle architecture du système.

Le présent chapitre est donc structuré comme suit : la section VII.1 présente l'architecture, les fonctions ainsi qu'une évaluation de la première version du serveur de stockage fiable Goofy. Le changement d'architecture entre Guide−1 et Guide−2 avec les impacts que cela a entraîné sur le support de la persistance font l'objet de la section VII.2. La section VII.3 définit la nouvelle version du serveur de stockage Goofy et en donne une évaluation préliminaire. Nous concluons ce chapitre dans la section VII.4.

VII.1 Un prototype pour le système Guide−1

L'objectif principal de cette expérimentation était de récolter des informations sur l'impact et la faisabilité d'un serveur de stockage fiable dans un système réparti tel que Guide. Pour ce faire, nous nous sommes basé sur la plate−forme du système Guide−1 alors disponible sur le micro−noyau Mach 2.5(1).

VII.1.1 Architecture

La gestion de la mémoire est ce qui caractérise l'architecture du prototype. On retrouve une architecture similaire à celle de Guide−2 (cf section IV.1) basée principalement sur l'utilisation de gérants de mémoire (paginateurs externes) pour gérer l'accès, le partage et le stockage des données. La différence entre les deux architectures est en fait liée à la nature des données gérées : les gérants de mémoire du prototype gèrent des objets et non pas des grappes.

Le fait de devoir gérer des objets au niveau des gérants de mémoire ne résulte pas d'un choix de conception mais est la conséquence du portage du système Guide−1 sur Mach 2.5. Le système Guide−1 n'introduisait pas de séparation claire entre la gestion des objets au niveau langage et la gestion du partage et du stockage au niveau système. Lors de l'adaptation de ce système à l'architecture micro−noyau, nous n'avons pas voulu tout bouleverser en modifiant profondément la structure et les interfaces internes. Nous aurons l'occasion de revenir sur ce sujet dans la section VII.2.

L'interface des gérants de mémoire du prototype est donc différente de celle définie par la machine à grappes. Les gérants de mémoire mettent en œuvre les deux familles de fonctions suivantes :

- La création/destruction et le chargement/stockage des objets depuis l'espace de conservation dans la mémoire d'exécution.
- La liaison/déliaison des objets dans la mémoire virtuelle des applications ainsi que le traitement des fautes de pages (sur les objets) envoyées par les noyaux.

(1) Dans toute la suite de ce chapitre, cette plateforme du système Guide−1 sur le micro−noyau Mach 2.5. sera référencée sous le vocable : prototype Guide/Mach.

La première famille concerne la gestion de l'espace de conservation tandis que la seconde concerne la gestion de la mémoire d'exécution. Afin d'accroître la modularité du gérant, ces deux familles sont séparées au sein d'un gérant de mémoire à l'aide d'une interface bien identifiée.

L'utilisation et l'intégration du serveur de stockage fiable Goofy a été largement facilité par cette séparation des fonctions au sein des gérants de mémoire. En effet, le serveur Goofy a pour rôle d'augmenter la disponibilité et la fiabilité des données en espace de conservation. Il s'agit donc clairement d'améliorer la qualité du service de conservation fourni par les gérants de mémoire.

Le serveur Goofy est donc structuré comme un serveur distribué de stockage des objets qui gère un espace de conservation fiable (en conservant plusieurs copies des objets) et dont les clients sont les gérants de mémoire des différents sites Guide. L'interface entre Goofy et les gérants de mémoire est de type transfert explicite d'objets à l'aide des primitives `Create`, `Delete`, `Load` et `Store`. La figure Fig. 7.1 illustre l'architecture de la gestion de la mémoire dans le prototype Guide/Mach utilisant le serveur Goofy.

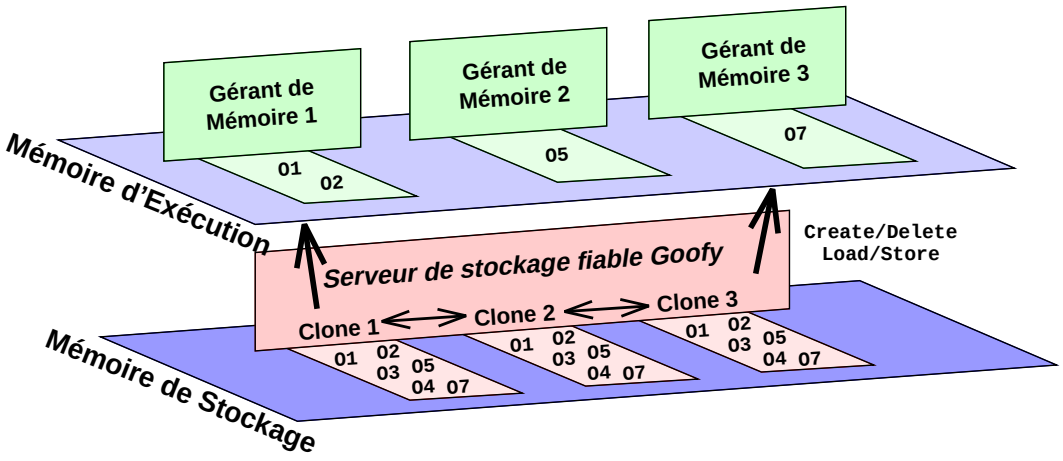

Fig. 7.1 : Architecture du serveur Goofy pour le prototype Guide/Mach

Le serveur réparti Goofy est composé d'un ensemble de ***clones***(2) (c'est à dire d'un ensemble de serveurs), qui coopèrent entre eux pour cacher aux gérants de mémoire le fait que les objets sont dupliqués. Cette coopération est essentielle pour

(2) Par opposition à des copies, les clones sont des entités actives qui agissent de la même manière.

maintenir la cohérence des différentes copies qui sont gérées par chacun des clones. Elle permet aussi de détecter et traiter les pannes de sites (et donc des clones) et leur réintégration.

Les clones doivent être capables de traiter des requêtes provenant soit des gérants de mémoire soit des autres clones. Ils sont composés de deux flots d'exécution : un pour traiter les requêtes des gérants et un autre pour traiter les requêtes internes. Cette réalisation a l'avantage de ne pas trop consommer de ressources système tout en étant relativement modulaire.

Il convient de noter que le serveur Goofy (en fait, ses clones) n'est pas intégré au prototype Guide/Mach mais est vu comme un serveur de stockage externe. Cela nous a permis de bâtir une architecture de système souple dans laquelle le changement d'un serveur par un autre se fait très facilement (cela est d'autant plus aisé que l'interface du service de stockage est des plus simples) et, où dans laquelle plusieurs serveurs de stockage ayant des qualités de services différentes peuvent coexister.

VII.1.2 Fonctions principales

On suppose que les stations sont du type "silencieuses sur défaillance" ("Fail−stop"), c'est à dire que soit une station est en panne, soit elle fonctionne correctement.

Le réseau Ethernet est quant à lui, sujet à deux types de pannes : la perte de messages et le partitionnement. Le serveur de stockage fiable devra donc faire face aux pannes de sites, aux pertes de messages, ainsi qu'aux partitionnements du réseau.

On peut remarquer que notre objectif principal est d'augmenter la disponibilité des objets en les dupliquant, tout en conservant la cohérence des différentes copies et ce malgré les partitionnements du réseau. Or si un tel objectif peut−être atteint dans un environnement sans partitions en utilisant l'algorithme optimiste des copies disponibles ("Optimistic Available Copy") décrit dans [Long 90], il est impossible par exemple de garantir la cohérence des copies en cas de partition sans réduire la disponibilité des données. Nous essayerons donc de décrire les compromis que nous avons du faire pour satisfaire au mieux ce double objectif.

VII.1.2.1 Cohérence

Afin de garantir que les applications partagent la même image (celle en mémoire d'exécution) d'un objet, nous aurions pu choisir l'une des deux solutions suivantes :

- Un seul gérant de mémoire a la charge de l'objet en mémoire d'exécution.
- Tous les gérants de mémoire peuvent gérer localement l'objet.

Dans les deux cas, les gérants doivent donner aux applications avec lesquelles ils interagissent une vision cohérente de l'objet. Cependant dans le deuxième cas, le service Goofy doit garantir que les gérants partagent une image cohérente de l'objet. Cela introduit donc une double gestion de la mémoire : entre Goofy et les gérants puis entre les gérants et les applications.

Pour éliminer cette double gestion de la cohérence, nous avons opté pour la première solution. Elle est mise en œuvre par un simple mécanisme de verrouillage des objets : tant qu'un objet est verrouillé par un gérant de mémoire, le service Goofy refuse aux autres gérants le droit de le charger en mémoire d'exécution.

L'utilisation du verrouillage garantit la cohérence de l'objet puisqu'il n'existe qu'une seule image de l'objet en mémoire d'exécution et que celle−ci est gérée par un seul gérant de mémoire.

Néanmoins, les gérants de mémoire sont aussi sensibles aux pannes de site ou aux partitions réseaux. Le mécanisme de verrouillage doit donc tolérer ou faire face à ces événements. La panne d'un gérant peut−être traitée de deux manières :

- Le système attend que le gérant redémarre. Cela revient à ignorer la panne, mais alors tous les objets verrouillés sont inaccessibles.
- La panne est considérée comme permanente (ou grave), en conséquence de quoi tous les verrous pris par le gérant en panne sont invalidés. Un autre gérant peut alors prendre le relais grâce aux copies des objets qui sont conservées par le service de stockage fiable.

Goofy utilise la technique de l'invalidation des verrous car elle a l'avantage de préserver la cohérence des objets tout en fournissant une meilleure disponibilité en cas de panne d'un des gérants de mémoire. Notons que cette solution est similaire à la gestion des jetons dans Echo (cf sous−section VI.4.2.1).

VII.1.2.2 Verrouillage des objets et invalidation

Le verrouillage d'un objet par un gérant de mémoire est basé sur l'utilisation d'un protocole de vote majoritaire entre les clones qui composent le service Goofy. La politique pessimiste ainsi mise en œuvre permet de garantir la cohérence des objets puisque seul un gérant de mémoire peut verrouiller, et donc charger en mémoire d'exécution, un objet à un instant donné.

La panne d'un clone ou la perte d'un message sont détectées par les clones à l'aide d'estampilles : un clone qui a été en panne associe une estampille obsolète à un objet donné ; à la réception d'un message concernant cet objet, le clone détecte automatiquement que sa copie n'est pas à jour. Si un clone diffuse un estampille obsolète, les autres clones le previennent aussitôt.

L'information de verrouillage ainsi que les estampilles des objets verrouillés sont stockées de manière fiable (par l'utilisation d'un fichier fantôme) par chacun des clones à chaque opération : verrouillage/déverrouillage.

Le choix d'une politique de verrouillage pessimiste peut sembler en contradiction avec notre objectif de disponibilité élevée. Les politiques pessimistes sont en effet connues comme limitant la disponibilité au profit de la cohérence stricte (cf section VI.3). Cependant le mécanisme d'invalidation des verrous nous permet d'améliorer la disponibilité tout en conservant la cohérence. L'exemple suivant illustre l'utilisation de ce mécanisme :

> *Soit un objet O1 verrouillé par le gérant de mémoire G1. Si ce gérant tombe en panne, l'objet O1 reste inaccessible tant que le site sur lequel s'exécutait G1 reste indisponible. L'invalidation, par le service Goofy, des verrous et de la copie de O1 que possède G1, permet à un autre gérant d'utiliser la dernière image de O1 toujours conservée en mémoire de stockage fiable. En agissant de la sorte, Goofy assure la fonction d'une mémoire de sauvegarde (certaines modifications ont pu être perdues) mais il offre surtout de la disponibilité.*

Les clones testent périodiquement la présence des gérants de mémoire (ils utilisent aussi les notifications de destruction d'un port de communication émises par le noyau Mach). Si un gérant de mémoire est considéré comme indisponible, tous les verrous qui lui sont associés, sont marqués dans l'état libérable. Notons que, tant qu'aucun autre gérant n'essaye de verrouiller ces objets, les verrous restent valides. Cela permet de résister à la panne temporaire d'une machine ou du réseau, puisqu'elle sera purement et simplement ignorée lorsque le gérant fautif réintégrera le système.

Par contre, lorsqu'un gérant de mémoire réintegre le système, toutes les copies des objets, dont les verrous ont été invalidés, sont détruites. Seule la copie du gérant qui possède le verrou est prise en compte par les clones. Cette politique peut donc conduire à la perte de modifications enregistrées par le gérant fautif.

VII.1.2.3 Localisation des objets

Le schéma de localisation des objets dans le prototype Guide/Mach est assez coûteux dans la mesure où un objet peut−être chargé en mémoire d'exécution par différents gérants de mémoire au cours de sa vie. Afin de décrire ce schéma de localisation, nous allons d'abord étudier le schéma original de localisation du prototype puis nous décrirons la version de ce schéma utilisée pour accéder au service Goofy.

Le schéma de localisation est basée sur l'utilisation des références systèmes qui identifient de manière interne les objets. Une référence système (sysref) est composée de deux champs (cf figure Fig. 7.2) : un premier champ identifie le système d'objets auquel appartient l'objet (c'est à dire le support physique de stockage de

l'objet) et un deuxième champ identifie l'objet au sein de son système d'objets (il s'agit en fait d'une estampille locale au système d'objet).

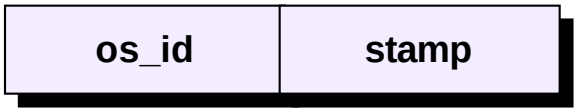

Fig. 7.2 : Format d'une référence système

La localisation d'un objet est réalisée en tenant compte des deux caractéristiques suivantes du prototype : chaque gérant de mémoire ne gère qu'un seul système d'objets et il n'existe qu'un seul système d'objets par site.

Il suffit donc de connaître l'association site/système d'objets pour être capable de localiser l'objet en mémoire de stockage et de s'adresser au gérant de mémoire du site en question (c'est ce gérant qui est responsable de la gestion en mémoire d'exécution du système d'objets du site) afin d'être capable de coupler l'objet dans l'espace d'adressage d'une application.

Ce très simple schéma de localisation ne fonctionne malheureusement pas pour les objets gérés par le service de stockage fiable Goofy. En effet, dans la mesure où on veut que le système puisse tolérer une panne de site, il nous a paru important d'être capable de faire gérer rapidement un objet par un nouveau gérant de mémoire. Cela implique donc qu'un objet soit potentiellement géré par différents gérants de mémoire au cours de son existence. Le schéma de localisation ne peut donc pas reposer sur l'hypothèse d'association statique entre les systèmes d'objets et les sites.

Nous avons donc séparé le traitement des objets fiables des autres objets. Pour ce faire, nous utilisons un identificateur de système d'objets particulier que nous associons à l'espace de conservation fiable. Cet identificateur est conservé dans une variable d'administration du système (`RPS_id`). Tout objet qui appartient à ce système d'objets est localisé en s'adressant directement au service Goofy. Le service Goofy conserve l'association objet fiable/gérant de mémoire afin d'être capable de rediriger les applications vers le gérant de mémoire en charge de l'objet. Le schéma de localisation est alors le suivant :

1. Si `os_id` est différent de `RPS_id` alors s'adresser directement au gérant du site `os_id`.
2. Sinon demander à Goofy l'identificateur du gérant en charge de l'objet.
3. Si l'objet est déjà géré sur un site désigné par Goofy, s'adresser au gérant de ce site.
4. Si l'objet n'est pas présent en mémoire d'exécution, choisir un site quelconque (en général, celui sur lequel s'exécute l'application qui fait la

demande de localisation) et s'adresser au gérant de ce site pour qu'il charge l'objet.

On constate qu'ainsi les objets fiables seront chargés et verrouillés par le premier gérant de mémoire qui en fait la demande. Si le taux de partage n'est pas trop important, cette politique tend à rapprocher les objets des applications (ou du moins de la première application qui les accède).

VII.1.3 Évaluation

Le serveur de stockage fiable Goofy a été conçu pour fournir un espace de conservation fiable aux applications s'exécutant sur le système distribué à objets Guide−1. Le serveur est capable de résister aux pannes de machines ainsi qu'aux partitions du réseau de communication tout en préservant la cohérence des objets. Il nous apparaît en effet important que le serveur garantisse la cohérence des objets.

Goofy essaye de fournir aux applications des objets fiables (c'est à dire robustes et disponibles) tout en garantissant leur cohérence. Les principales caractéristiques de Goofy sont :

- Le ***protocole de gestion de la duplication pessimiste,*** mis en œuvre par un algorithme de vote majoritaire, garantit la cohérence stricte des données répliquées.
- La ***capacité des clients*** (c'est à dire des gérants de mémoire) ***d'opérer localement*** (dès que les objets dont ils ont besoin, ont été chargés dans la mémoire d'exécution du gérant) permet d'offrir une grande disponibilité même en cas de partitionnement du réseau (similaire à ce qui est réalisé dans le système Coda).
- Le ***mécanisme d'invalidation des verrous*** augmente la disponibilité des objets dans la partition qui contient la majorité des clones du serveur Goofy (au prix de la perte éventuelle de quelques modifications).

La mise en œuvre du serveur Goofy sur le prototype Guide/Mach a servi à valider notre conception. Bien que cette réalisation n'ait pas été réellement optimisée, les performances obtenues au cours de plusieurs expérimentations [Chevalier 92b] ont montré que le surcoût induit par Goofy lors des opérations de stockage est raisonnable. Nous pensons que cette expérience a clairement démontré que l'utilisation d'un protocole pessimiste (même au prix d'une réduction du débit du système) était compatible avec les besoins des applications supportées par le système Guide−1.

Cependant et même si cette expérience peut être considérée comme positive, les solutions techniques mises en œuvre ont un certain nombre d'inconvénients qui limitent la capacité du système à fournir de très bonnes performances :

- L'algorithme de vote majoritaire utilisé lors de la phase de verrouillage des objets et lors de chaque opération de stockage offre une grande souplesse dans la gestion des machines (la panne ou le retour d'une machine est automatiquement pris en compte lors des phases de vote) mais il est surtout conçu pour coordonner un nombre important de clones (plus de dix).

  Or les expériences de Coda et Echo ont montré qu'il n'était pas nécessaire de stocker plus de trois copies d'une donnée afin de résister aux pannes (au moins dans la plupart des cas). On peut donc s'interroger sur l'utilité réelle et le coût des algorithmes utilisés par Goofy.

- L'objet est actuellement utilisé comme unité de verrouillage. Ceci a des conséquences sur la phase de chargement des objets. Or les objets dans le système Guide sont petits et nombreux. Même si tous les objets ne sont pas stockés dans la mémoire de stockage fiable, il paraît nécessaire de factoriser les opérations de verrouillage en augmentant la taille de l'entité verrouillée.

- Le mécanisme d'invalidation des verrous a un effet de bord en contradiction avec notre objectif de transparence. En effet, l'invalidation d'une copie et la restauration d'un objet à partir d'une copie conservé par les clones, peut revenir, du point de vue de l'utilisateur, à perdre des modifications. Or cette invalidation est décidée par le système, sans le prévenir, ce qui risque de le surprendre. La solution qui consiste à demander une confirmation à l'utilisateur à chaque restauration d'objet n'est pas envisageable : le nombre d'objet est trop grand. Par ailleurs, nous avons justement choisi un protocole pessimiste afin de décharger l'utilisateur de la gestion de la cohérence des objets.

  Ce mécanisme permet donc d'augmenter la disponibilité mais il risque de compliquer la tâche des utilisateurs.

En outre, nous aimerions augmenter l'autonomie des clients (c'est à dire des gérants de mémoire) afin de fournir, vu des applications, une plus grande résistance aux partitionnements du réseau. Cela implique notamment d'être capable de pré−charger sur le site du gérant les objets potentiellement utilisés par les applications (c'est la technique du "hoaring" utilisée dans Coda, cf sous−section VI.4.1).

VII.2 De Guide−1 à Guide−2

Lors de la conception du système Guide−2, nous avons été amené à repenser un certain nombre des choix de conception du système Guide−1. Certains de ces choix ont une influence directe sur l'architecture du serveur de stockage fiable. Dans la suite de la présente section, nous allons décrire les principaux changements introduit dans le système Guide−2 qui ont une influence sur le serveur.

Un des objectifs majeurs de Guide−2 est de définir et de bâtir une architecture de système modulaire et ouverte (cf section IV.1). Cela implique notamment que le système doit être capable d'utiliser des serveurs de stockage externes. La notion d'objet apparaît alors comme étant de trop haut niveau (cf sous−section III.2.3). De plus, la recherche de meilleures performances a entraîné l'introduction des grappes comme unité de transfert entre la mémoire de stockage et la mémoire d'exécution. Ce changement va dans le sens de l'accroissement de la taille de l'unité de verrouillage.

L'administration du système et principalement de la mémoire de stockage de Guide−2 est basée sur l'utilisation de montages. L'opération de montage associe à un gérant de mémoire une entité de conservation physique qui est le volume(3).

Il apparaît souhaitable de ne pas introduire de volume particulier pour le stockage fiable afin de bénéficier de l'ensemble des outils d'administration et de rendre le système plus souple dans sa configuration matérielle.

Le schéma de localisation doit nécessairement prendre en compte cette notion de montage et essayer d'en tirer partie pour être plus efficace. Nous désirons surtout éliminer l'envoi systématique d'une requête de localisation au serveur de stockage. Par ailleurs, nous voulons bénéficier de cette notion de montage pour fournir des mécanismes de base pour supporter des clients autonomes.

La principale amélioration que nous avons voulu apporter est de réduire significativement le coût du protocole de gestion de la duplication en l'absence de panne. Il apparaît en effet raisonnable d'estimer que les cas de pannes ne sont pas trop fréquents et d'essayer de privilégier le fonctionnement normal du système. Le protocole de vote majoritaire utilisé dans la première version de Goofy introduit un coût minimal constant (celui dû à la phase de vote) même lorsque tous les clones sont opérationnels. Nous avons donc introduit un nouvel algorithme basé sur la propagation asynchrone des modifications.

---

(3) Nota: un volume a plus ou moins le même rôle qu'un système d'objets dans Guide−1, tout en étant de plus bas niveau sémantique. Il est assimilé à une simple partition disque.

VII.3 Un serveur de stockage fiable pour Guide−2

La conception de la seconde version du serveur de stockage fiable Goofy a eu comme objectif principal d'essayer de tirer partie au maximum du rôle central joué par les gérants de mémoire dans la gestion de la persistance des données (cf sous−section IV.1.1).

Nous bénéficions, par exemple, du schéma de montage des volumes et notamment du fait qu'un volume n'est géré que par un seul et unique gérant de mémoire afin d'augmenter la localité des opérations de stockage (et donc les performances). Cette propriété est par ailleurs utilisée pour supprimer les opérations de verrouillage dans la mesure où nous assimilons la définition d'un montage par l'administrateur du système à un verrouillage de tout le volume.

La grappe étant utilisée comme entité de conservation en mémoire de stockage par les gérants de mémoire, elle est donc utilisée par Goofy comme unité de duplication en lieu et place de l'objet. Par la suite, lorsque nous utiliserons le mot copie, il s'agira exclusivement de la copie d'une grappe.

VII.3.1 Architecture

Le serveur de stockage utilise une architecture de type copie primaire/copies secondaires. La copie primaire d'une grappe est stockée sur le site du gérant de mémoire qui monte le volume auquel appartient la grappe. Les copies secondaires, que nous appellerons copies de secours, sont conservées par deux ou trois serveurs de stockage externes (c'est à dire situés sur des machines ne supportant pas forcément des applications Guide).

Les opérations de modifications des grappes sont effectuées localement par le service de stockage fiable (SSF) du gérant de mémoire (cf figure Fig. 4.11), puis elles sont propagées de manière asynchrones aux serveurs de secours. Cette propagation asynchrone est une caractéristique essentielle du serveur car elle permet de garantir de très bonnes performances. Notons cependant que cette solution a une influence certaine sur la cohérence des copies.

Les serveurs de secours utilisent localement le service de stockage rapide (SSR) pour stocker les grappes dans l'image locale des volumes fiables.

Lorsqu'un partitionnement du réseau se produit, le gérant de mémoire peut continuer à modifier sa copie. Les serveurs de secours sont remis à jour (du moins les copies conservées par eux) lorsque la panne est réparée (cf sous−section VII.3.4). Le cas particulier de la panne d'un gérant est traité dans la sous−section VII.3.3.

La figure Fig. 7.3 présente l'architecture du serveur de stockage fiable Goofy et son intégration avec le système Guide−2.

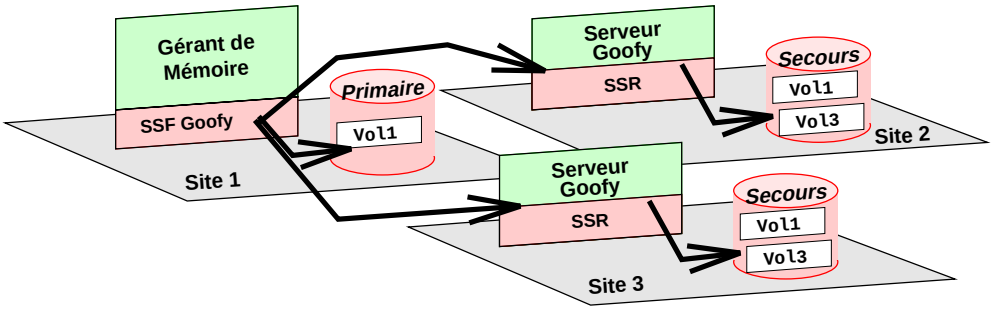

Fig. 7.3 : Architecture du serveur Goofy pour le système Guide−2

Les trois composants qui constituent le serveur de stockage fiable Goofy dans cette figure sont : le service de stockage fiable (ou service de stockage Goofy) inclus dans le gérant de mémoire du site 1 et qui gère les copies primaires des grappes (en l'occurrence celle du volume `Vol1`), et les deux serveurs de secours (ou serveurs de stockage Goofy) situés sur les sites 2 et 3. Le volume `Vol3` est monté par le gérant de mémoire d'un autre site.

Les principaux avantages de cette architecture sont :

- Le compromis entre la fiabilité fournie par les trois copies et l'utilisation des ressources des disques est particulièrement intéressant : la disponibilité est suffisamment élevée et l'on n'occupe pas trop d'espace sur disque.
- Le surcoût introduit par Goofy en cas d'absence de panne est minime dans la mesure où les modifications sont propagées de manière asynchrone.
- Le temps de réponse (vu du gérant de mémoire) est proche de celui du service de stockage local puisque les opérations de stockage sont exécutées localement.
- La cohérence stricte des données est garantie dans la mesure où il n'existe qu'une seule copie modifiable à un instant donné, la copie primaire.
- L'autonomie des gérants est assurée par la présence physique des volumes montés sur le disque local (cf sous−section VII.3.5)

VII.3.2 Fonctions principales

Propagation des modifications

Le mécanisme de mise à jour des copies des grappes sur les serveurs de secours est basé sur la propagation asynchrone des pages modifiées. De manière similaire à

[Bhide 92], nous transférons des pages complètes plutôt qu'un journal des modifications afin de minimiser les entrées/sorties et l'utilisation de la puissance de calcul sur le site primaire. Cette méthode augmente la charge du réseau mais elle devrait être compatible avec les performances susceptibles d'être fournies par les futurs réseaux de communication.

Par ailleurs, si le service de stockage sous−jacent fournit une journalisation des opérations de stockage (comme SpriteLFS [Rosenblum 91]), il est démontré dans [Burrows 92] qu'un algorithme de compression des données peut améliorer significativement les performances. Cette voie fait partie des expériences que nous devrions conduire dans le futur en conjonction avec la définition d'un mécanisme de propagation optimisé.

Plusieurs politiques peuvent être utilisées pour la propagation. Par exemple, on peut envoyer toutes les pages modifiées d'une grappe dès qu'elles ont été stockées sur le disque primaire ou envoyer des ensembles de pages afin de diminuer le nombre des entrées/sorties. Il s'agit généralement de définir un compromis entre les performances et la cohérence des copies ou encore le coût de restauration. Nous avons choisi la première politique car elle minimise le délai pendant lequel les copies de sauvegarde ne sont plus à jour. De plus, elle bénéficie intrinsèquement du regroupement des pages produit par la politique de stockage atomique des grappes : en effet, toutes les pages modifiées d'une grappe sont stockées en même temps de manière atomique par les gérants de mémoire.

Cependant, il faudrait évaluer différentes politiques afin de déterminer si l'utilisation de politiques de propagation adaptées à chaque grappe, permet d'améliorer les performances du système.

VII.3.3 Traitement de la panne d'un gérant de mémoire

Les gérants de mémoire jouant un rôle essentiel dans notre architecture, le système doit être capable de résister à la panne d'un gérant.

La solution utilisée dans le système ARM [Bhide 92] est d'enregistrer chaque modification auprès d'un serveur de journalisation avant de procéder à la modification. Le serveur de journalisation est situé sur une machine spéciale renforcée contre les pannes matérielles. Cette solution n'est pas souhaitable dans notre système car elle requièrt du matériel spécialisé ce que nous voulons justement éviter. De plus, la propagation et la journalisation des modifications sur le serveur entraînent un surcoût important.

Une autre solution est de garantir que les opérations de mise jour sont d'abord exécutées sur les serveurs de secours avant de l'être sur le site primaire. Si le site

primaire tombe en panne, les serveurs de secours possèdent une image à jour des données. Le principal inconvénient de cette politique, dite politique primaire/secondaire synchrone, c'est qu'elle entraîne un surcoût important même en l'absence de panne dans le système. Le serveur Harp [Liskov 91] utilise une unité électrique de secours pour garantir la re−copie de toutes les modifications sur disque avant l'arrêt du système.

Notre approche est basée sur l'expérience acquise avec la première version du serveur Goofy. Elle s'apparente à l'utilisation des copies de secours utilisées quotidiennement dans les entreprises. Aujourd'hui, toute entreprise soucieuse de préserver ses informations utilise des copies de secours sur bande ou disquette magnétique. En cas de panne d'une machine ou d'un disque, la copie de secours est utilisée pour restaurer sur une autre machine ou un autre disque une copie des informations précédemment sauvegardées. Il est possible que la copie de secours ne soit pas suffisamment à jour et que l'entreprise décide d'attendre que la machine soit réparée plutôt que de restaurer des données périmées.

Nous pensons que ce mode de fonctionnement est toujours d'actualité même au sein d'un environnement distribué dans lequel les utilisateurs et les applications partagent des données. Nous sommes même persuadés que la nature même des données stockées encourage ce mode de fonctionnement. On peut généralement diviser les données en deux catégories : les données privées à un utilisateur (ce sont des données qui peuvent être fréquemment modifiées) et les données publiques ou partagées (ce sont des données relativement stables). En cas de panne temporaire, il est probable qu'un utilisateur préfèrera attendre la réparation plutôt que de perdre une ou deux journées de travail. Par contre, pour les données partagées, il est souhaitable de les rendre le plus rapidement possible disponibles.

Le serveur de stockage autorise l'administrateur d'un système Guide à spécifier la politique appliquée aux différents volumes en cas de panne d'un gérant de mémoire (ou en cas de panne du site sur lequel s'exécute le gérant). Actuellement, nous étudions deux politiques :

- La ***politique d'attente de la réparation*** considère que tous les volumes montés sur le site en panne sont inaccessibles.

  Notons qu'un site peut−être considéré comme en panne alors qu'il est isolé suite à un partitionnement du réseau. Cette politique autorise l'utilisateur de ce site à continuer de travailler tout en étant isolé.

- La ***politique de restauration immédiate*** spécifie qu'un autre gérant doit prendre le relais du gérant en panne. Les volumes gérés par le gérant défaillant seront restaurés sur un nouveau site à partir de l'image des volumes conservées sur les serveurs de secours. Cette image peut éventuellement être

à jour (c'est à dire que toutes les modifications effectuées sur le site primaire ont été reportées) ou non (à cause de la propagation asynchrone).

Dans les deux cas, l'administrateur du système est prévenu et le serveur attend que celui−ci confirme l'opération avant de continuer plus en avant. L'administrateur peut ainsi adapter la gestion des volumes aux situations particulières qui ne manqueront pas de survenir.

Nous pensons qu'une telle approche est parfaitement acceptable. Par rapport à une gestion plus ou moins manuelle des sauvegardes, le délai entre la mise à jour de la copie primaire et la propagation des modifications sur les serveurs de secours est très faible (de l'ordre de la minute). On peut donc considérer que les copies de secours seront la plupart du temps à jour ou que la perte sera minime.

VII.3.4 Restauration d'un serveur de secours

Le principe qui gouverne la restauration d'un serveur de secours est de remettre à jour l'ensemble des copies de secours qui ont été modifiées pendant la panne du serveur. Une solution brutale et inefficace consiste à recopier l'intégralité des copies primaires sur le serveur de secours(4). Nous avons préféré utiliser une approche plus fine basée, comme dans Coda [Kistler 92], sur l'utilisation de l'information contenue dans les fichiers de journalisation.

Le service de stockage utilisé localement par Goofy gère un journal pour assurer l'atomicité des opérations de stockage des grappes (cf sous−section IV.2.3). Chaque opération de stockage est enregistrée dans le journal avant d'être reportée sur disque. On peut donc en comparant le journal du serveur de secours avec le journal primaire (celui du gérant de mémoire) déterminer avec précision quelles sont les opérations qui n'ont pas été effectuées par le serveur de secours. La figure Fig. 7.4 illustre cette propriété des journaux.

(4) Notons cependant que cette solution est utilisée lorsqu'un nouveau serveur de secours est installé dans le système.

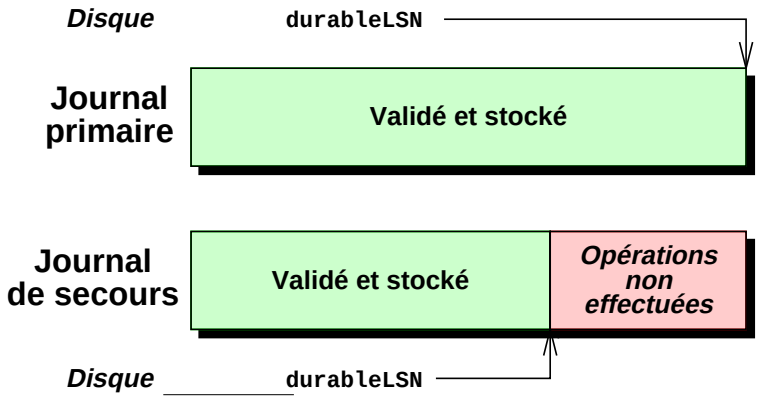

Fig. 7.4 : Utilisation des journaux pour la remise à jour

L'algorithme de restauration est donc assez simple :

1. Lorsque le serveur de secours redémarre, il détermine la liste des volumes dont il gère une copie de secours (à l'aide d'une table d'administration).
2. Pour chaque volume, le serveur envoie au gérant de mémoire (au service Goofy) qui a la charge de la copie primaire une requête de restauration qui contient la valeur du `durableLSN`(5) du volume connu du serveur de secours.
3. Le gérant de mémoire acquitte la requête en indiquant au gérant la valeur actuelle du `durableLSN` et il bascule dans une phase de restauration.
4. La phase de restauration consiste pour le gérant à envoyer toutes les opérations contenues entre le `durableLSN` primaire et le `durableLSN` du serveur de secours. Pour le serveur de secours, elle consiste à reporter normalement toutes les opérations reçues.
5. Lorsque le volume a été remis à jour, le serveur envoie un message d'acquittement au gérant de mémoire. A partir de cet instant, le serveur est à même de traiter les requêtes concernant ce volume.

Remarquons que pour des questions de synchronisation durant la phase de restauration, le gérant de mémoire bloque toutes les opérations de stockage sur ce volume. Cela a pour but d'éviter que de nouvelles opérations soient enregistrées pendant que le serveur est remis à jour, ce qui risquerait d'entraîner une remise à jour perpétuelle du serveur. Cet inconvénient n'est pas trop important car cela ne concerne qu'un seul volume à la fois et la procédure n'est pas sensée être exécutée fréquemment.

(5) Le pointeur dans le journal qui indique quelles opérations ont été répercutées sur le disque de stockage.

VII.3.5 Opérations en mode déconnecté

L'arrivée des machines portables a entraîné l'émergence de nouveaux besoins. Le support du travail mobile (ou du travail en mode déconnecté) a fait l'objet de nombreuses études notamment dans les systèmes de fichiers Coda [Kistler 92] et Ficus [Heidemann 92].

Le support de ce mode de travail ne faisait pas partie des objectifs du projet Guide. Cependant, il est apparu que le serveur de stockage fiable pouvait offrir une excellente base sur laquelle bâtir un tel support. Le but du travail en mode déconnecté est de permettre à un utilisateur de continuer à travailler et donc à modifier ses données alors même qu'il se trouve isolé du "reste du monde" (c'est à dire de son environnement normal de travail, réseau d'entreprise ou autre). Cela est très proche de ce que nous voulons offrir en cas de partitionnement du réseau, à la différence essentielle que la deconnexion est en général prévue et demandée par l'utilisateur.

Nous avons donc commencé à explorer quelques pistes. Une étude plus complète est actuellement menée afin de déterminer précisément quelles sont les fonctions nécessaires au support de la deconnexion et qui font actuellement défaut. Nous allons donc nous contenter de donner quelques indications.

L'architecture primaire/secondaires que nous utilisons, nous permet de garantir l'accès aux volumes locaux lors de la deconnexion. L'utilisateur peut modifier les grappes stockées localement. Les modifications sont automatiquement propagées sur les serveurs de secours lorsque la reconnexion intervient. Cependant, ce fonctionnement n'est que partiel dans la mesure où l'utilisateur ne peut accéder qu'à un faible sous−ensemble des volumes avec lesquels il peut potentiellement travailler.

Une solution envisageable est d'autoriser la recopie d'objets (ou de grappes) sur le disque local avant que la deconnexion n'intervienne et d'autoriser l'accès en lecture−seule (afin de ne pas mettre en péril la cohérence des données) aux données qui appartiennent à d'autres utilisateurs et en lecture−écriture aux données qui appartiennent au propriétaire de la machine. De manière similaire à la technique de "*hoaring*" utilisée dans Coda, le système peut conserver la trace des données auxquelles on accède le plus fréquemment afin de les copier automatiquement et par avance.

VII.3.6 Évaluation

La nouvelle version du serveur Goofy est en cours de développement. Une première version qui ne fournit pas la restauration des serveurs de secours est actuellement disponible.

La propagation asynchrone des modifications n'est pas complètement disponible en l'état ce qui ne nous permet pas d'en donner une évaluation précise. Cependant, la

version actuelle nous a permis de faire quelques mesures donnant une idée du coût maximal d'une configuration avec deux serveurs de secours.

Nous avons mesuré le coût du stockage d'une grappe de 200 pages (c'est à dire 800 Ko) lorsque les modifications sont envoyées de manière synchrone aux serveurs de secours : le service Goofy stocke d'abord la copie primaire localement, puis il envoie les modifications au premier serveur de secours ; lorsque celui−ci a acquitté la mise à jour, le service envoie les modifications au second serveur ; à la réception du deuxième acquittement, l'opération est considérée comme terminée.

Le coût total du stockage de la grappe sur le site primaire et les deux serveurs de secours est de 29,4 s. Par comparaison, le coût du stockage de la même grappe par le service de stockage local est de 5,4 s. La figure montre comment se décompose ce coût.

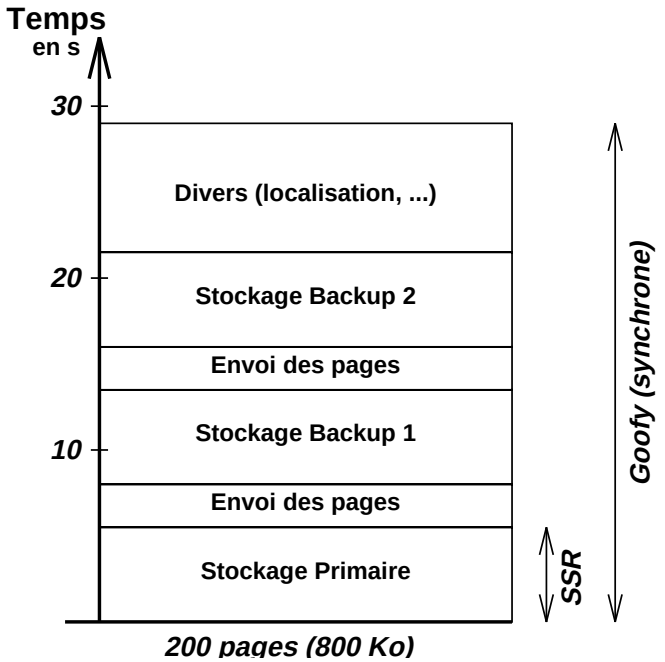

Fig. 7.5 : Goofy (synchrone), écriture page par page

On constate que le schéma synchrone grève lourdement les performances. La localisation des serveurs de secours ainsi que la lecture des descripteurs de grappe par les serveurs paraissent avoir un coût elevé (environ 6 s). Le surcoût de ce schéma de stockage par rapport au simple stockage local est d'un facteur 5.

La diffusion asynchrone et en parallèle des requêtes de mise jour doit fortement diminuer le temps total d'exécution et nous ramener aux alentours de 8 s (6 s pour le stockage local plus les coûts de localisation). En fonction de la charge de la machine et du réseau, le surcoût du au stockage fiable sera donc compris entre 1,2 et 5. Ce qui

semble raisonnable dans la mesure où il sera essentiellement visible lors de la terminaison d'une application.

VII.4 Conclusion

Le serveur Goofy a pour but de garantir la disponibilité et la robustesse des grappes en espace de conservation. Les principales caractéristiques de la nouvelle version de Goofy sont :

- L'algorithme de ***propagation asynchrone des modifications*** privilégie le fonctionnement en l'absence de panne et fournit de très bonnes performances.
- Le ***protocole de gestion de la duplication pessimiste*** mis en œuvre par un algorithme de copie primaire/copies secondaires, garantit la cohérence stricte des données dupliquées en privilégiant la copie du site du gérant de mémoire.
- La ***capacité des clients*** (c'est à dire des gérants de mémoire) ***d'opérer localement*** sur les copies primaires assure une grande disponibilité même en cas de partitionnement du réseau.
- Le ***mécanisme d'invalidation des volumes*** permet de repartir d'une version stockée sur les serveurs de secours en cas de panne prolongée du site primaire (au prix de la perte éventuelle de quelques modifications).
- L'***utilisation d'information contenue dans les fichiers de journalisation*** permet d'identifier rapidement et simplement les mises à jour qui n'ont pas pu être propagées sur un serveur secours.

Par comparaison avec la première version du serveur Goofy, la nouvelle version peut sembler moins avancée (au niveau des algorithmes notamment). Cependant, elle fournit de nouvelles fonctions telles que le support de la déconnexion et doit s'avérer beaucoup plus performante (grâce au protocole asynchrone).

On peut considérer en quelque sorte que l'expérience acquise avec le premier prototype à laquelle s'ajoute celle acquise avec le développement des gérants de mémoire, nous a amené à être plus pragmatiques. Ainsi, nous avons privilégié le mode de fonctionnement sans panne et nous avons limité le degré de copie des grappes à quelques exemplaires.

On peut regretter que l'état d'avancement du nouveau serveur ne nous permette pas de dresser un bilan plus détaillé des caractéristiques internes du serveur. Nous

sommes cependant convaincus que les solutions employées dans le serveur de stockage fiable Goofy vont nous permettre de fournir à la fois la performance et la qualité de service attendues par les applications distribuées qui seront supportées par le système Guide.

Chapitre VIII

Conclusion

Le travail présenté relève du domaine des systèmes informatiques répartis. Il s'est articulé autour de deux thèmes majeurs : le support de la persistance et la disponibilité des données dont nous avons décrit la mise en œuvre dans le système réparti à objets Guide.

L'objectif du projet Guide est la conception d'un environnement (système, langages et outils) permettant le développement d'applications coopératives, structurées en termes d'objets partagés et persistants. Après une première version, Guide−1 [Balter 91], qui a été développée sur le système Unix entre 1987 et 1990, une seconde version du système a été réalisée sur le micro−noyau Mach 3.0 fournie par l'OSF (Open Software Foundation) [Balter 93]. Notre travail s'inscrit dans la phase de conception et de réalisation de cette seconde version appelée Guide−2.

Le principal résultat du projet Guide−2 est la réalisation d'un noyau de système réparti opérationnel sur un ensemble de stations de travail (une douzaine de compatibles IBM PC) connectées par un réseau Ethernet. Ce noyau, appelé Eliott [Rousset 93], fournit la base nécessaire à l'exécution d'applications réparties programmées à l'aide du langage orienté−objet Guide et d'un C++ étendu.

Le noyau Eliott est constitué de trois composants indépendants les uns des autres

- La machine à grappes qui gère des unités non−typées et non−structurées de partage et de conservation appelées grappes.
- La machine d'exécution qui gère les structures d'exécution réparties qui mettent en œuvre le modèle d'exécution de Guide.
- La machine à objets qui met en œuvre les objets dans les grappes [Hagimont 93] et gère efficacement leur adressage et leur protection [Chevalier 94b].

Le noyau Eliott est le résultat du travail d'une équipe. Les spécifications ont duré environ dix huit mois. Le développement a commencé il y a trois ans et se poursuit. La programmation d'Eliott a nécessité environ 60.000 lignes de langage C. Des

services d'administration et de désignation symbolique ont également été développés afin que le système soit utilisable dans de bonnes conditions.

Le noyau Eliott et le compilateur Guide sont actuellement utilisés par des étudiants de DEA et de thèse qui s'en servent comme plate−forme pour mener des expériences dans le cadre de leurs activités de recherche. Des applications ont également été développées en langage Guide à des fins de démonstrations. La plate−forme Guide−2 a notamment été présentée aux démonstrations suivantes :

- ***Enterprise'93***, manifestation organisée à Boston, MA en juin 1993, par Bull, le MIT, l'OSF sur les applications des systèmes d'information à l'entreprise.
- ***Forum des Recherches en Informatique 1993***, rencontre des Projets de Recherche Coordonnés (PRC) avec l'industrie, organisé à Paris en Juin 1993.
- ***École d'été Bull−IMAG−INRIA*** à Autrans (Septembre 1993).
- Premier ***workshop*** du projet ESPRIT ***Broadcast*** (Basic Research On Advanced Distributed Computing: from Algorithms to SysTems), organisé en octobre 1993 à Newcastle upon Tyne.

Par ailleurs, le noyau Eliott sert de base à une opération de transfert de technologie au sein de la compagnie Bull. Ce transfert a pour but la réalisation d'une plate−forme distribuée à objets (Oode). Cette action est conduite par une équipe de développement formée, en partie, par le personnel Bull ayant participé à la mise en œuvre du noyau Eliott.

Le rôle de ce chapitre est triple. Dans un premier temps (section VIII.1), nous faisons un résumé du travail que nous avons réalisé. Ce résumé couvre aussi bien les aspects liés à l'implantation des mécanismes qui fournissent la persistance dans le système Guide que les aspects plus prospectifs liés à la tolérance aux pannes. Puis, nous donnons, dans la section VIII.2, notre vision de l'évolution du domaine dans les années à venir. Enfin, dans la section VIII.3, nous décrivons quelques perspectives nées de ce travail.

VIII.1 Contribution

La recherche dans le domaine des systèmes d'exploitation, et notamment la recherche appliquée (qui est une composante essentielle de l'environnement industriel dans lequel ce travail s'est déroulé), consiste essentiellement à choisir de "bons" compromis entre des buts contradictoires. L'objectif final est toujours de bâtir un système d'exploitation cohérent, fonctionnel et efficace.

La définition du "bon" compromis et de la définition de la "bonne" plate−forme système est donc à la base de notre travail. Notre analyse du support de la persistance (chapitre II) s'est donc tout naturellement orientée vers l'analyse, au moyen du modèle de persistance de Thatte, des choix de réalisation et des fonctions fournies par quatre systèmes : Clouds [Dasgupta 90], DSR [Kato 93], Cricket [Shekita 90] et O_2 [Deux 91].

Cette étude nous a permis d'isoler quatre fonctions fondamentales d'un système support de la persistance.

- La ***désignation***, c'est à dire la manière dont les entités conservées sont identifiées. Le choix d'une technique de désignation plus ou moins liée au support physique des données, résulte d'un compromis entre efficacité de la localisation et mobilité des entités désignées.
- Le ***partage***, c'est à dire la mise en œuvre de l'accès concurrent et simultané aux données par les applications. A ce niveau, les concepteurs de systèmes cherchent à offrir le maximum de souplesse pour la meilleure efficacité possible.
- La ***protection***, c'est à dire, principalement, le contrôle des droits d'accès aux données. L'objectif est de garantir une sécurité maximum pour un coût de mise en œuvre des mécanismes minimum.
- Le ***stockage***, c'est à dire la mise en œuvre de la conservation des données au travers de fonctions garantissant (ou non) la cohérence de l'espace de stockage après chaque opération. Là encore, le compromis est à trouver entre la souplesse des mécanismes et leur efficacité. Les besoins d'efficacité amènent les concepteurs de système à privilégier des transferts sur disque importants, alors que les applications tendent, elles, à privilégier une gestion très fine des données.

L'expérience acquise lors de cette étude nous a montré que la qualité d'un système résulte tout autant de la sélection des "bons" mécanismes que de leur utilisation de manière optimale. En ce sens, des politiques d'utilisation qui tirent pleinement partie des mécanismes fournis sont la condition essentielle à l'obtention de très bonnes performances. Par exemple, la grappe minimise le coût du partage d'objet, à condition que la politique de regroupement des objets dans les grappes soit bien choisie (c'est à dire qu'elle soit adaptée au comportement des applications).

Or il est souvent difficile, voire impossible, de définir au niveau des couches basses du système, des politiques qui optimisent de manière globale la gestion des quatre critères précédemment cités. Ceci est essentiellement dû au fait que le système ne peut "deviner" le comportement des applications. Cette tâche est d'autant plus

ardue que l'on cherche à gérer (la persistance) des entités de haut niveau sémantique que sont les objets.

L'objectif de notre travail était d'essayer de concilier un support efficace de la persistance avec les besoins des applications réparties qui sont la cible du projet Guide. Nous avons donc suivi une approche qui consiste à fournir un support de la persistance de bas niveau car c'est celle qui nous offrait le plus de souplesse dans la définition des politiques d'utilisation.

> Le support de la persistance est réalisé, de manière répartie, par la machine à grappes. La grappe est une entité, non−structurée et non−typée, de conservation des données. Elle est utilisée par les couches hautes du système Guide comme entité de regroupement et de stockage des objets manipulés par les langages de programmation.

La définition d'une interface de bas niveau pour le support de la persistance, qui dépasse les besoins du seul système Guide et qui peut être vu comme un support générique de persistance, est un des principaux résultats de notre travail.

Par ailleurs, nous avons porté une attention toute particulière à la définition d'une architecture modulaire et ouverte. La séparation de l'espace d'exécution de l'espace de conservation revêt un caractère fondamental dans l'accomplissement de cet objectif. Cette séparation accentue, de plus, le caractère générique de la machine à grappes.

La réalisation de la machine à grappes a grandement bénéficié de la possibilité offerte par le micro−noyau Mach 3.0 de définir des gestionnaires de mémoire externes(1). Cela nous a permis d'implanter notre architecture sans sacrifier les performances.

On retrouve la recherche du "bon" compromis à propos de la tolérance aux pannes qui fut le deuxième volet de notre étude.

Comme point de départ, à l'exploration de ce nouveau domaine de recherche au sein du projet Guide, et afin de prolonger le travail effectué sur le support de la persistance, nous nous avons concentré nos efforts sur le problème du stockage fiable et de la disponibilité des données.

Nous avons vu au chapitre VI que la disponibilité est généralement mise en œuvre par des techniques de duplication des données. Il existe deux grandes classes d'algorithmes pour gérer cette duplication :

---

(1) Notons que cette possibilité est généralement offerte par tous les micro−noyaux.

- Les ***algorithmes optimistes*** qui mettent en œuvre une ***cohérence lâche*** des copies des données. Ces algorithmes privilégient l'efficacité et surtout la disponibilité au détriment de la cohérence des données.
- Les ***algorithmes pessimistes*** qui mettent en œuvre une ***cohérence stricte*** (toutes les copies reflètent le "même" état). Ils sont capables de résister aux partitionnements du réseau mais peuvent réduire fortement la disponibilité des données.

Il s'agit ici de faire un compromis entre la cohérence des données, leur disponibilité, mais aussi les performances du système (en l'absence ou non de panne) ainsi que la qualité du service fournie (par exemple, la résistance aux partitionnements du réseau).

Comme support de notre étude, nous avons présenté deux systèmes de gestion de fichiers : Coda [Satyanarayanan 90] et Echo [Birrell 93], qui nous paraissent, en ce sens, exemplaires. Les choix de conception qui les régissent : un algorithme optimiste pour Coda afin de garantir le maximum de disponibilité à des données faiblement partagées et un algorithme pessimiste pour Echo afin de supporter le développement par des équipes industrielles de gros logiciels, ont été validés en situation par de nombreux utilisateurs.

L'expérience que nous avions acquise dans ce domaine au cours d'une première étude, nous a conduits à suivre une approche plus pragmatique et à privilégier les performances du système en l'absence de panne. Compte tenu du fait que les applications devant utiliser le système, accèdent à un grand nombre de données, nous avons privilégié les politiques pessimistes qui garantissent la cohérence des données en cas de panne.

Nous avons défini un serveur de stockage fiable structuré selon une architecture primaire/secondaire.

Les principales caractéristiques techniques du serveur sont :

- La mise à jour asynchrone des copies secondaires (dites copies de secours) permettant d'obtenir le stockage fiable des données au prix d'un très faible surcoût en l'absence de panne.
- La possibilité de restaurer un site primaire à partir des copies secondaires (au prix de la perte éventuelle de certaines modifications).
- La ré−utilisation de l'information de journalisation gérée par le service de stockage afin de mettre en œuvre, de manière simple et efficace, la restauration, suite à une panne quelconque, d'un serveur de secours.

La mise en œuvre de ce serveur, bien qu'incomplète en l'état, nous ouvre des perspectives intéressantes. Elle a, de plus, servi à valider l'architecture globale de

notre système et notamment sa qualité "d'ouverture". L'intégration du serveur de stockage fiable n'a nécessité que la mise à jour d'une table d'indirection dans le système.

VIII.2 Évolution du domaine

Le domaine des systèmes informatiques répartis et notamment des systèmes à objets est actuellement étudié par de nombreuses équipes de recherche. Par ailleurs, on constate que les milieux industriels, par des initiatives telles que celles menées au sein de l'OMG (Object Management Group), visent à structurer, voire à normaliser, le domaine. Cette tentative est d'autant plus nécessaire que les systèmes répartis deviennent des outils de base dans un nombre croissant de domaines d'activité. Ainsi, on assiste à leur intégration avec les technologies utilisées dans les systèmes d'information (au sens large) : bases de données, génie logiciel, intelligence artificielle ainsi qu'interface homme/machine.

L'intégration s'accompagne de l'émergence de nouveaux besoins (coopération, multimédia, réseaux à grandes distances). Ces besoins sont liés aux nouveaux domaines d'applications et aux modes de travail que les concepteurs de systèmes répartis envisagent de supporter dans le futur. Par ailleurs, l'amélioration de l'infrastructure technique : machines, noyaux de systèmes et réseaux de communication, offre des possibilités de performance et de services jusqu'ici non disponibles.

VIII.2.1 Évolution des besoins

On constate l'émergence d'un mode de travail caractérisé par la ***coopération*** autour d'une tâche commune d'équipes géographiquement réparties, avec ***partage*** étroit d'informations et communication en temps réel. Le succès des infrastructures de communication et d'accès à des informations distantes, actuellement disponibles, telles que le World Wide Web, Wais et Gopher, prouve, si il en était besoin, l'importance nouvelle de ce mode de travail. Citons quelques exemples d'applications coopératives :

- La gestion de documents, au sens large, y compris des données multimédia [Bly 93] (images, son, etc), ou des ensembles de documents interconnectés (hypertextes et hypermédia [Grønbæk 94]) ;
- Les outils d'aide à la prise de décision dans un groupe [Nunamaker 91], c'est à dire la communication par courrier électronique ("e−mail" et "news") ou par panneaux d'affichage ("shared whiteboard") ainsi que la gestion d'agendas ;
- L'ingénierie simultanée, c'est à dire la gestion coopérative de projet ou le support système pour la conception assistée en environnement réparti ;

- Le développement de logiciel, qui comprend la gestion de versions et de configurations complexes.

Deux autres besoins importants qui se manifestent, sont l'***intégration*** de techniques différentes (exemples : gestion de logiciel et documentation technique, communication et données multimédia) et la nécessité de ***réutilisation*** des applications existantes, qu'il serait trop coûteux de réécrire.

Les besoins croissants de communication et de partage de l'information, ainsi que l'intégration de plus en plus étroite de l'informatique et des télécommunications, conduisent au développement de systèmes de grande étendue géographique mettant en jeu des organisations différentes. Ces facteurs amplifient les problèmes de conception et de mise en œuvre des systèmes, et introduisent des problèmes nouveaux pour la désignation des entités, la maîtrise des performances, la sécurité, la fiabilité et l'administration.

VIII.2.2 Évolution des techniques

Les systèmes informatiques en temps partagé sont remplacés par des réseaux de postes de travail et de serveurs (de fichiers ou de calcul). L'évolution des techniques correspondantes, longtemps stable, s'accélère actuellement :

- L'apparition de nouveaux processeurs RISC, rapides, avec des capacités étendues d'adressage virtuel (64 bits) rend possible la représentation en mémoire virtuelle de l'ensemble de la mémoire persistante disponible (au sein d'un même réseau local). Les projets Opal [Chase 92] et Mungi [Heiser 93] explorent actuellement cette voie.
- La construction de réseaux très rapides de communication (par exemple, les débits prévisibles des réseaux ATM sont au delà du Gbit/s [Traw 93]), permet d'envisager des solutions nouvelles au partage d'information (sous la forme d'une mémoire virtuelle répartie, notamment), mais aussi l'intégration dans un même système d'applications ayant des bases temps réel (multimédia) et d'applications interactives.
- Le développement de l'informatique "nomade", c'est à dire l'accès quasi permanent d'utilisateurs mobiles à des services via de nouveaux systèmes de communication (grâce aux satellites et aux réseaux de communication par ondes radiophoniques). Les projets Coda et Ficus visent à supporter ce nouveau mode de travail [Kistler 92][Heidemann 92].

La conception des systèmes et des applications réparties fait intervenir des domaines divers de l'informatique : systèmes d'exploitation, langages et environnements de programmation, bases de données, protocole de communication. L'interaction croissante entre ces domaines conduit à l'émergence de thèmes communs, notamment autour des langages à objets persistants, de la gestion de la

mémoire (mémoire virtuelle, stockage d'information), du partage d'objets complexes, des mécanismes transactionnels ainsi que du support de communication multimédia.

Par ailleurs, un des défauts majeurs des systèmes actuellement disponibles sur le marché, est qu'ils ont été conçus à la fin des années soixante−dix, début des années quatre−vingt. En conséquence de quoi, les politiques qu'ils mettent en œuvre ne sont pas adaptées au comportement des nouvelles applications. Brian Bershad cite dans [Bershad 94], la gestion de la mémoire virtuelle et la politique de remplacement des pages dans Unix qui suppose une bonne localité dans les références. Or, les applications de recherche d'information et les applications multimédia ont un comportement aléatoire ou des contraintes temporelles à respecter, qui ne correspondent pas à ce comportement type.

Les bases de données sont un exemple typique d'applications qui n'ayant pu se satisfaire des services généraux fournis par les systèmes d'exploitation, ont reconstruit des services ad hoc au−dessus de ces mêmes systèmes [Stonebraker 81].

Nous pensons que le défi majeur dans les années à venir est la définition d'un système qui soit capable de supporter des domaines d'applications variés et exigeants (en termes de services et de performances) tels que les grandes bases de données, le multimédia et les applications parallèles. Un tel support ne peut, vu la diversité des besoins, se concevoir comme étant universel. C'est pourquoi, la définition de noyaux de système configurables est un des thèmes de recherche qui émerge dans la communauté système (cf le noyau SPIN [Bershad 94] et dans une moindre mesure, le système Spring [Khalidi 93]).

De plus, les aspects liés à la fiabilité et la sécurité, qui sont quasi inexistants dans les produits actuellement sur le marché, vont prendre une importance considérable dans le futur.

VIII.3 Perspectives

Nos perspectives de travail s'inscrivent dans cette double évolution des besoins et des techniques. Nous pensons travailler dans deux directions : d'une part mener des expérimentations sur la plate−forme Eliott et d'autres part réfléchir à ce que doivent être les mécanismes à mettre en œuvre pour disposer d'un système capable de répondre aux nouveaux besoins.

VIII.3.1 Expérimentations

Nous aimerions bénéficier de la disponibilité de la plate−forme Guide−2 pour conduire des expérimentations dans les domaines suivants :

- Les politiques adaptatives de pré−chargement des pages [Grimsrud 93], que se soit pour améliorer les performances comme dans [Song 93] ou

pour offrir des garanties de disponibilités des pages à un instant donné (indispensable pour le multimédia, cf [Staehli 93]).

Nous nous proposons de conserver un historique d'accès aux grappes et de "rejouer" cet historique au cours des utilisations futures des grappes.

- La définition de politiques utilisant au mieux les mécanismes fournis par le système, passe par la collaboration des langages (c'est à dire des compilateurs) et des applications avec le système. La mise en place de "capteurs" permettant de caractériser le comportement des applications a déjà fait l'objet d'une première étude [Bellissard 93].

  La définition et le contrôle du regroupement des objets dans les grappes, vu son importance au niveau des performances, sera une de nos priorités. Pour ce faire, nous nous proposons de rendre visible la notion de grappe au niveau des langages de programmation (cf sous−section IV.1.2.4). Il s'agit là d'un point délicat qui demande à être étudié et traité avec précaution.
- L'utilisation de bancs de mesures "standards" (et de niveau langage) comme OO1 de Cattell et Skeen [Cattell 92] ou ACOB, utilisé pour évaluer les performances de O_2 [DeWitt 90], doit nous guider dans l'évaluation des mécanismes de base de la machine à grappes.
- L'étude de la pagination distribuée, décrite dans la section V.2.3, doit être poursuivie, si possible sur un réseau à haut débit.
- La définition et la mise en œuvre de politiques de propagation des mises à jour, adaptées aux différents types de grappes, doit nous permettre d'améliorer les performances du serveur de stockage fiable Goofy.
- Le support de la déconnexion des utilisateurs est en cours de réalisation. L'étude des problèmes spécifiques posés par le travail "nomade" [Lu 94], est une des actions de recherche que nous aimerions conduire dans le futur.

VIII.3.2 Plate−forme

On assiste depuis quelques années au rapprochement entre le domaine des bases de données et celui des systèmes d'exploitation. Le support de la persistance, tel que nous l'avons défini et réalisé pour le système réparti à objets Guide, semble très proche de ce qui est nécessaire pour supporter un langage de programmation pour base de données (cf le langage PEPLOM [Dechamboux 93]). Notre objectif est donc de définir et de réaliser une infrastructure de système adaptée à la gestion efficace et sûre de données persistantes réparties et partagées, en prenant en compte les besoins des bases de données.

Dans un premier temps, nous nous proposons d'étendre la machine à grappes en lui ajoutant des mécanismes similaires aux mécanismes transactionnels fournis classiquement par les bases de données.

Cette première étape doit nous permettre ensuite de définir un noyau de persistance qui soit extensible et configurable (nous voulons continuer d'offrir un environnement d'exécution pour les applications coopératives). Cette démarche est similaire à celle suivie, pour ce qui concerne les systèmes d'exploitation, dans le projet SPIN [Bershad 94].

Par ailleurs, on assiste à l'émergence de nouveaux modèles de construction d'applications répartis qui ne sont pas issus du domaine des langages à objets. Les langages fonctionnels, tels que Facile [Thomsen 93], proposent par exemple la notion d'agents (c'est à dire de fonctions transmissibles et exécutables à distance) comme moyen de structuration des applications. Afin de valider le caractère générique de la machine à grappes, tout en étendant ce type de langage en leurs ajoutant la notion de persistance (dans un environnement réparti), nous envisageons de conduire une étude visant à porter le langage Facile sur notre plate−forme.

Bibliographie

[Accetta 86] M.J. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian et M. Young, ''Mach: A new kernel foundation for Unix development'', *Proceedings of the USENIX 1986 Summer Conference*, pp. 93−112, USENIX Association, Eté 1986.

[Atkinson 83] M. P. Atkinson, P. Bailey, K. Chisholm, P. Cockshott et R. Morrison, ''An approach to persistent programming'', *Computer Journal*, 26(4), pp. 360−365, Novembre 1983.

[Atkinson 87] M. P. Atkinson et O. P. Buneman, ''Types and Persistence in Database Programming Languages'', *ACM Computing Survey*, 19(2), pp. 105−190, Juin 1987.

[Amaral 92] P. Amaral, C. Jacquemot et R. Lea, ''A model for persistent shared memory addressing in distributed systems'', *Proceedings of the 3rd International Workshop on Object Oriented Operating Systems (IWOOOS'92)*, édité par L.−P. Cabrera and E. Jul, Paris, Septembre 1992.

[Balter 85] R. Balter, *Maintien de la cohérence dans les systèmes d'information répartis*, Doctorat Es Sciences, Université Scientifique et Médicale de Grenoble et Institut National Polytechnique de Grenoble, Juillet 1985.

[Balter 91] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville et G. Vandôme, ''Architecture and implementation of Guide, an object−oriented distributed system'', *Computing Systems*, 4(1), pp. 31−67, Hiver 1991.

[Balter 93] R.Balter, P.Y. Chevalier, A. Freyssinet, D. Hagimont, S. Lacourte et X. Rousset de Pina, ''Is the Micro−Kernel Technology well suited for the support of Object−Oriented Operating Systems: the Guide Experience'', *Proceedings of the 2nd USENIX Symposium on Microkernels and Other Kernel Architectures (MOKA)*, pp. 1−11, USENIX Association, San Diego, CA, Septembre 1993.

[Banâtre 91] M. Banâtre, P. Heng, G. Muller et B. Rochat, ''How to Design Reliable Servers using Fault Tolerant Micro−Kernel Mechanisms'', *Proceedings of the 1991 USENIX Mach Symposium*, pp. 223−231, USENIX Association, Monterey, CA, Novembre 1991.

[Banâtre 92] M. Banâtre, Y. Belhamissi and I. Puaut, ''Some features of Gothic: a Distributed Object−Based System'', *Proceedings of the 3rd International Workshop on Object Oriented Operating Systems (IWOOOS'92)*, édité par L.−P. Cabrera and E. Jul, pp. 117−123, Paris, Septembre 1992.

[Bartlett 81] J. A. Bartlett, ‘‘A NonStop Kernel’’, *Proceedings of the 8th ACM SOSP*, ACM Press, Décembre 1981.

[Bellissard 93] L. Bellissard, *Contrôle et Configuration pour applications réparties*, DEA d’Informatique, Institut National Polytechnique de Grenoble et Université Joseph Fourier − Grenoble I, Juin 1993.

[Benzaken 90] V. Benzaken et C. Delobel, ‘‘Dynamic clustering strategies in the O_2 object−oriented database system’’, *Proceedings of the 4th Workshop on Persistent Object Systems: Desgin, Implementation and Use*, Martha’s Vineyard, MA, Septembre 1990.

[Bernabeu 88] J.M. Bernabeu−Auban, P.W. Hutto, M.Y.A. Khalidi, M. Ahamad, W.F. Appelbe, P. Dasgupta, R.J. LeBlanc et U. Ramachandran, *The Architecture of Ra: A Kernel for Clouds*, (GIT−ICS−88/25), School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, 1988.

[Bershad 94] B. N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage et E. Gün Sirer, *SPIN − An Extensible Microkernel for Application−specific Operating System Services*, (TR 94−03−03), University of Washington, Seattle, WA, Février 1994.

[Bhide 92] A. Bhide, ‘‘Experiences With Two High Availability Designs’’, *Proccedings of the 2nd Workshop on Management of Replicated Data*, pp. 51−54, Monterey, CA, Novembre 1992.

[Birrell 93] A. D. Birrell, A. Hisgen, C. Jerian, T. Mann et G. Swart, *The Echo Distributed File System*, (SRC Research Report 111), Digital Equipment Corporation, Palo Alto, Septembre 1993.

[Birman 87] K. P. Birman et T. A. Joseph, ‘‘Reliable Communication in the Presence of Failures’’, *ACM Transaction on Computer Systems*, 5(1), pp. 47−76, Février 1987.

[Black 86] A. P. Black, N. Hutchinson, E. Jul et H. Levy, ‘‘Object structure in the Emerald system’’, *Proceedings of the ACM conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA’86)*, Portland OG, édité par *ACM SIGPLAN Notices*, 21(11), pp. 78−86, Novembre 1986.

[Black 90] A. P. Black et Y. Artsy, ‘‘Implementing Location Independent Invocation’’, *IEEE Transactions on Parallel and Distributed Systems*, 1(1), pp. 107−119, Janvier 1990.

[Bly 93] S. A. Bly, S. R. Harrison et S. Irwin, ‘‘MediaSpaces: Bringing People Together in a Video, Audio and Computing Environment’’, *Communications of the ACM*, 36(1), pp. 28−47, Janvier 1993.

[Boyer 91a] F. Boyer, J. Cayuela, P.Y. Chevalier, A. Freyssinet et D. Hagimont, ‘‘Supporting an object−oriented distributed system: experience with Unix, Chorus and

Mach'', *Proceedings of the 2nd Symposium on Experience with Distributed and Multiprocessor Systems (SEDMS)*, pp. 283−299, USENIX Association, Atlanta, GA, Mars 1991.

[Boyer 91b] F. Boyer, ''A Causal Memory'', *Proceedings of the USENIX Mach Symposium*, pp. 41−57, USENIX Association, Monterey, CA, Novembre 1991.

[Bryant 93] B. Bryant, S. Sears, D. Black et A. Langerman, *An Introduction to Mach 3.0's XMM Subsytem*, Operating Systems − Collected Papers Vol. 2 OSF Research Institute, Octobre 1993.

[Burrows 92] M. Burrows, C. Jerian, B. Lampson et T. Mann, ''On−Line Data Compression in a Log−Structured File System'', *Proceedings of ASPLOS V*, pp. 2−9, ACM Press, Novembre 1992.

[Cahill 93] V. Cahill, R. Balter, X. Rousset de Pina et N. Harris (Eds.), *The Comandos Distributed Application Plateform*, ESPRIT Research Reports Series, Springer−Verlag, Berlin Heidelberg, 1993.

[Cattell 92] R. G. G. Cattell et J. Skeen, ''Object Operation Benchmark'', *ACM Transactions on Database Systems*, 17(1), pp. 1−31, Mars 1992.

[Chase 92] J. S. Chase, H. Levy, M. Baker−Harvey et E. D. Lazowska, ''Opal: A Single Address Space System for 64−bits Architectures'', *Proceedings of the 3rd Workshop on Workstation Operating Systems (WWOS III)*, pp. 80−85, Key Biscayne, FL, Avril 1992.

[Chang 84] J.−M. Chang et N. F. Maxemchuck, ''Reliable Broadcast Protocols'', *ACM Transactions on Computer Systems*, 2(1), pp. 39−59, Août 1984.

[Chevalier 92a] P.Y. Chevalier, D. Hagimont, S. Krakowiak et X. Rousset de Pina, ''System Support for Shared Objects'', *Proceedings of the 5th ACM European SIGOPS Workshop*, Le Mont−Saint−Michel, Septembre 1992.

[Chevalier 92b] P.Y. Chevalier, ''A Replicated Object Server for a Distributed Object−Oriented System'', *Proceedings of 11th Symposium on Reliable Distributed Systems (S.R.D.S.)*, pp. 4−11, IEEE Computer Society Press, Houston. TX, Octobre 1992.

[Chevalier 93a] P.Y. Chevalier, A. Freyssinet, D. Hagimont, S. Krakowiak, S. Lacourte et X. Rousset de Pina, ''Experience with Shared Object Support in the Guide System'', *Proceedings of the 4th Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, pp. 157−173, USENIX Association, San Diego, CA, Septembre 1993.

[Chevalier 93b] P.Y. Chevalier et X. Rousset de Pina, ''A Reliable Storage Server for Distributed Cooperative Applications'', *Proceedings of the first open workshop of the BROADCAST BRA Project*, Newcastle upon Tyne, Octobre 1993.

[Chevalier 94a] P.Y. Chevalier, M. Riveill et F. Saunier, ‘‘Towards a Generic System Support for Co−operative Applications’’, *Proceedings of the 3rd Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, (à paraître), IEEE Computer Society Press, MorganTown, WV, Avril 1994.

[Chevalier 94b] P.Y. Chevalier, A. Freyssinet, D. Hagimont, S. Krakowiak, S. Lacourte, J. Mossière et X. Rousset de Pina, ‘‘Persistent Shared Object Support in the Guide System: Evaluation & Related Work’’, *Accepté à OOPSLA’94*, Portland, OG, Octobre 1994.

[Chevalier 94c] P.Y. Chevalier et X. Rousset de Pina, *Separating persistence from sharing in distributed persistent stores*, (BI−TR−94−27), *Soumis à publication*, Unité Mixte Bull−IMAG/Systèmes, Gières, France, Mai 1994.

[Cheung 92] S. Y. Cheung, M. H. Ammar et M. Ahamad, ‘‘The Grid Protocol: A High Performance Scheme for Maintaining Replicated Data’’, *IEEE Transactions on Knowledge and Data Engineering*, 4(6), pp. 582−592, Décembre 1992.

[Cockshot 84] W. P. Cockshot, M. P. Atkinson, K. J. Chisholm, P. J. Bailey et R. Morrison, ‘‘Persistent Object Management System’’, *Software− Practice and Experience*, Vol. 14, pp. 49−71, 1984.

[Cristian 91] F. Cristian, ‘‘Understanding Fault−Tolerant Distributed Systems’’, *Communications of the ACM*, 32(2), pp. 56−78, Février 1991.

[Dasgupta 90] P. Dasgupta, R.C. Chen, S. Menon, M.P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R.J. LeBlanc, W.F. Appelbe, J.M. Bernabeu−Auban, P.W. Hutto, M.Y.A. Khalidi et C.J. Wilkenloh, ‘‘The Design and Implementation of the Clouds Distributed Operating System’’, *Computing Systems*, 3(1), pp. 11−45, Hiver 1990.

[Davidson 89] S.B. Davidson, ‘‘Replicated Data and Partition Failures’’, *In Distribued Systems*, édité par S. Mullender, pp. 265−292, ACM Press − Frontier Series, 1989.

[Day 92] M. Day, B. Liskov, U. Maheshwari et A. C. Myers, *Naming and Locating Objects in Thor*, Laboratory of Computer Science, MIT, 1992.

[Dechamboux 93] P. Dechamboux, *Gestion d’objets persistants : du langage de programmation au système*, Thèse de doctorat, Université Joseph Fourier − Grenoble I, Février 1993.

[Decouchant 93] D. Decouchant, V. Quint, M. Riveill et I. Vatton, *Griffon: A Cooperative, Structured, Distributed Document Editor*, (BI−TR), Bull−IMAG/Systèmes, Gières, France, Mai 1993.

[Deppish 91] U. Deppisch, H.−B. Paul, H.−J. Schek et G. Weikum, ‘‘Managing Complex Objects in the Darmstadt Database Kernel System’’, *On Object−Oriented Database Systems*, pp. 357−375, Springer−Verlag, Berlin Heidelberg, 1991.

[Deux 91] O. Deux et al., ''The O_2 System'', *Communications of the ACM*, 34(10), pp. 35−48, Octobre 1991.

[DeWitt 90] D. DeWitt, P. Futtersack, D. Maier et F. Vélez, ''A study of three alternative workstation/server architecture for object−oriented database systems'', *Proceedings of the 16th VLDB Conference*, édité par D. McLeod, R. Sachs−Davis et H. Shek, pp. 107−121, Morgan Kaufmann Publishers, Brisbane, Australia, Août 1990.

[Duda 93] J. Cayuela et A. Duda, ''System Management in the Guide Distributed System'', *Proceedings of the first International Workshop on System Management*, 1993.

[El Abbadi 85] A. El Abbadi, D. Skeen et F. Cristian, ''An Efficient, Fault−Tolerant Protocol for Replicated Data Management'', *Proceedings of the 4th ACM SIGACT−SIGMOD Symposium on Principles of Database Systems*, pp. 215−228, ACM Press, Portland, Mars 1985.

[Freyssinet 91] A. Freyssinet, *Architecture et Réalisation d'un Système Réparti à Objets*, Thèse de doctorat, Université Joseph Fourier − Grenoble I, Juillet 1991.

[Gray 93] J. Gray et A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[Golding 91] R. A. Golding, *Accessing replicated data in a large−scale distributed system*, Master of Science in Computer and Information Sciences, University of California, Santa Cruz (U.C.S.C.), Juin 1991.

[Golding 92] R. A. Golding et D. D. E. Long, *The Performance of Weak−consistency Replication Protocols*, (UCSC−CRL−92−30), University of California, Santa Cruz, Santa Cruz, CA, Juillet 1992.

[Golub 91] D. B. Golub et R. P. Draves, ''Moving the Default Memory Manager out of the Mach Kernel'', *Proceedings of the USENIX Mach Symposium*, pp. 177−188, USENIX Association, Monterey, CA, Novembre 1991.

[Grimsrud 93] K. S. Grimsrud, J. K. Archibald et B. E. Nelson, ''Multiple Prefetch Adaptive Disk Caching'', *IEEE TRansactions on Knowledge and Data Engineering*, 5(1), pp. 88−103, Février 1993.

[Grønbæk 94] K. Grønbæk, J. A. Hem, O. L. Madsen et L. Sloth, ''Cooperative Hypermedia Systems: A Dexter−Based Architecture'', *Communications of the ACM*, 37(2), pp. 64−74, Février 1994.

[Gruber 92] O. Gruber, L. Amsaleg, L. Daynès et P. Valduriez, ''Eos, an Environment for Object−based Systems'', *Proceedings of the 25th Hawaii International Conference on System Sciences*, volume 1, pp. 757−768, Janvier 1992.

[Hagimont 93] D. Hagimont, *Adressage et protection dans les systèmes répartis*, Thèse de docteur ingénieur, Institut National Polytechnique de Grenoble, Octobre 1993.

[Hagimont 94] D. Hagimont, ‘‘Protection in the Guide object−oriented distributed system’’, *Accepté à ECOOP’94*, Bologne, Juillet 1994.

[Heidemann 92] J. S. Heidemann, T. W. Page, R. G. Guy et G. J. Popek, ‘‘Primarily Disconnected Operation: Experiences With Ficus’’, *Proceedings of the 2 Workshop on Management of Replicated Data*, pp. 2−5, Monterey, CA, Novembre 1992.

[Heiser 93] G. Heiser, K. Elphinstone, S. Russell et J. Vochteloo, *Mungi: A Distributed Single Adress−Space Operating System*, (SCS&E Report 9314), School of Computer Science and Engineering, The University of New South Wales, Novembre 1993.

[Hosking 93] A. L. Hosking et J. E. B. Moss, ‘‘Object Fault Handling for Persistent Programming Languages: A Performance Evaluation’’, *Proceedings of the ACM conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA’93)*, Washington, DC, édité par ACM SIGPLAN Notices, 28(10), pp. 288−303, Octobre 1993.

[Jajodia 89] S. Jajodia et D. Mutchler, ‘‘A Pessimitic Consistency Control Algorithm for Replicated Files which Achieves High Availability’’, *IEEE Transactions on Software Engineering*, 15(1), pp. 39−45, Janvier 1989.

[Jajodia 90] S. Jajodia et D. Mutchler, ‘‘Dynamic Voting Algorithms for Maintaining the Consistency of Replicated Database’’, *ACM Transactions on Database Systems*, 15(2), pp. 230−280, Juin 1990.

[Joseph 89] T. A. Joseph et K.P. Birman, ‘‘Reliable Broadcast Protocols’’, *In Distributed Systems*, édité par S. Mullender, pp. 293−317, ACM Press − Frontier Series, 1989.

[Kaashoek 91] M. F. Kaashoek et A. S. Tanenbaum, ‘‘Group Communication in the Amoeba Distributed Operating System’’, *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS)*, pp. 222−230, IEEE Computer Society Press, Los Alamitos , CA, Mai 1991.

[Kato 93] K. Kato, A. Narita, S. Inohara et T. Masuda, ‘‘Distributed Shared Repository: A Unified Approach to Distribution and Persistency’’, *Proceedings of the 13th International Conference on Distributed Computing Systems (ICDCS)*, pp. 20−29, IEEE Computer Society Press, Pittsburgh, PA, Mai 1993.

[Khalidi 93] Y. A. Khalidi et M. N. Nelson, ‘‘An Implementation of UNIX on an Object−Oriented Operating System’’, *Proceedings of the 1993 Winter USENIX Conference*, pp. 469−480, USENIX Association, Janvier 1993.

[Kistler 92] J.J. Kistler et M. Satyanarayanan, ‘‘Disconnected Operation in the Coda File System’’, *ACM Transactions on Computer Systems*, 10(1), Février 1992.

[Ladin 91] R. Ladin, B. Liskov et L. Shrira, ‘‘Lazy replication: exploiting the semantics of distributed services’’, *Position paper for the 4th ACM−SIGOPS European Workshop (Bologne, Septembre 1990) − ACM Operating System Review* , 25(1), pp. 49−55,

Janvier 1991.

[Lamport 78] L. Lamport, ‘‘Time, clocks, and the ordering of events in a distributed system’’, *Communications of the ACM*, 21(7), pp. 58−65, 1978.

[Laprie 89] J. C Laprie., ‘‘Sûreté de fonctionnement des systèmes informatiques et tolérance aux fautes’’, *Les Techniques de l’Ingénieur*, H.4.450 pp. 1−12, Septembre 1989.

[Layaida 93] N. Layaida, *Schéma de Désignation Extensible dans un Système Réparti*, DEA d’Informatique, Institut National Polytechnique de Grenoble et Université Joseph Fourier − Grenoble I, Juin 1993.

[Leach 83] P.J. Leach, P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson et B.L. Stumpf, ‘‘The Architecture of an Integrated Local Network’’, *IEEE Journal on Selected Areas in Communication*, SAC−1(5), pp. 842−856, Novembre 1983.

[Legatheaux 88] J. Legatheaux Martins et Y. Berbers, ‘‘La désignation dans les systèmes d’exploitation répartis’’, *Technique et Science Informatique*, 7(4), pp. 359−372, 1988.

[Leslie 93] I. M. Leslie, D. McAuley et S. J. Mullender, ‘‘Pegasus − Operating System Support for Distributed Multimedia Systems’’, *ACM Operating System Review*, 25(5), pp. 69−78, Janvier 1993.

[Li 89] K. Li et P. Hudak, ‘‘Memory Systems’’, *ACM Transactions on Computer Systems*, 7(4), pp. 321−359, Novembre 1989.

[Little 90] M.C. Little et S.K. Shrivastava, ‘‘Replicated K−ResilientObject in Arjuna’’, *Proceedings of the Workshop on Management of Replicated Data*, édité par L.F. Cabrera et J.−F. Pâris, pp. 53−58, IEEE Computer Society Press, Los Alamitos, CA, Novembre 1990.

[Liskov 85] B. H. Liskov, *The Argus language and system*, Distributed Systems: Methods and Tools for Specification, Lecture Notes in Computer Science, Springer−Verlag, Berlin Heidleberg, pp. 343−430, 1985.

[Liskov 91] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira et M. Williams, ‘‘Replication in the Harp File System’’, *Operating System Review*, 25(5), pp. 226−238, 1991.

[Long 90] D. Long, *The Management of Replication in a Distributed System*, (UCSL−CRL−90−35), University of California, Santa Cruz (U.C.S.C.), Juilllet 1990.

[Lopere 92a] K. Loepere, *Mach Kernel Principles*, Open Software Foundation (OSF), Août 1992.

[Lopere 92b] K. Loepere, *Mach Kernel Interface*, Open Software Foundation (OSF), Août 1992.

[Lu 94] Q. Lu et M. Satyanarayanan, ‘‘Isolation−Only Transactions for Mobile

Computing'', *ACM Operating Systems Review*, 28(2), pp. 81−87, Avril 1994.

[Mann 89] T. Mann, A. Hisgen and G. Swart, *An Algorithm for Data replication*, (TR−46), Digital System Research Center, Palo Alto, CA, Juin 1989.

[McNamee 90] D. McNamee et K. Armstrong, ''Extending the Mach External Pager Interface to Accomodate User−Level Page Replacement Policies'', *Proceedings of the USENIX Mach workshop*, pp. 31−42, USENIX Association, Burlington, VE, Octobre 1990.

[Millard 93] B. R. Millard, P. Dasgupta, S. Rao et R. Kuramkote, ''Run−Time Support and Storage Management for Memory−Mapped Persistent Objects'', *Proceedings of the 13th International Conference on Distributed Computing Systems (ICDCS)*, pp. 508−515, IEEE Computer Society Press, Pittsburgh, PA, Mai 1993.

[Nunamaker 91] J.F. Nunamaker, A. R. Dennis, J. S. Valacich, D. R. Vogel et J. F. George, ''Electronic Meeting Systems to Support Group Work'', *Communications of the ACM*, 34(7), pp. 40−61, Juillet 1991.

[Organick 72] E.I. Organick, *The Multics system: an examination of its structure*, MIT Press, 1972.

[Pâris 92] J.−F. Pâris et P. K. Sloope, ''Dynamic Management of Highly Replicated Data'', *Proceedings of the 11th Symposium on Reliable Distributed Systems (S.R.D.S.)*, pp. 20−27, IEEE Computer Society Press, Houston, TX, Octobre 1992.

[Perret 93] S. Perret, *Algorithmes de partage d'objets dans un systeme réparti*, DEA Informatique, Institut National Polytechnique de Grenoble et Université Joseph Fourier − Grenoble I, Juin 1993.

[Pitts 87] D. V. Pitts et P. Dasgupta, *Object Memory and Storage Management in the Clouds Kernel*, (TR GIT−ICS−87/15), School of Information and Computer Science − Georgia Institute of Technology, Atlanta, GA, 1987.

[Pu 91] C. Pu et A. Leff, ''Replica Control in Distributed Systems: An Asynchronous Approach'', *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, édité par J. Clifford et R. King, pp. 377−386, ACM Press, Denver, CO, Mai 1991.

[Rabinovitch 92] M. Rabinovitch et E. D. Lazowska, ''Improving Fault−Tolerance and Supporting Partial Writes in Structured Coterie Protocols for Replicated Objects'', *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, édité par M. Stonebraker, pp. 226−235, ACM Press, San Diego, CA, Juin 1992.

[Rosenblum 91] M. Rosenblum et J. K. Ousterhout, ''The Design and Implementation of a Log−Structured File System'', *Proceedings of the 13th ACM SOSP*, pp. 1−15, ACM Press, Pacific Grove, CA, Octobre 1991.

[Rousset 93] X. Rousset de Pina, *Spécification et Construction de Systèmes Opératoires*,

Habilitation à diriger des recherches, Université Joseph Fourier, Janvier 1993.

[Rozier 88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, P. Léonard, S. Langlois et W. Neuhauser, ‘‘Chorus Distributed Operating Systems’’, *Computing Systems*, 1(4), pp. 305−370, 1988.

[Satyanarayanan 90] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel et D.C. Steere, ‘‘Coda: A Highly Available File System for a Distributed Workstation Environment’’, *IEEE Transactions on Computers*, 39(4), Avril 1990.

[Satyanarayanan 93] M. Satyanarayanan, J.J. Kistler, L.B. Mummert, M.R. Ebling, P. Kumar et Q. Lu, *Experience with Disconnected Operation in a Mobile Computing Environment*, (CMU−CS−93−168), Carnegie Mellon University, Pittsburgh, PA, Juin 1993.

[Schmidt 77] J. W. Schmidt, ‘‘Some high level language constructs for data of type relation’’, *ACM Transactions on Database Systems*, 2(3), pp. 247−261, Septembre 1977.

[Scioville 84] R. Scioville−Garcia, *Gestion des Informations Persistantes dans un Système Réparti à Objets*, Thèse de doctorat, Univeristé Joseph Fourier − Grenoble I, Juin 1984.

[Shapiro 92] M. Shapiro, P. Dickman et D. Plainfossé, *SSP Chains: Robust, Distributed References Supporting Acyclic Garbage Collection*, (RT−1799), INRIA − Rocquencourt, Le Chesney, France, Novembre 1992.

[Shekita 90] E. Shekita et M. Zwilling, ‘‘Cricket: A Mapped Persistent Object Store’’, *Proceedings of the 4th International Workshop on Persistent Systems*, édité par A. Dearle, G. M. Shaw et S. B. Zdonik, pp. 89−102, Martha’s Vineyard, MA, Septembre 1990.

[Shrivastava 92] S. K. Shrivastava, P. D. Ezhilchelvan, N. A. Speirs, S. Tao et A. Tully, ‘‘Principal Features of the VOLTAN Family of Reliable Node Architectures for Distributed Systems’’, *IEEE Transactions on Computers*, 41(5), pp. 542−549, Mai 1992.

[Song 93] I. Song et Y. Cho, ‘‘Page Prefetching Based on Fault History’’, *Proceedings of the USENIX Mach III Symposium*, pp. 203−213, USENIX Association, Santa−Fé, NM, Avril 1993.

[Staehli 93] R. Staehli et J. Walpole, ‘‘Constrained−Latency Storage Access’’, *IEEE Computer*, 26(3), pp. 44−53, Mars 1993.

[Stonebraker 81] M. Stonebraker, ‘‘Operating System Support for Database Management’’, *Communications of the ACM*, 24(7), pp. 412−418, Juillet 1981.

[Thatte 86] S. Thatte, ‘‘PERSISTENT MEMORY: A Storage Architecture for Object−Oriented Database Systems’’, *Proceedings of ACM/IEEE 1986 International*

Workshop on Object Oriented Databse Systems, pp. 148–159, ACM Press, Pacific Grove, CA, Septembre 1986.

[Thomsen 93] B. Thomsen, L. Leth, S. Prasad, T.–M. Kuo, A. Kramer, F. Knabe et A. Giacalone, *Facile Antigua Release Programming Guide*, (ECRC–93–20), European Computer–Industry Research Center, Décembre 1993.

[Traw 93] C.B.S. Traw et J. M. Smith, ''Hardware/Software Organization of a High Performance ATM Host Interface'', *Journal on Selected Areas in Communications*, Février 1993.

[Vaughan 92] F. Vaughan, T. L. Basso, A. Dearle, C. Marlin et C. Barter, ''Casper: a Cached Architecture Supporting Persistence'', *Computing Systems*, 5(3), pp. 337–359, Eté 1992.

[Wu 92] C. Wu et G. G. Belford, ''The Triangular Lattice Protocol: A Highly Fault Tolerant and Higly Efficient Protocol for Replicated Data'', *Proceedings of the 11th Symposium on Reliable Distributed Systems (S.R.D.S.)*, pp. 66–73, IEEE Computer Society Press, Houston, TX, Octobre 1992.