Numéro d'ordre: 1924

USTL

Année : **1997**

LIFL Laboratoire d'Informatique Fondamentale de Lille



# THESE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Cedric Dumoulin

# Dream : Une mémoire partagée répartie à cohérence programmable

Thèse soutenue le 9 janvier 1997, devant la commission d'examen :

| | | |
|---|---|---|
| Président: | J.L. DEKEYSER | LIFL |
| Rapporteurs: | H. GUYENNET | LIB |
| | G. LIBERT | PIP |
| Examinateurs: | | |
| | I. DEMEURE | ENST |
| | J-M. GEIB | LIFL |
| | J-F. MEHAUT | LIFL |
| | J-F. ROOS | LIFL |

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
U.F.R. d'I.E.E.A. Bât M3. 59655 Villeneuve d'Ascq CEDEX
Tél. 03. 20.43.47.24 Fax. 03.20.43.65.66

UNIVERSITE DES SCIENCES
ET TECHNOLOGIES DE LILLE

---

## DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M. H. LEFEBVRE, M. PARREAU

## PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

## PROFESSEUR EMERITE

M. A. LEBRUN

## ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PARREAU, J. LOMBARD, M. MIGEON, J. CORTOIS, A.DUBRULLE

## PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

M. P. LOUIS

## PROFESSEURS - CLASSE EXCEPTIONNELLE

| | |
|---|---|
| M. CHAMLEY Hervé | Géotechnique |
| M. CONSTANT Eugène | Electronique |
| M. ESCAIG Bertrand | Physique du solide |
| M. FOURET René | Physique du solide |
| M. GABILLARD Robert | Electronique |
| M. LABLACHE COMBIER Alain | Chimie |
| M. LOMBARD Jacques | Sociologie |
| M. MACKE Bruno | Physique moléculaire et rayonnements atmosphériques |

| | |
|---|---|
| M. TURREL Georges | Spectrochimie infrarouge et raman |
| M. VANDIJK Hendrik | |
| Mme VAN ISEGHEM Jeanine | Modélisation,calcul scientifique,statistiques |
| M. VANDORPE Bernard | Chimie minérale |
| M. VASSEUR Christian | Automatique |
| M. VASSEUR Jacques | Biologie |
| Mme VIANO Marie Claude | |
| M. WACRENIER Jean Marie | Electronique |
| M. WARTEL Michel | Chimie inorganique |
| M. WATERLOT Michel | géologie générale |
| M. WEICHERT Dieter | Génie mécanique |
| M. WERNER Georges | Informatique théorique |
| M. WIGNACOURT Jean Pierre | |
| M. WOZNIAK Michel | Spectrochimie |
| Mme ZINN JUSTIN Nicole | Algèbre |

| | |
|---|---|
| M. MIGEON Michel | EUDIL |
| M. MONTREUIL Jean | Biochimie |
| M. PARREAU Michel | Analyse |
| M. TRIDOT Gabriel | Chimie appliquée |

## PROFESSEURS - 1ère CLASSE

| | |
|---|---|
| M. BACCHUS Pierre | Astronomie |
| M. BIAYS Pierre | Géographie |
| M. BILLARD Jean | Physique du Solide |
| M. BOILLY Bénoni | Biologie |
| M. BONNELLE Jean Pierre | Chimie-Physique |
| M. BOSCQ Denis | Probabilités |
| M. BOUGHON Pierre | Algèbre |
| M. BOURIQUET Robert | Biologie Végétale |
| M. BRASSELET Jean Paul | Géométrie et topologie |
| M. BREZINSKI Claude | Analyse numérique |
| M. BRIDOUX Michel | Chimie Physique |
| M. BRUYELLE Pierre | Géographie |
| M. CARREZ Christian | Informatique |
| M. CELET Paul | Géologie générale |
| M. COEURE Gérard | Analyse |
| M. CORDONNIER Vincent | Informatique |
| M. CROSNIER Yves | Electronique |
| Mme DACHARRY Monique | Géographie |
| M. DAUCHET Max | Informatique |
| M. DEBOURSE Jean Pierre | Gestion des entreprises |
| M. DEBRABANT Pierre | Géologie appliquée |
| M. DECLERCQ Roger | Sciences de gestion |
| M. DEGAUQUE Pierre | Electronique |
| M. DESCHEPPER Joseph | Sciences de gestion |
| Mme DESSAUX Odile | Spectroscopie de la réactivité chimique |
| M. DHAINAUT André | Biologie animale |
| Mme DHAINAUT Nicole | Biologie animale |
| M. DJAFARI Rouhani | Physique |
| M. DORMARD Serge | Sciences Economiques |
| M. DOUKHAN Jean Claude | Physique du solide |
| M. DUBRULLE Alain | Spectroscopie hertzienne |
| M. DUPOUY Jean Paul | Biologie |
| M. DYMENT Arthur | Mécanique |
| M. FOCT Jacques Jacques | Métallurgie |
| M. FOUQUART Yves | Optique atmosphérique |
| M. FOURNET Bernard | Biochimie structurale |
| M. FRONTIER Serge | Ecologie numérique |
| M. GLORIEUX Pierre | Physique moléculaire et rayonnements atmosphériques |
| M. GOSSELIN Gabriel | Sociologie |
| M. GOUDMAND Pierre | Chimie-Physique |
| M. GRANELLE Jean Jacques | Sciences Economiques |
| M. GRUSON Laurent | Algèbre |
| M. GUILBAULT Pierre | Physiologie animale |
| M. GUILLAUME Jean | Microbiologie |
| M. HECTOR Joseph | Géométrie |
| M. HENRY Jean Pierre | Génie mécanique |
| M. HERMAN Maurice | Physique spatiale |
| M. LACOSTE Louis | Biologie Végétale |
| M. LANGRAND Claude | Probabilités et statistiques |

| | |
|---|---|
| M. LATTEUX Michel | Informatique |
| M. LAVEINE Jean Pierre | Paléontologie |
| Mme LECLERCQ Ginette | Catalyse |
| M. LEHMANN Daniel | Géométrie |
| Mme LENOBLE Jacqueline | Physique atomique et moléculaire |
| M. LEROY Jean Marie | Spectrochimie |
| M. LHENAFF René | Géographie |
| M. LHOMME Jean | Chimie organique biologique |
| M. LOUAGE Françis | Electronique |
| M. LOUCHEUX Claude | Chimie-Physique |
| M. LUCQUIN Michel | Chimie physique |
| M. MAILLET Pierre | Sciences Economiques |
| M. MAROUF Nadir | Sociologie |
| M. MICHEAU Pierre | Mécanique des fluides |
| M. PAQUET Jacques | Géologie générale |
| M. PASZKOWSKI Stéfan | Mathématiques |
| M. PETIT Francis | Chimie organique |
| M. PORCHET Maurice | Biologie animale |
| M. POUZET Pierre | Modélisation - calcul scientifique |
| M. POVY Lucien | Automatique |
| M. PROUVOST Jean | Minéralogie |
| M. RACZY Ladislas | Electronique |
| M. RAMAN Jean Pierre | Sciences de gestion |
| M. SALMER Georges | Electronique |
| M. SCHAMPS Joël | Spectroscopie moléculaire |
| Mme SCHWARZBACH Yvette | Géométrie |
| M. SEGUIER Guy | Electrotechnique |
| M. SIMON Michel | Sociologie |
| M. SLIWA Henri | Chimie organique |
| M. SOMME Jean | Géographie |
| Melle SPIK Geneviève | Biochimie |
| M. STANKIEWICZ François | Sciences Economiques |
| M. THIEBAULT François | Sciences de la Terre |
| M. THOMAS Jean Claude | Géométrie - Topologie |
| M. THUMERELLE Pierre | Démographie - Géographie humaine |
| M. TILLIEU Jacques | Physique théorique |
| M. TOULOTTE Jean Marc | Automatique |
| M. TREANTON Jean René | Sociologie du travail |
| M. TURRELL Georges | Spectrochimie infrarouge et raman |
| M. VANEECLOO Nicolas | Sciences Economiques |
| M. VAST Pierre | Chimie inorganique |
| M. VERBERT André | Biochimie |
| M. VERNET Philippe | Génétique |
| M. VIDAL Pierre | Automatique |
| M. WALLART Françis | Spectrochimie infrarouge et raman |
| M. WEINSTEIN Olivier | Analyse économique de la recherche et développement |
| M. ZEYTOUNIAN Radyadour | Mécanique |

## PROFESSEURS - 2ème CLASSE

| | |
|---|---|
| M. ABRAHAM Francis | |
| M. ALLAMANDO Etienne | Composants électroniques |
| M. ANDRIES Jean Claude | Biologie des organismes |
| M. ANTOINE Philippe | Analyse |
| M. BALL Steven | Génétique |
| M. BART André | Biologie animale |
| M. BASSERY Louis | Génie des procédés et réactions chimiques |
| Mme BATTIAU Yvonne | Géographie |
| M. BAUSIERE Robert | Systèmes électroniques |
| M. BEGUIN Paul | Mécanique |
| M. BELLET Jean | Physique atomique et moléculaire |
| M. BERNAGE Pascal | Physique atomique, moléculaire et du rayonnement |
| M. BERTHOUD Arnaud | Sciences Economiques |
| M. BERTRAND Hugues | Sciences Economiques |
| M. BERZIN Robert | Analyse |
| M. BISKUPSKI Gérard | Physique de l'état condensé et cristallographie |
| M. BKOUCHE Rudolphe | Algèbre |
| M. BODARD Marcel | Biologie végétale |
| M. BOHIN Jean Pierre | Biochimie métabolique et cellulaire |
| M. BOIS Pierre | Mécanique |
| M. BOISSIER Daniel | Génie civil |
| M. BOIVIN Jean Claude | Spectrochimie |
| M. BOUCHER Daniel | Physique |
| M. BOUQUELET Stéphane | Biologie appliquée aux enzymes |
| M. BOUQUIN Henri | Gestion |
| M. BROCARD Jacques | Chimie |
| Mme BROUSMICHE Claudine | Paléontologie |
| M. BUISINE Daniel | Mécanique |
| M. CAPURON Alfred | Biologie animale |
| M. CARRE François | Géographie humaine |
| M. CATTEAU Jean Pierre | Chimie organique |
| M. CAYATTE Jean Louis | Sciences Economiques |
| M. CHAPOTON Alain | Electronique |
| M. CHARET Pierre | Biochimie structurale |
| M. CHIVE Maurice | Composants électroniques optiques |
| M. COMYN Gérard | Informatique théorique |
| Mme CONSTANT Monique | Composants électroniques et optiques |
| M. COQUERY Jean Marie | Psychophysiologie |
| M. CORIAT Benjamin | Sciences Economiques |
| Mme CORSIN Paule | Paléontologie |
| M. CORTOIS Jean | Physique nucléaire et corpusculaire |
| M. COUTURIER Daniel | Chimie organique |
| M. CRAMPON Norbert | Tectolique géodynamique |
| M. CURGY Jean Jacques | Biologie |
| M. DANGOISSE Didier | Physique théorique |
| M. DE PARIS Jean Claude | Analyse |
| M. DECOSTER Didier | Composants électroniques et optiques |
| M. DEJAEGER Roger | Electrochimie et Cinétique |
| M. DELAHAYE Jean Paul | Informatique |
| M. DELORME Pierre | Physiologie animale |
| M. DELORME Robert | Sciences Economiques |
| M. DEMUNTER Paul | Sociologie |
| Mme DEMUYNCK Claire | Physique atomique, moléculaire et du rayonnement |
| M. DENEL Jacques | Informatique |
| M. DEPREZ Gilbert | Physique du solide - christallographie |

| | |
|---|---|
| M. DERIEUX Jean Claude | Microbiologie |
| M. DERYCKE Alain | Informatique |
| M. DESCAMPS Marc | Physique de l'état condensé et cristallographie |
| M. DEVRAINNE Pierre | Chimie minérale |
| M. DEWAILLY Jean Michel | Géographie humaine |
| M. DHAMELINCOURT Paul | Chimie physique |
| M. DI PERSIO Jean | Physique de l'état condensé et cristallographie |
| M. DUBAR Claude | Sociologie démographique |
| M. DUBOIS Henri | Spectroscopie hertzienne |
| M. DUBOIS Jean Jacques | Géographie |
| M. DUBUS Jean Paul | Spectrométrie des solides |
| M. DUPONT Christophe | Vie de la firme |
| M. DUTHOIT Bruno | Génie civil |
| Mme DUVAL Anne | Algèbre |
| Mme EVRARD Micheline | Génie des procédés et réactions chimiques |
| M. FAKIR Sabah | Algèbre |
| M. FARVACQUE Jean Louis | Physique de l'état condensé et cristallographie |
| M. FAUQUEMBERGUE Renaud | Composants électroniques |
| M. FELIX Yves | Mathématiques |
| M. FERRIERE Jacky | Tectonique - Géodynamique |
| M. FISCHER Jean Claude | Chimie organique, minérale et analytique |
| M. FONTAINE Hubert | Dynamique des cristaux |
| M. FORSE Michel | Sociologie |
| M. GADREY Jean | Sciences économiques |
| M. GAMBLIN André | Géographie urbaine, industrielle et démographie |
| M. GOBLOT Rémi | Algèbre |
| M. GOURIEROUX Christian | Probabilités et statistiques |
| M. GREGORY Pierre | I.A.E. |
| M. GREMY Jean Paul | Sociologie |
| M. GREVET Patrice | Sciences Economiques |
| M. GRIMBLOT Jean | Chimie organique |
| M. GUELTON Michel | Chimie physique |
| M. GUICHAOUA André | Sociologie |
| M. HAIMAN Georges | Modélisation,calcul scientifique, statistiques |
| M. HOUDART René | Physique atomique |
| M. HUEBSCHMANN Johannes | Mathématiques |
| M. HUTTNER Marc | Algèbre |
| M. ISAERT Noël | Physique de l'état condensé et cristallographie |
| M. JACOB Gérard | Informatique |
| M. JACOB Pierre | Probabilités et statistiques |
| M. JEAN Raymond | Biologie des populations végétales |
| M. JOFFRE Patrick | Vie de la firme |
| M. JOURNEL Gérard | Spectroscopie hertzienne |
| M. KOENIG Gérard | Sciences de gestion |
| M. KOSTRUBIEC Benjamin | Géographie |
| M. KREMBEL Jean | Biochimie |
| Mme KRIFA Hadjila | Sciences Economiques |
| M. LANGEVIN Michel | Algèbre |
| M. LASSALLE Bernard | Embryologie et biologie de la différenciation |
| M. LE MEHAUTE Alain | Modélisation,calcul scientifique,statistiques |
| M. LEBFEVRE Yannic | Physique atomique,moléculaire et du rayonnement |
| M. LECLERCQ Lucien | Chimie physique |
| M. LEFEBVRE Jacques | Physique |
| M. LEFEBVRE Marc | Composants électroniques et optiques |
| M. LEFEVRE Christian | Pétrologie |
| Melle LEGRAND Denise | Algèbre |
| M. LEGRAND Michel | Astronomie - Météorologie |
| M. LEGRAND Pierre | Chimie |
| Mme LEGRAND Solange | Algèbre |
| Mme LEHMANN Josiane | Analyse |
| M. LEMAIRE Jean | Spectroscopie hertzienne |

| | |
|---|---|
| M. LE MAROIS Henri | Vie de la firme |
| M. LEMOINE Yves | Biologie et physiologie végétales |
| M. LESCURE François | Algèbre |
| M. LESENNE Jacques | Systèmes électroniques |
| M. LOCQUENEUX Robert | Physique théorique |
| Mme LOPES Maria | Mathématiques |
| M. LOSFELD Joseph | Informatique |
| M. LOUAGE Francis | Electronique |
| M. MAHIEU François | Sciences économiques |
| M. MAHIEU Jean Marie | Optique - Physique atomique |
| M. MAIZIERES Christian | Automatique |
| M. MANSY Jean Louis | Géologie |
| M. MAURISSON Patrick | Sciences Economiques |
| M. MERIAUX Michel | EUDIL |
| M. MERLIN Jean Claude | Chimie |
| M. MESMACQUE Gérard | Génie mécanique |
| M. MESSELYN Jean | Physique atomique et moléculaire |
| M. MOCHE Raymond | Modélisation,calcul scientifique,statistiques |
| M. MONTEL Marc | Physique du solide |
| M. MORCELLET Michel | Chimie organique |
| M. MORE Marcel | Physique de l'état condensé et cristallographie |
| M. MORTREUX André | Chimie organique |
| Mme MOUNIER Yvonne | Physiologie des structures contractiles |
| M. NIAY Pierre | Physique atomique,moléculaire et du rayonnement |
| M. NICOLE Jacques | Spectrochimie |
| M. NOTELET Francis | Systèmes électroniques |
| M. PALAVIT Gérard | Génie chimique |
| M. PARSY Fernand | Mécanique |
| M. PECQUE Marcel | Chimie organique |
| M. PERROT Pierre | Chimie appliquée |
| M. PERTUZON Emile | Physiologie animale |
| M. PETIT Daniel | Biologie des populations et écosystèmes |
| M. PLIHON Dominique | Sciences Economiques |
| M. PONSOLLE Louis | Chimie physique |
| M. POSTAIRE Jack | Informatique industrielle |
| M. RAMBOUR Serge | Biologie |
| M. RENARD Jean Pierre | Géographie humaine |
| M. RENARD Philippe | Sciences de gestion |
| M. RICHARD Alain | Biologie animale |
| M. RIETSCH François | Physique des polymères |
| M. ROBINET Jean Claude | EUDIL |
| M. ROGALSKI Marc | Analyse |
| M. ROLLAND Paul | Composants électroniques et optiques |
| M. ROLLET Philippe | Sciences Economiques |
| Mme ROUSSEL Isabelle | Géographie physique |
| M. ROUSSIGNOL Michel | Modélisation,calcul scientifique,statistiques |
| M. ROY Jean Claude | Psychophysiologie |
| M. SALERNO Françis | Sciences de gestion |
| M. SANCHOLLE Michel | Biologie et physiologie végétales |
| Mme SANDIG Anna Margarette | |
| M. SAWERYSYN Jean Pierre | Chimie physique |
| M. STAROSWIECKI Marcel | Informatique |
| M. STEEN Jean Pierre | Informatique |
| Mme STELLMACHER Irène | Astronomie - Météorologie |
| M. STERBOUL François | Informatique |
| M. TAILLIEZ Roger | Génie alimentaire |
| M. TANRE Daniel | Géométrie - Topologie |
| M. THERY Pierre | Systèmes électroniques |
| Mme TJOTTA Jacqueline | Mathématiques |
| M. TOURSEL Bernard | Informatique |
| M. TREANTON Jean René | Sociologie du travail |

| | |
|---|---|
| M. TURREL Georges | Spectrochimie infrarouge et raman |
| M. VANDIJK Hendrik | |
| Mme VAN ISEGHEM Jeanine | Modélisation,calcul scientifique,statistiques |
| M. VANDORPE Bernard | Chimie minérale |
| M. VASSEUR Christian | Automatique |
| M. VASSEUR Jacques | Biologie |
| Mme VIANO Marie Claude | |
| M. WACRENIER Jean Marie | Electronique |
| M. WARTEL Michel | Chimie inorganique |
| M. WATERLOT Michel | géologie générale |
| M. WEICHERT Dieter | Génie mécanique |
| M. WERNER Georges | Informatique théorique |
| M. WIGNACOURT Jean Pierre | |
| M. WOZNIAK Michel | Spectrochimie |
| Mme ZINN JUSTIN Nicole | Algèbre |

# *Table des Matières*

# *Liste des figures*

# Introduction

Ces dernières années ont vu un vaste déploiement des architectures et systèmes distribués dans le paysage informatique. Plus qu'un phénomène de mode, ces systèmes répondent à des besoins très précis :

- Partage des ressources. Le réseau est l'élément crucial permettant aux utilisateurs de partager des ressources lourdes et coûteuses, comme les imprimantes ou les unités de sauvegarde.
- Communication. Le monde d'aujourd'hui est un monde où les besoins en communication sont de plus en plus importants. Les réseaux et les systèmes distribués vont répondre à cette demande en proposant les outils pour communiquer (*talk*, *mail*), mais également des possibilités pour se partager des informations (système de fichier réparti, mémoires partagées, ...).
- Evolutivité. Les entreprises, et plus généralement les structures utilisant l'informatique sont en plein essor. Elles exigent des systèmes informatiques un maximum d'évolutivité en terme de poste de travail, de puissance de calcul, d'espace de stockage, d'intégration des nouvelles technologies,...

**Les systèmes distribués**

Les systèmes distribués apportent des solutions à ces besoins. Ils ont la particularité d'offrir, en plus de ces solutions, une puissance potentielle de calcul proportionnelle au nombre de machines connectées. Ils restent une bonne alternative aux machines parallèles au niveau de la puissance de calcul disponible.

Le développement d'applications, dans un contexte distribué, reste encore délicat de par la pauvreté des modèles de programmation pour systèmes distribués et suite au lourd héritage des applications développées dans un contexte centralisé.

**Programmation des systèmes distribués**

D'un point de vue opératoire, une application distribuée peut se définir comme un ensemble de processus s'exécutant sur différents sites communiquant entre eux par envois de messages. Pour faciliter la programmation de telles applications, et notamment la mise en oeuvre des communications entre processus, différents modèles de programmation ont été développés. A travers ces modèles, trois axes méthodologiques se dégagent :

- les processus communicants : les processus communiquent par échanges de messages circulant sur le réseau de communication. La communication est calquée sur l'architecture sous-jacente. Différentes implémentations de ce modèle sont disponibles sous diverses formes (PVM, MPI, CSP, OCCAM).
- les RPC (Remote Procedure Call). Ils étendent la notion d'appel procédural aux systèmes distribués. Un processus effectue une exécution distante (dans un autre processus) par appel d'un RPC. Cet appel est similaire à l'appel d'une procédure

locale.

- les Mémoires Partagées Réparties (MPR ou DSM Distributed Shared Memory). Les processus se partagent de l'information dans une mémoire commune fictive (sans réalité physique) appelée Mémoire Partagée Répartie.

Ce dernier axe fait l'objet de ce travail.

**Les mémoires partagées réparties**

Les mémoires partagées réparties donnent l'illusion d'avoir une mémoire commune à plusieurs sites (ou machines, ou processus) alors qu'il n'existe aucune mémoire physique entre ces sites. Les processus d'une application communiquent alors par l'intermédiaire de cette mémoire partagée.

Les mémoires partagées offrent trois avantages par rapport aux autres modèles de programmation des systèmes distribués:

- La programmation reste proche de la programmation classique d'un système centralisé. En effet, dans ce dernier, les processus communiquent par le biais d'une mémoire commune.
- La mise en commun des informations est naturelle : elle se fait dans la mémoire partagée. Les données communes sont placées dans la mémoire partagée et sont potentiellement accessibles de tous.
- La localisation des données dans la mémoire partagée et non dans les processus, évite la prise de connaissance du processus les stockant.

Mais, les DSM présentent aussi un inconvénient : les applications développées avec une mémoire partagée sont généralement moins performantes que lorsqu'elles sont réalisées avec le modèle des processus communicants ou les RPC.

Ce désavantage vient du fait que la DSM est construite sur une architecture où la coopération ne peut se faire qu'avec le réseau de communication. L'implémentation de la DSM passe alors obligatoirement par un mécanisme projetant la mémoire partagée en échanges de messages. Ce mécanisme introduit un délai supplémentaire par rapport à une solution écrite directement avec des échanges de messages.

Ce mécanisme doit aussi assurer, à chaque processus, la perception d'une vue cohérente de la mémoire partagée. Ainsi, une modification de cette mémoire doit généralement être perçue "dès que possible" par l'ensemble des processus de l'application.

Plusieurs modèles de DSM sont proposés, relâchant les contraintes imposées sur le comportement de la mémoire partagée. Schématiquement, dans ces modèles, des processus différents peuvent percevoir des états différents de la mémoire partagée, mais les différents états autorisés sont bien définis par ce que l'on appelle le modèle de cohérence.

Différents modèles de cohérence existent, mais tous proposent une cohérence que nous qualifions de "générale", c'est à dire une cohérence adaptée à toutes les situations possibles. Dans ces modèles, une modification de la mémoire partagée est perçue par l'ensemble des processus si les conditions imposées par la cohérence sont remplies.

Pourtant, chaque application a des besoins spécifiques en matière de cohérences. Par exemple, certaines applications peuvent fort bien fonctionner avec des états "un peu anciens" de la mémoire partagée. Ainsi, la mémoire partagée n'est pas contrainte de reproduire systématiquement les modifications, elle peut retarder leurs propagations. D'autres applications ont besoin d'une vue cohérente de la mémoire partagée uniquement en certains

points. Par ailleurs, elles peuvent, elles aussi, se contenter d'une vue "un peu ancienne".

D'autre part, les modèles de DSM basent la coopération inter-processus sur l'utilisation exclusive de la mémoire partagée. Dans certains cas, cet usage exclusif peut s'avérer moins efficace, en terme de simplicité de programmation, que de simples échanges de messages.

Prenons l'exemple de la synchronisation d'un processus par un autre. Avec une DSM, un processus positionne une variable partagée, signalant ainsi à l'autre qu'il peut continuer. Avec des messages, la synchronisation se fait par envoi et réception d'un message. La résolution est aussi simple dans un cas comme dans l'autre, mais la mémoire partagée introduit un délai supplémentaire car elle doit transformer les accès en échanges de messages.

**Dream**

Partant des constatations précédentes, nous proposons Dream (Distributed Region and Shared Memory).

Dream est un modèle de programmation hybride composé d'une mémoire partagée et de processus communicants. Ainsi les avantages des deux modèles sont disponibles. Cela permet de construire des applications mixant mémoire partagée et processus communicant.

Dream définit uniquement la mémoire partagée. Celle-ci est construite sur un modèle de processus communicants existant par ailleurs, modèle restant accessible lors de l'utilisation de Dream.

La mémoire partagée de Dream est structurée en régions de taille variable. Chaque région est une entité de partage individuelle destinée à accueillir un ou plusieurs objets à partager (variables, structures,...). Les objets partagés sont ainsi indépendants les uns des autres.

La cohérence proposée par Dream est une cohérence que nous qualifions de faible[1] : une modification de la mémoire partagée n'est pas toujours immédiatement perçue par l'ensemble des processus d'une application.

Cette cohérence est le comportement par défaut de Dream. Elle ne convient pas à toutes les applications. Dream propose alors de renforcer cette cohérence par un contrôle explicite lors du déroulement de l'application. La cohérence est donc programmable. Elle peut être modifiée durant l'application afin de l'adapter à différents problèmes.

Cette thèse est consacrée à l'étude et à la validation du modèle Dream.

**Cadre de travail**

Mon travail de thèse s'est déroulé au sein de l'équipe GOAL (Groupe Objet et Acteur de Lille) du LIFL (Laboratoire d'Informatique Fondamentale de Lille). Cette équipe fait partie de l'axe de recherche Informatique Parallèle et Distribuée (ParDis) et est dirigée par Jean-Marc Geib.

L'équipe GOAL est divisée en différents groupes de travail, dont le projet ESPACE (Execution Support for Parallel Application in High Performance Computing Environment) auquel je collabore. Ce projet est encadré par Jean-François Méhaut, et a pour objectif la conception et la réalisation d'un environnement de programmation à grain fin pour architectures distribuées.

Dans le cadre de ce projet, deux autres thèses sont en cours, une portant sur la réalisation d'un environement de programmation multithreadé ($PM^2$, Raymond Namyst) et une autre sur

1. La cohérence faible de *Dream* n'est pas la *weak consistency*. Faible est pris dans le sens opposé à forte. Malheureusement, *weak consistency* se traduit aussi par cohérence faible.

la résolution de problèmes irréguliers à l'aide d'un parallélisme massif de tâches par processus léger (Yves Denneulin).

**Plan de la thèse**

Cette thèse est décomposée en six chapitres complétés par des annexes.

Dans le premier chapitre nous présenterons les architectures et systèmes distribués, en insistant plus particulièrement sur la problèmatique du logiciel. Nous decrirons à la fois les modèles de programmation, les supports d'exécutions (PVM, MPI, RPC), mais aussi les outils d'aide au développement (régulateur de charge, dévermineur, ramasse-miettes).

Le second chapitre est un état de l'art sur les modèles de mémoire partagée. Nous introduirons le concept de cohérence et les différentes solutions qui ont déjà été proposée. Une caractéristique générale des approches est de retenir un niveau de cohérence assez fort, favorisant une sémantique forte du partage, au détriment des performances des applications.

Dans le chapitre trois nous développons notre proposition *Dream*. Nous proposons une classification de la cohérence à partir de différents critères relatifs à l'utilisation des objets partagés (constant, faiblement variable, fortement variable) qui vont nous permettre de définir des niveaux de cohérences. Dream propose à la base une cohérence dite faible, qui peut ensuite être renforcée explicitement.

Dans le chapitre quatre, nous abordons le contexte de développement ainsi que les éléments principaux d'implémentation. Nous décrirons précisément la représentation des régions et la stratégie de maintien de la cohérence.

Le chapitre cinq présente plusieurs applications développées avec Dream. L'objectif est ici de montrer que le modèle Dream facilite la conception d'applications nécessitant le partage d'objets (variables, structures), et que les performances de ces applications, avec le niveau de cohérence proposé, sont tout à fait interessants.

Une comparaison synthétique, sous forme de tableaux, de différents modèles de mémoires partagées est fournie en annexe, ainsi que le manuel complet de description de l'interface de programmation.

# 1 Programmation des Systèmes Distribués

Au milieu des années 1980, deux évolutions technologiques majeures en informatique ont été à l'origine d'une nouvelle ère : *l'informatique distribuée*. Ces évolutions ont porté sur :

1) Les recherches en micro-électronique sur les composants et leur intégration à grande échelle sur une petite surface (VLSI) ont permis l'émergence dans un premier temps du microprocesseur, ensuite des micro-ordinateurs et des stations de travail. On constate aujourd'hui que ces recherches sont toujours très actives et que les performances n'arrêtent pas de croître: taille des mots machine à 64 bits (processeurs ALPHA, HyperSparc), fréquences d'horloge qui dépassent 440MHZ, jeux d'instructions réduits (concept de processeur RISC).

2) Les réseaux locaux (LAN) qui sont apparus également dans les années 1980. Des architectures de réseaux extrêmement rustiques (bus pour *ethernet*, anneau à jeton pour *token ring*) ont été proposées et mises en place. Ces architectures présentent de nombreuses limitations en terme d'extensibilité. Il est en effet difficile de connecter un nombre important de sites. De nouveaux équipements (ponts filtrant, répéteurs) ont permis de construire des réseaux de taille plus importante. Des évolutions ont été proposées récemment comme FastEthernet, FDDI ou aussi ATM avec des performances de communication atteignant aujourd'hui les 600Mbits/s et on s'attend dans quelques années à atteindre le giga-bits/s.

Cette évolution matérielle rapide nécessite une évolution des logiciels afin de pouvoir exploiter pleinement les nouvelles possibilités offertes. Malheureusement, on constate que cette évolution du logiciel se fait plus lentement. Un des principaux problèmes vient du fait que les langages et les méthodologies ne sont pas encore complètement maîtrisées dans un contexte d'applications distribuée. Un des enjeux pour le succès de l'informatique distribuée reste le problème des logiciels et de leur développement.

Différents modèles de programmation font, ou ont fait, l'objet de recherches afin de faciliter le développement de logiciels pour des systèmes distribués. Parmi ces modèles nous étudierons le modèle des processus communicants, qui s'apparente à l'architecture des systèmes distribués, puis le modèle des RPC, étendant l'appel procédural aux systèmes distribués, et enfin les mémoires partagées réparties, donnant l'illusion d'une mémoire commune, alors qu'aucune mémoire physique commune n'existe.

Dans ce chapitre nous présenterons les systèmes distribués en insistant plus particulièrement sur les aspects "applications" et modèles de programmation.

Dans une première partie nous commencerons par donner une définition de ce qu'est un système distribué, aussi bien du point de vue matériel que du point de vue logiciel. Ceci nous amènera à évoquer les différents services de base fournis par un système distribué (exécution de processus, communication, synchronisation), ainsi que quelques logiciels faisant partie de l'environnement d'exécution, et facilitant la tâche du programmeur.

La seconde partie de ce chapitre sera consacrée à une présentation des principaux modèles de programmation pour architectures distribuées.

## 1.1 Introduction aux systèmes distribués

Nous allons définir ce qu'est un système distribué, et évoquer les différents services de base fournis.

### 1.1.1 Qu'est ce qu'un système distribué?

Plusieurs propositions ont été faites pour définir ce qu'est un système distribué. Le lecteur intéressé peut se reporter à celle de Silberschatz et Galvin [Silberschatz94] ou encore de Tanenbaum [Tanenbaum95]. Pour notre part, nous proposons une définition qui étend la définition d'un système centralisé.

**Définitions**

Pour nous, un système distribué est une *machine virtuelle répartie*. Son objectif est de fournir une machine abstraite évoluée où les détails matériels de bas niveau sont cachés à l'utilisateur. Ainsi le système distribué va cacher des détails tels que le découpage en trame des données transférées ou le dialogue avec les contrôleurs de périphériques.

En fait, il existe deux aspects dans la définition d'un système distribué: l'*aspect matériel*, et l'*aspect logiciel*.

Du point de vue *matériel*, un système distribué comprend un ensemble de sites (processeurs, mémoire périphériques) connectés par un ou plusieurs réseaux de communication. Le type du réseau de communication permet de distinguer les *réseaux de stations*, présentant une architecture *faiblement couplée* et les *machines parallèles* dont l'architecture est *fortement couplée*.

Du point de vue *logiciel*, l'exploitation de cet ensemble de sites est plus difficile que l'exploitation de l'unique processeur d'un système centralisé. Le programmeur peut avoir conscience de la multitude et de la diversité des systèmes d'exploitation de chaque site, nous parlons alors de *systèmes d'exploitation de réseau*. Il peut aussi avoir l'impression de travailler sur un "système centralisé virtuel" au lieu d'un ensemble de machines distinctes, nous parlons alors de *système d'exploitation distribué*.

**Aspect matériel**

L'architecture matérielle des systèmes distribués est de type *MIMD* (Multi-Instructions, Multi-Données). Nous nous intéresserons dans un premier temps aux *architectures faiblement couplées*, type *réseau de stations* et ensuite aux *architectures fortement couplées* type *machine parallèle*.

1) Réseau de stations : architectures faiblement couplées

L'informatique distribuée se développe plus particulièrement au travers des réseaux de stations. Cette architecture est basée sur un réseau local de communication (type Ethernet, FDDI ou ATM) sur lequel est connecté un ensemble de sites ou stations. Une station est un ordinateur avec un ou plusieurs processeurs, une mémoire locale et un ensemble de périphériques. La mémoire n'est accessible que localement par les processeurs. De même, les périphériques ne sont pilotables que localement.

Tous les sites ne présentent pas la même architecture. Ils peuvent différer par leur type de processeur, leur fréquence d'horloge, leur capacité mémoire, ou par les périphériques présents. Nous parlons alors de réseau hétérogène de stations.

Le but de ce type d'environnement est de fournir un environnement de travail interactif

où l'utilisateur a la possibilité de stocker des fichiers, compiler des programmes, imprimer des textes ou envoyer du courrier électronique (E-mail). Un des objectifs est de fournir un maximum d'interactivité et des facilités d'accès aux différents outils.

La figure 1.1 donne un exemple d'un réseau de stations. Celui-ci est constitué d'un ensemble de stations de travail, de deux serveurs, d'imprimantes, et d'un groupe (pool ou cluster) de stations interconnectées par un réseau à très haut débit. Ce réseau de stations, inspiré de celui du LIFL, ressemble aux réseaux de la plupart des laboratoires ou entreprises.

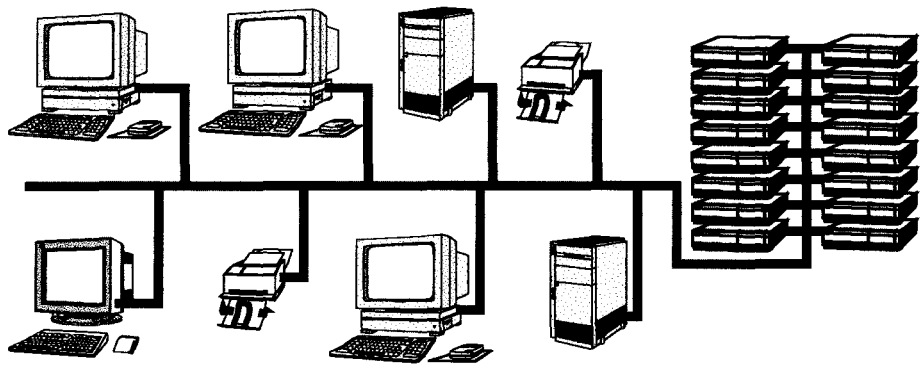


**figure 1.1** Réseau de stations

Les stations de travail sont chacune dédiées à une personne qui travaille directement sur cette machine. Les serveurs gèrent le système de fichiers dont chaque composant est accessible de n'importe quelle machine en utilisant les mêmes conventions de nommage. Ils fournissent un espace de stockage persistant de données pour l'utilisateur. Les imprimantes permettent l'impression des documents, et sont directement reliées au réseau. Le pool de stations n'est pas directement accessible par un utilisateur. Ce dernier doit se connecter par l'intermédiaire d'une station de travail. Le pool est utilisé simultanément par différents utilisateurs. Au LIFL, les stations de travail sont en majorité des Sparc de Sun sous SunOs ou Solaris. Le pool est, quant à lui, constitué d'une ferme de seize stations Alpha sous OSF/1 connectées par un réseau FDDI.

2) Machines parallèles: architectures fortement couplées

Une architecture parallèle peut se décrire par un ensemble de processeurs et d'un réseau de communication spécialisé utilisé, soit pour connecter directement les différents processeurs entre eux, soit pour connecter les processeurs à des composants mémoire. Différents types de réseaux d'interconnection ont été étudiés comme les cross-bars, les treillis ou les hypercubes.

L'objectif de ce type de système est de fournir une grande puissance de calcul pour des applications scientifiques, comme par exemple les calculs pour les prévisions météo ou les problèmes de simulation moléculaire.

Ce type d'environnement privilégiera les aspects "exécution parallèle" au détriment d'aspects plus interactifs. Ce type de système proposera généralement des modes d'interaction qui s'apparentent plus aux modes traditionnels **batch** avec des possibilités de partitionnement de machine.

Un exemple de machine parallèle est donné par la machine Cray T3D [CrayT3D] (figure 1.2). C'est une machine multi-noeuds à mémoire distribuée construite autour d'un réseau d'interconnection très performant type hypercube. Chaque noeud est constitué de deux éléments indépendants: un processeur (RISC, 64 bits) cadencé à 150 Mhz et une mémoire locale pouvant atteindre 64MB (16MWords). L'interface matérielle (hardware)

d'interconnection donne à la T3D un espace d'adressage unique. Un processeur peut accéder sa mémoire locale, et également la mémoire des autres processeurs donnant ainsi une mémoire distribuée. Bien sûr les accès distants sont plus lents que les accès locaux (Un accès distant comprend un accès local (25 cycles) plus 2 cycles par noeud traversé plus encore 2 à 10 cycles pour le routage).

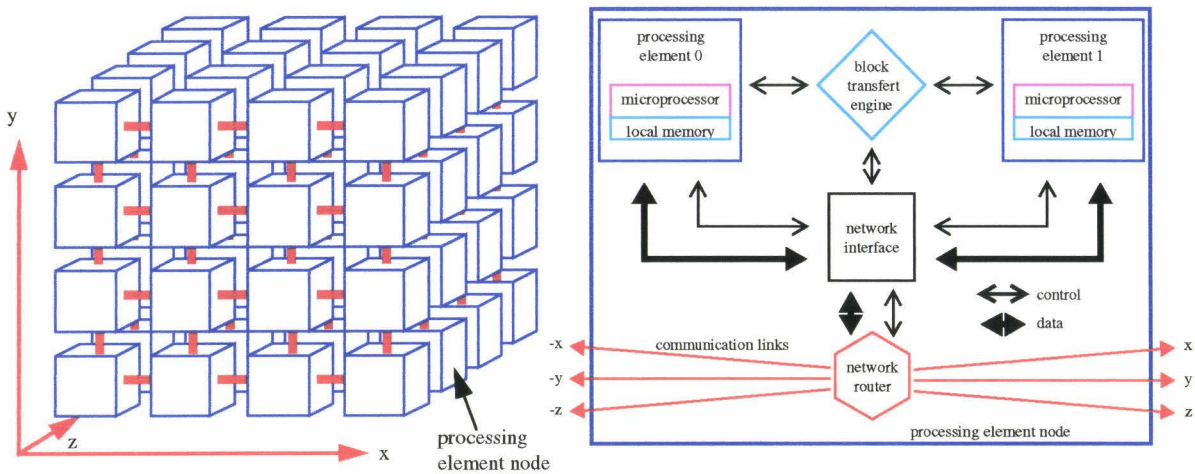



**figure 1.2** Machine parallèle: architecture et détail d'un noeud de la Cray T3D

Les machines parallèles sont en général des machines spécifiques conçues pour travailler sur un problème demandant beaucoup de temps CPU. Elles sont très coûteuses et ne sont pas encore abordables par des structures de taille moyenne (laboratoires, PME).

C'est pourquoi nous nous intéressons plus aux réseaux de stations de travail. Ces derniers sont constitués de machines à usage général, et peuvent être étendus progressivement par l'ajout de nouvelles stations. De plus, les performances des réseaux de communication ne cessent de croître, améliorant encore leurs possibilités.

Dans la suite de cette thèse, nous parlerons aussi bien de site que de machine pour désigner une station de travail.

**Aspect logiciel**

Si l'aspect matériel d'un système distribué est important, l'aspect logiciel l'est tout autant, car son interface, si elle est simple et conviviale, facilite sont utilisation et donc son succès. En effet, toutes les interactions entre le matériel et l'utilisateur se font par l'intermédiaire du système d'exploitation. Deux familles de systèmes d'exploitation ont été proposées et développées: les **systèmes d'exploitation de réseau** **systèmes d'exploitation distribués**

1) Systèmes d'exploitation réseau - NOS

Les systèmes d'exploitation réseau (ou NOS Network Operating System) sont constitués d'un ensemble de systèmes d'exploitation mis les uns à coté des autres. Ces systèmes d'exploitation peuvent être différents (réseau hétérogène), mais ils fournissent un environnement où les utilisateurs sont au courant de la multiplicité des machines, et permettent d'accéder à une ressource éloignée soit en se connectant à la machine distante, soit en transférant les données.

Un exemple de NOS est donné par un réseau de stations où chaque utilisateur exécute ses programmes sur sa machine. Quand un utilisateur trouve sa machine trop chargée, il en

cherche une inutilisée (ou peu utilisée), y installe un terminal et y exécute une partie de ses commandes. L'utilisateur a connaissance de chacune des stations et demande explicitement l'exécution à distance, nous avons donc à faire à un système d'exploitation de réseau.

Les systèmes d'exploitation de réseau sont formés d'un ensemble de systèmes d'exploitation auxquels on a ajouté une couche offrant des possibilités d'accès à des ressources distantes. L'implémentation de cette couche n'est pas toujours efficiente (exemple: répartition de la charge de travail) car les machines coopèrent très peu entre elles.

2) Les systèmes d'exploitation distribués - DOS

Dans un système d'exploitation distribué (ou DOS Distributed Operating System, à ne pas confondre avec Disk Operating System), le système d'exploitation est réparti sur les machines, chacune coopérant avec les autres afin d'équilibrer l'utilisation des ressources.

L'utilisateur n'a pas connaissance de la topologie du réseau, et l'accès aux ressources éloignées se fait de la même manière que l'accès aux ressources locales. La migration des données ou des processus d'un site à un autre est placée sous le contrôle du système d'exploitation réparti. Ainsi, quand l'utilisateur veut exécuter une commande, le système recherche une machine, y exécute la commande, tout en affichant les résultats dans la console de l'utilisateur, sans que celui-ci ne sache exactement sur quelle machine sa commande est exécutée.

Prenons l'exemple du système d'exploitation Amoeba [Tanenbaum90]. Ce système fonctionne sur un réseau de station comparable à celui de la figure 1.1, mais à la différence de ce dernier, chaque station possède un micro-noyau similaire (à la différence d'architecture près). La création d'un nouveau processus provoque la recherche, par le système, du meilleur endroit pour l'exécuter. Cet endroit peut être la machine de l'utilisateur, une machine inutilisée appartenant à une personne absente, ou encore un des processeurs du pool. Cette recherche est entièrement transparente à l'utilisateur. L'ensemble du système composé du réseau de machines apparaît alors à l'utilisateur comme un système centralisé virtuel.

Les systèmes d'exploitation distribués prennent en charge la répartition des ressources, et les problèmes de communication, allégeant d'autant la tâche de l'utilisateur. Cependant, ils s'appuient sur des nouveaux noyaux et leur interface de programmation ne vérifie pas toujours les standards. De plus, ils nécessitent bien souvent un changement du noyau des machines. Par conséquent leur utilisation n'est pas encore très répandue parmi les architectures distribuées existantes.

Les NOS sont plus répandus, ils ne demandent qu'un ensemble de machines connectées par un réseau de communication et quelques services de base. C'est pourquoi, dans le cadre de cette thèse, nous nous intéressons plus particulièrement aux NOS. Dans la suite de cette thèse, toute référence à un système d'exploitation sous entendra une référence à un NOS, sauf indication contraire.

### 1.1.2 Services de base des systèmes distribués

La programmation des systèmes distribués est complexe, et certaines tâches élémentaires sont répétitives et fastidieuses. C'est pourquoi le système d'exploitation prend en charge la gestion de bas niveau d'un certain nombre de services. Il fournit à l'utilisateur une interface simple d'emploi, interface qui n'est pas toujours homogène entre les machines (la syntaxe des commandes peut varier entre les différentes machines). Les services de base fournis par un système distribué permettent l'exécution des processus, ainsi que la communication et la synchronisation entre ces processus.

### Exécution des processus

La notion de processus est la même dans les systèmes distribués et dans les systèmes centralisés. Un processus est considéré comme un processeur virtuel exécutant un programme. Un processus possède alors son propre contexte (pile, registres), son espace d'adressage (de zéro à la taille maximale du site), et son code à exécuter, placé dans cet espace.

Un site peut supporter l'exécution «simultanée» de plusieurs processus, ceux-ci se partageant le temps du processeur (time-sharing). L'ensemble des processus du site est géré par un ordonnanceur qui se charge de distribuer l'utilisation du processeur physique entre les différents processus.

Dans un système distribué, plusieurs sites sont accessibles à l'utilisateur qui peut alors choisir celui sur lequel il veut exécuter un processus. Chacun de ces différents sites possède son propre ordonnanceur local qui ne communique pas avec les autres ordonanceurs. Cette absence d'ordonnanceur global complique la tâche du choix d'un site lors de la création d'un nouveau processus.

Le choix d'un site peut se faire manuellement par le programmeur: après s'être posé la question "où mon programme a-t-il les meilleures chances de s'exécuter rapidement ?", celui-ci recherche un site, soit en regardant «de visu» si il y a une machine qui n'est pas utilisée, soit en examinant la charge des sites à l'aide de commandes à distance. Une fois le site trouvé, le programme est lancé avec le même procédé de commande à distance (*rsh site_peu_utilisé nom_du_programme*).

Des outils d'aide au placement de processus ont été développés [Hemery94]. Ils prennent en charge la répartition des processus entre les différents sites disponibles. Ainsi, LSF (Load Share Facilities) [Zhou] fournit des outils facilitant l'exécution et la répartition des processus sur un ensemble de sites hétérogènes.

Il faut remarquer que dans un DOS, la répartition des processus est gérée par le système. Celui-ci exécute chaque nouveau processus sur un des sites disponibles. Le placement des processus se fait de manière transparente pour l'utilisateur: celui-ci a l'impression d'utiliser une unique machine, il n'a pas conscience de la répartition.

Un système distribué se doit donc de fournir des mécanismes permettant de créer les processus sur les différents sites du système. Il fournit des mécanismes permettant d'intervenir sur les sites locaux ou distants, aussi bien pour la création que pour la terminaison d'un processus. La répartition des processus sur les différents sites se fait à l'aide de ces mécanismes, soit manuellement, soit à l'aide d'outils spécifiques. Cette répartition demande la connaissance d'informations qui sont réparties sur les différents sites.

### Communication inter-processus

La communication est un élément essentiel des systèmes distribués. Elle permet aux processus d'échanger de l'information.

Dans un système centralisé mono-processeur, elle prend la forme de tubes ou d'IPC (Inter Process Communication) et se fait essentiellement à l'aide de la mémoire commune: Les données à transmettre sont lues et écrites dans une zone partagée entre deux processus.

Les systèmes distribués possèdent rarement de la mémoire commune. La communication utilise donc un autre mécanisme: elle se fait par des échanges de messages circulant sur le réseau de communication.

Un mécanisme très répandu est la communication par *socket* qui autorise la communication entre des processus quelconques, appartenant à un même site ou à des sites

distincts reliés par un réseau de communication. Ce mécanisme offre une interface standard, et se rencontre sur tous les systèmes Unix.

Un problème important lié aux communications est le problème de l'hétérogénéité dû à la différence de représentation des données sur les différentes architectures. C'est pourquoi le protocole XDR (External Data Representation) a été défini [Corbin90]. Il permet l'échange de données entre machines ayant des représentations internes différentes. Le protocole XDR définit une norme de représentation des données. Un processus émetteur encode ses données selon la norme, données qui seront décodées par le processus récepteur.

L'utilisation du protocole XDR, si il permet l'échange de données entre machines hétérogènes, complique encore le travail du programmeur qui doit se charger de l'encodage et du décodage.

Les sockets proposent essentiellement des communications point à point, c'est à dire entre deux machines distinctes. Un autre mécanisme été développé, permettant d'envoyer un même message à plusieurs récepteurs : ce sont les groupes de communication (figure 1.3). Un groupe est un ensemble de processus considéré comme une seule entité. Quand un message est envoyé au groupe, chaque membre du groupe (chaque processus) reçoit le message.

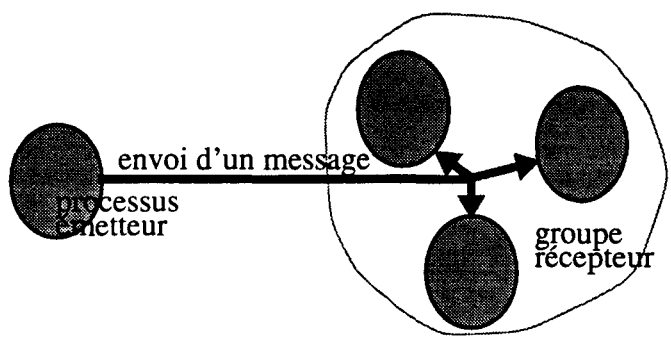



**figure 1.3** Groupe de communication

Les groupes sont généralement employés dans des applications ayant besoin de tolérance aux pannes, dans lesquelles des processus (ou plutôt des sites) peuvent "tomber en panne".

Les groupes de communications apparaissent rarement directement au niveau des systèmes d'exploitations NOS. On les trouve dans des systèmes DOS comme Amoeba [Amoeba90], et plus généralement dans des modèles de programmation comme HORUS [Renesse93] qui basent toute leur stratégie dessus, ou dans des modèles de processus communicants comme PVM [Geist94].

La communication entre processus fait intervenir un certain nombre de couches aussi bien matérielles que logicielles servant à la mise en forme et au transport des messages. La traversée de ces couches introduit des surcoûts de communication qui ne sont pas négligeables, notamment si on les compare aux performances des processeurs et de la mémoire. Les technologies nouvelles (comme par exemple la fibre optique) accroissent sans cesse les performances des communications, mais celles-ci restent quand même pénalisantes comparées à la vitesse de calcul des processeurs.

### Synchronisation

La synchronisation permet à un ensemble de processus de s'accorder sur une vue partielle et cohérente du système. Ainsi, au point de synchronisation, chaque processus est dans un état définissable (qui peut être déterminée) par le programmeur.

Des outils de synchronisation ont été développés pour les systèmes centralisés, comme les sémaphores et les moniteurs. Malheureusement, ces outils ne sont pas adaptés aux systèmes distribués car ils nécessitent une vue centralisée de l'information, information qui est répartie entre les sites.

Mais les systèmes distribués possèdent une forme de synchronisation qui leur est intrinsèque: c'est la synchronisation implicite par échanges de messages. En effet, un processus attendant un message est synchronisé par la réception de ce dernier. La synchronisation de deux processus peut alors simplement se faire avec deux messages: chaque processus envoie un message puis attend une réponse en provenance de l'autre processus (figure 1.4). Quand les messages sont reçus, les deux processus sont synchronisés.

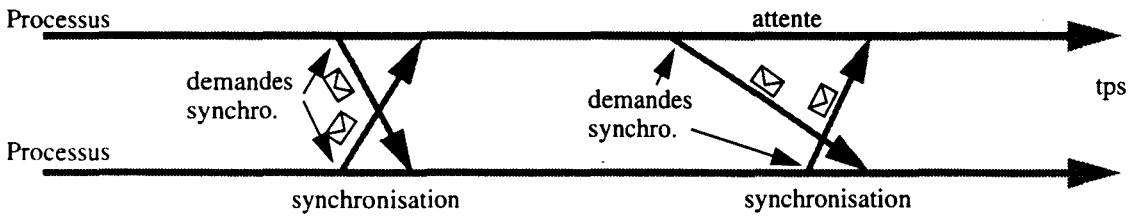



**figure 1.4** Synchronisation implicite par messages

La généralisation de ce mécanisme à plus de deux processus est lourde à mettre en oeuvre, et se pose la problématique de la synchronisation dans un contexte distribué. Cette synchronisation doit utiliser des algorithmes distribués afin d'éviter de centraliser la prise de décision.

Différents algorithmes ont été développés pour les systèmes distribués. Parmi ceux-ci, nous pouvons citer les algorithmes basés sur des horloges logique [Lamport79], les algorithmes utilisant le passage de jeton sur un anneau distribué, ou encore les algorithmes de transactions. Ces algorithmes sont assez complexes et leur étude sort du cadre de cette thèse.

### 1.1.3 Environnement d'exécution

L'environnement d'exécution comprend l'ensemble des logiciels qui ne font pas partie du système d'exploitation, et qui ne sont pas non plus les applications de l'utilisateur. Parmi ces logiciels on trouve les outils de développement, de programmation et de mise au point d'applications distribuées, tels que les langages de programmation ou les dévermineurs. On trouve aussi des outils déchargeant le programmeur de tâches fastidieuses et répétitives, tels que les ramasses-miettes (Garbage Collector, GC) et les outils d'aide au placement des processus (déjà rencontrés page 20).

Dans le cadre de cette thèse, nous ne décrirons pas tous les logiciels disponibles. Nous allons nous contenter d'aborder les outils tel que le ramasse-miettes et le dévermineur, qui correspondent aux applications que nous visons.

**Ramasse-miettes**

Un programme en cours d'exécution crée dynamiquement des objets (structures, variables) au fur et à mesure de ses besoins. Afin d'éviter de saturer l'espace mémoire disponible, ces objets doivent être détruits dès qu'ils deviennent inutiles, ou inaccessibles par le programme. Malheureusement, il arrive bien souvent que le programmeur ne sache pas quand détruire un objet, ou simplement qu'il ne s'en soucie pas. Le rôle d'un ramasse-miettes est de récupérer et de détruire de tels objets.

La réalisation d'un ramasse-miettes dans un contexte distribué est rendue délicate de par

le fait que les objets sont répartis sur différents sites, et que ces objets peuvent dépendre les uns des autres (comme par exemple une liste chaînée dont les maillons sont répartis sur les sites).

Une solution a été proposée [Dumoulin93] pour des objets actifs. Cette solution est bien sûr applicable à des objets passifs. Elle consiste à construire un graphe de dépendance des objets, et d'y appliquer un algorithme détectant les miettes (algorithme de Kafura [Kafura91]). Malheureusement, l'absence de mémoire commune complique la construction du graphe. La solution retenue a été de le construire sur un site, d'appliquer une version modifiée de l'algorithme, et de communiquer les résultats aux autres sites.

Une meilleure solution serait de construire le graphe dans une mémoire partagée, et de faire appliquer l'algorithme de récupération par chaque site voulant récupère ses miettes.

Le graphe représente une "photographie" de l'application à un instant *t*. Sa construction ne nécessite pas l'arrêt de l'application qui, continuant de s'exécuter, crée de nouvelles miettes. Les miettes récupérées correspondent à celles de l'instant *t*, les miettes créées par la suite seront récupérées au prochain passage du ramasse-miettes. Ainsi, le ramasse-miettes peut fonctionner avec des données un peu anciennes.

**Mise au point des programmes**

La mise au point des applications passe par l'utilisation d'outils appelés *dévermineur* ou *debugger*. Ces outils permettent de suivre l'exécution d'un programme, instruction par instruction, de s'arrêter à des endroits choisis (points d'arrêts), et même pour certain de visualiser l'état des variables.

Dans un système centralisé, ces outils tirent partie du fait que toutes les informations nécessaires sont présentes localement. Leur réalisation dans un système distribué est rendue difficile par le fait que les informations sont réparties sur les sites, mais aussi par le non déterminisme de l'application, l'absence d'horloge commune permettant d'ordonner les événements, et l'absence de mémoire commune dans laquelle partager les variables et l'état global de l'application.

Une solution a été proposée [Roos94] permettant une réexécution post-mortem d'une application. Cette application est exécutée normalement une première fois. Cette exécution génère des traces qui sont ensuite utilisée post-mortem par le dévermineur afin de réexécuter l'application exactement comme la première fois. Chaque nouvelle exécution est identique à la première, permettant au programmeur d'examiner le comportement de l'application en enlevant le facteur non déterministe entre deux exécutions. Par contre, l'absence de mémoire commune rend difficile aussi bien la visualisation des variables, que la récolte des traces.

## 1.2 Modèles de programmation

Des modèles de programmation ont été développés afin de formaliser le développement des applications sur des systèmes distribués. Parmi ces modèles de programmation, nous allons étudier les processus communicants, dans lequel chaque processus est indépendant, et communique avec les autres à l'aide de canaux, sans possibilité de mise en commun de l'information. Puis nous aborderons le modèle des appels de procédure à distance (RPC) qui étend le modèle procédural classique à un environnement distribué, mais sans possibilité de données globales. Enfin, nous introduirons le modèle des mémoires partagées distribuées, pour lequel la programmation est proche de celle d'un système centralisé.

### 1.2.1 Processus communicants

Le modèle de programmation des processus communicants a été introduit par Hoare [Hoare78] avec le modèle Communicating Sequential Process (CSP). C'est un modèle basé sur

la communication entre processus. Une application est constituée d'un ensemble de processus s'exécutant simultanément (du point de vue du programmeur) et coopérant entre eux.

Dans la description originale du modèle CSP, un processus permet d'exprimer des actions qui peuvent être du calcul, ou de la communication avec d'autres processus. Cette communication se fait par l'intermédiaire de canaux. Un canal permet à deux processus de communiquer de manière unidirectionnelle: un processus écrit dans le canal tandis que l'autre lit le canal. La communication est synchrone, si l'un des deux processus n'est pas dans une phase de communication, l'autre processus qui a commencé la phase de communication reste en attente. A la fin de la communication, chaque processus reprend sa propre exécution.

Le langage Occam [Occam88] a été développé à partir du modèle CSP. Il permet au programmeur d'exprimer simplement la structure hiérarchique et modulaire de son application par encapsulation d'un ensemble de processus communicants qui est alors vu comme un seul processus. Ces processus peuvent s'exécuter sur un ou plusieurs processeurs. Cette répartition n'influence pas la phase de conception de l'application parallèle.

Dans Occam, il n'y a pas de variables globales. Toutes les variables sont définies dans les processus. Pour utiliser une valeur globale, le programmeur doit soit déclarer une valeur constante, soit mettre en place un mécanisme propageant chaque modification de la valeur. Ce mécanisme utilisera les canaux de communication.

Occam a été initialement conçu pour les machines à base de Transputers possédant des liens de communication physique vers les processeurs voisins. Le langage est fortement orienté vers ce type d'architecture.

Le modèle CSP et l'implémentation Occam sont considérés comme les précurseurs des processus communicants. Depuis lors, d'autre implémentations basées sur le modèle CSP ont été proposées.

Dans toutes ces implémentations, une application est formée d'un ensemble de processus s'exécutant simultanément et coopérant entre eux par le biais des communications.

Les différences résident dans le mode de communication, synchrone ou asynchrone, dans la façon de créer et de répartir les processus et dans l'interface proposée au programmeur, langage ou bibliothèque.

**Création, répartition et contrôle des processus**

Dans le modèle des processus communicants, une application est constituée de différentes activités ou processus exécutés simultanément sur les différents processeurs ou sites proposés par l'architecture. Les problèmes qui se posent alors au programmeur concernent la répartition des activités, leur création, et le contrôle de l'application.

La répartition des activités sur les différents sites peut être statique ou dynamique.

Dans une répartition statique, le programmeur spécifie dans son application l'endroit où vont s'exécuter les processus. La répartition des activités et la configuration de l'application sont ainsi connues avant l'exécution.

Dans une répartition dynamique, le choix du placement des activités se fait au cours de l'exécution. Un mécanisme d'équilibrage de charge peut être employé afin de répartir uniformément les processus sur les différents sites. Ce mécanisme prend en compte des critères comme le nombre d'activités déjà créées sur un site, la puissance de ce site ou la disponibilité de certaines ressources.

La création des activités composant l'application peut se faire individuellement ou par création arborescente.

Dans le premier cas, le programmeur crée chaque processus individuellement. Il peut être aidé dans sa tâche par des outils lui permettant de décrire son application et de lancer l'ensemble de ces processus (exemple: Le *"schéma de l'application"* de certaines implémentations de MPI [MPI95]). Cette façon de faire donne au programmeur la possibilité d'ajouter dynamiquement des processus. Cependant, il doit exister un mécanisme permettant à un nouveau processus d'entrer en contact avec les autres processus.

Dans la création arborescente, le programmeur lance le premier processus qui lance d'autre processus. Ces derniers peuvent à leur tour lancer de nouveau processus. Il en résulte un arbre de processus ayant comme racine le processus lancé par le programmeur. Tous les processus ont un ancêtre, il est alors facile de les mettre en relation afin de communiquer.

Le contrôle de l'application peut être centralisé ou entièrement distribué.

Le contrôle centralisé est connu sous le nom de modèle maître - esclaves. Un processus maître dirige les opérations: il crée des esclaves, leur répartit le travail à faire puis attend l'arrivée des résultats. Le processus maître sert uniquement à contrôler l'application. Le calcul est effectué par les esclaves qui envoient leurs résultats au maître. Quand le maître a reçu tous les résultats, l'application est terminée.

Dans un contrôle entièrement distribué (*fully distributed*), tous les processus effectuent une part du calcul. La difficulté pour le programmeur est de répartir le calcul entre les processus et détecter la terminaison de l'application. En effet, il n'existe pas de variable partagée entre tous les processus et permettant de connaître l'état global de l'application. Cet état global doit être diffusé à l'aide de communications inter-processus.

**Implémentations**

Les implémentations du modèle des processus communicants se présentent comme des nouveaux langage de programmation ou comme des bibliothèques de fonctions.

Pour les langages, nous trouvons Occam, et pour les bibliothèques, principalement PVM et MPI.

1) Occam

Nous avons déjà évoqué le langage Occam (page 24). Ce langage est originellement conçu pour programmer des machines constituées d'un grand nombre de Transputers. Des versions distribuées pour réseaux de stations ont été étudiées, mais elles ne sont guère utilisées dans les architectures distribuées.

2) PVM

PVM (Parallel Virtual Machine) [Geist94] est une bibliothèque de fonctions appelables à partir des langages C ou Fortran. PVM permet de voir un ensemble de machines hétérogènes comme une seul machine parallèle. Les noeuds de la machine virtuelle peuvent être des machines spécialisées, des machines multiprocesseurs ou des stations de travail. Ces machines sont connectées entre elles par un réseau de communication (Ethernet, FDDI, ATM, ...).

PVM gère les processus, l'échange de données, et la synchronisation. Il est possible de créer des tâches (figure 1.5), de les faire communiquer par échanges de messages, et de les synchroniser par des barrières.

Une tâche est un processus Unix à part entière. Les tâches PVM s'exécutent en parallèle et sont indépendantes les unes des autres. Le programmeur peut préciser le noeud sur lequel il veut que la tâche soit exécutée, ou laisser ce choix au système PVM. Les tâches ne peuvent pas partager des variables communes même si elles ont des liens de parenté.

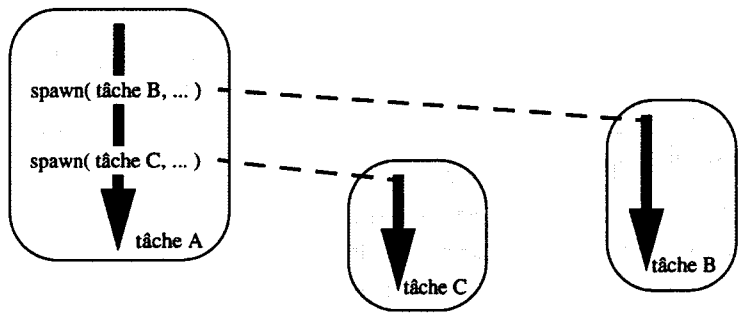



**figure 1.5** Tâches PVM

La communication entre tâches se fait par échange de message. La tâche émettrice construit le message dans un *buffer* PVM à l'aide de fonctions «d'empaquetage». Ces fonctions codent le message en utilisant le protocole XDR, ce qui permet d'envoyer le message à n'importe quel type de machine. Des fonctions de désempaquetage permettent à la tâche réceptrice d'extraire le message.

PVM offre aussi d'autres fonctionnalités, comme la synchronisation par barrière ou la communication de groupe. Dans cette dernière, il existe une fonction de réduction permettant de calculer une valeur tel que le minimum ou le maximum sur un ensemble de tâche.

La notion de barrière de synchronisation permet de synchroniser un ensemble de tâche. Une tâche appelant la barrière est bloqué jusqu'à ce que toutes les tâches participantes aient elles aussi appelées la barrière. A ce moment là, toutes les tâches sont débloquées et synchronisées.

3) MPI

MPI est une autre bibliothèque permettant l'écriture d'applications basées sur le modèle des processus communicant. MPI (Message Passing Interface) est un standard pour le passage des messages entre processus. Dans la version 1.1 du standard [MPI95], il est spécifié que MPI ne s'occupe pas de la création et de la gestion des processus. Ceci est en passe de changer dans la (future) version 2.x [MPI96].

MPI étant uniquement une bibliothèque de communication, rien n'est dit à propos de variables partagées. MPI offre cependant des fonctions de "communication collective" facilitant la diffusion et le calcul d'une valeur globale. On trouve tout d'abord la fonction *broadcast* permettant d'envoyer un message à un groupe de processus. Il y a ensuite les fonctions *scatter-gather*, et enfin les fonctions de réduction globales. Ces fonctions sont synchrones: elles aboutissent quand chaque processus participant à l'opération a appelé la fonction.

La fonction *scatter* permet d'envoyer un ensemble de données et de répartir ces données entre plusieurs processus. La fonction *gather* est la fonction inverse, elle permet à un processus de récupérer des données réparties sur plusieurs processus.

Les fonctions de réduction globale permettent le calcul d'une valeur globale à l'application. Chaque processus appelle la fonction de réduction, et l'un d'entre eux reçoit le résultat de l'opération. Un certain nombre d'opérations sont prédéfinies comme la somme, le minimum ou le maximum, mais le programmeur reste libre de définir ses propres opérations.

**Conclusion**

Le modèle des processus communicants facilite la programmation d'applications

distribuées. Ceux-ci coopèrent entre-eux par l'intermédiaire des communications. Ce modèle s'apparente à l'architecture sous-jacente: les messages circulent sur les liens de communications.

Cependant, l'absence de mémoire commune rend difficile la mise en commun de données ou la prise de connaissance de l'état global de l'application. Cet état global est nécessaire pour réaliser des fonctionnalités comme le contrôle des processus, leur répartition ou leur synchronisation. Dans un modèle proposant uniquement des communications et pas de mémoire commune, la réalisation de ces fonctionnalités nécessite un grand nombre d'échanges de messages. Le programme devient alors compliqué à écrire, et sa lisibilité est réduite.

### 1.2.2 Remote Procedure Call (RPC)

Le modèle RPC (Remote Procedure Call) [Birrell84] ou appel de procédure à distance, est un modèle de programmation permettant d'appeler des procédures situées sur d'autres machines. Le but des RPC est de cacher au processus demandeur le fait que l'appel se déroule à distance. Les RPC étendent donc la notion d'appel procédural à un ensemble de machines (figure 1.6).

#### Implémentation

Les RPC sont une forme évoluée du modèle client-serveur. Le processus client demande un service qui est rempli par le processus serveur. Cette demande ce fait par appel d'un *stub* qui est une fonction (ou procédure) chargée des manipulations de bas niveau (sockets, protocole XDR). Du point de vue du programmeur, la demande de service se fait par le simple appel d'une procédure.

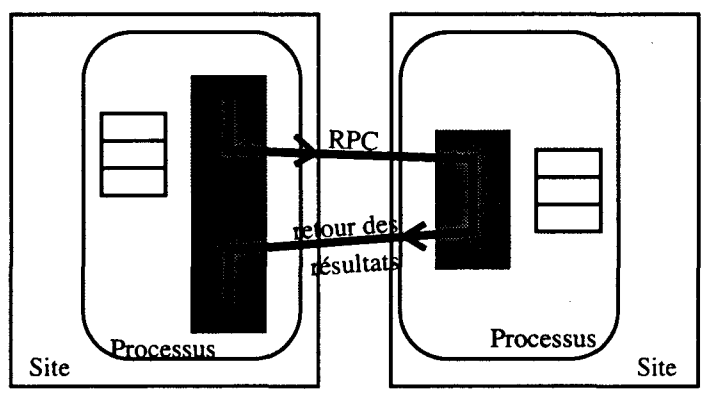



**figure 1.6** Le modèle RPC

Dans le modèle RPC, les processus sont généralement lancés individuellements. Il n'existe par conséquent pas de lien de filiation entre un processus demandeur et un processus serveur. Le premier localise le second à l'aide d'un "serveur de nom" qui gère une table de correspondance entre le nom d'un service et l'adresse du serveur remplissant ce service.

#### Conclusion

Le modèle des RPC est présent sur de nombreux systèmes. L'une des première implémentation est celle de Sun Micro System pour la réalisation de NFS.

Ce modèle de programmation est très utilisé dans les systèmes d'exploitation de réseau (NOS). Il souffre tout de même d'un inconvénient: il ne permet pas la déclaration de données globales comme dans un programme classique (données communes à l'ensemble des fonctions ou procédures). Le programmeur doit passer en argument de la fonction toutes les données globales nécessaires au serveur.

### 1.2.3 Mémoire partagée répartie

Le modèle des Mémoires Partagées Réparties (MPR ou DSM Distributed Shared Memory) a été proposé par Li en 1986 [Li86], puis par Li et Hudak en 1989 [Li89].

Une mémoire partagée répartie donne l'illusion d'avoir une mémoire commune accessible par chaque processus d'une application. La programmation d'une application distribuée est alors proche de la programmation classique d'un système centralisé. Le partage d'information est naturel: il se fait dans la DSM.

Pour illustrer ces avantages nous allons comparer la réalisation d'une même application d'une part avec des échanges de messages, et d'autre part avec une DSM.

Considérons pour cela une application recherchant une valeur minimum, comme par exemple le plus court chemin pour aller d'un point à un autre en passant par différents chemins. L'application est composée de plusieurs processus chacun chargé d'évaluer des solutions. La valeur minimum est alors une variable qui doit être connue de l'ensemble des processus. De plus, cette variable peut être modifiée par chacun des processus.

Dans un système à base d'échange de message, chaque modification de la variable nécessite la propagation par message de la nouvelle valeur. Chaque processus doit alors scruter régulièrement l'arrivée des messages afin de prendre en compte la nouvelle valeur. Il faut aussi mettre en place un mécanisme permettant d'assurer que la modification (quasi-)simultanée de la variable par plusieurs processus ne la mette pas dans un état incohérent.

Avec une DSM, la valeur minimum est partagée dans l'espace commun, la rendant ainsi accessible de tous. La figure 1.7 illustre la recherche du minimum par un ensemble de processus à l'aide d'une DSM: chaque processus calcule une solution, et la compare avec la plus petite solution déjà trouvée. Comme on peut le constater, l'algorithme est simple, et ressemble à celui employé dans un système centralisé.

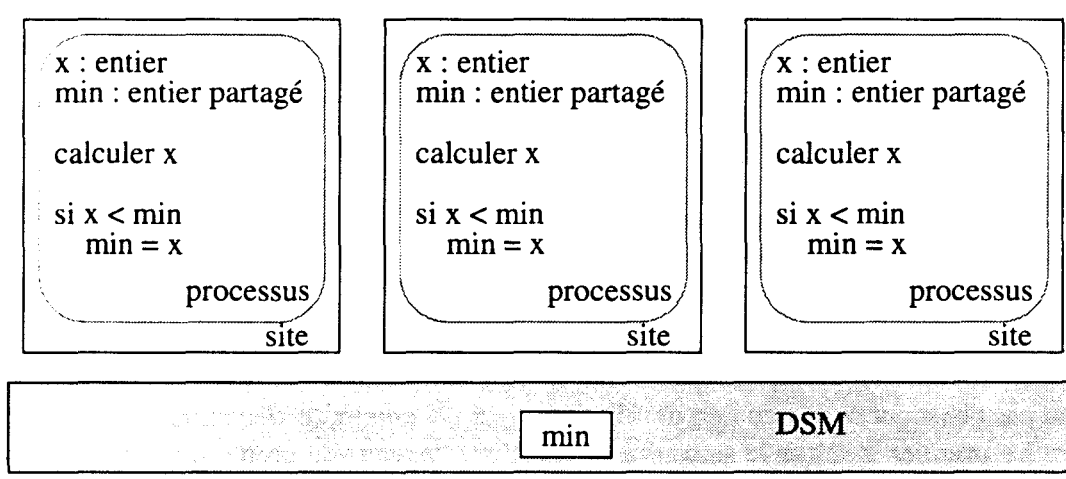



**figure 1.7** Recherche du minimum avec une mémoire partagée

#### Implémentation

Différents modèles de mémoire partagée ont été développés. Si leur objectif est le même, proposer une mémoire commune, leur implémentation et leur programmation diffèrent.

En absence de mémoire physique commune, l'implémentation cache au programmeur les échanges de messages. Pourtant ces échanges existent, et suivent (à quelques améliorations près) la solution à base de messages évoquée plus haut. Le mécanisme employé pour garantir

la cohérence de la valeur joue un rôle important dans cette implémentation, comme nous le verrons dans le chapitre suivant.

**Conclusion**

Une mémoire partagée permet un partage aisé des informations. Lors du développement d'une application, le programmeur se concentre sur les objets, et non sur leurs localisations dans les processus.

L'implémentation se fait à l'aide d'échanges de messages car il n'existe pas de mémoire physique commune. Les performances sont alors "au mieux" aussi bonnes que si l'application était réalisée en programmant les échanges de messages.

En réalité, la gestion interne de la DSM, et notamment le maintien de la cohérence des données font chuter ces performances. Afin de les améliorer, divers modèles de cohérence sont proposés, modèles qui sont étudiés dans le chapitre suivant.

## 1.3 Conclusion

Les systèmes distribués sont une réponse à la demande sans cesse croissante de puissance de calcul. Nous venons de voir que le développement d'applications demande des outils spécifiques et de nouveaux modèles de programmation. Ce développement est plus difficile que le développement sur un système centralisé, du fait notamment de l'absence de mémoire commune.

Notre travail s'est concentré sur des applications systèmes, ou plus exactement des applications situées juste au dessus du système, telles que les groupes de communication, les outils d'aide au placement (répartition de charge), les outils de déverminage ou le ramasse-miettes. Ces applications ont des points communs:

- Elles nécessitent une vue globale de l'état du système, ou une mise en commun de l'information. L'absence de mémoire commune rend difficile l'obtention de cette vue et la mise en commun de l'information qui doivent alors être réaliser avec des RPC ou des processus communicants.
- Elles peuvent le plus souvent fonctionner avec des informations qui ne sont pas toujours les plus récentes. Le besoin d'information "up to date" est ponctuel, quand il n'est pas inexistant comme pour le ramasse-miette.

Notre idée est d'utiliser une mémoire partagée répartie pour simplifier la réalisation de ces applications. Nous allons maintenant étudier le modèle de programmation des mémoires partagées réparties, dont l'objectif est de proposer une abstraction de mémoire commune.

# 2 Classification et modélisation des DSM

Nous nous intéressons dans ce chapitre aux différentes modélisations des mémoires partagées. La notion de mémoire partagée peut très simplement se définir par le fait que plusieurs processeurs (sites) peuvent lire et écrire dans un même espace mémoire.

Deux grandes directions se sont imposées pour la conception et la réalisation d'une mémoire partagée :

- Les recherches se sont dans un premier temps orientées vers des architectures matérielles à mémoire physiquement partagées. Plusieurs processeurs se partagent une même mémoire physique. On parle alors de machines multi-processeurs. Nous ne ferons qu'aborder cette voie dans ce chapitre.
- D'autres recherches se sont portées vers un partage au niveau logiciel. Chaque processeur possède sa propre mémoire physique (mémoire privée), et le logiciel se charge de donner l'illusion d'un espace de mémoire partagé. On parle alors de mémoire partagée distribuée ou répartie (MPR ou Distributed Shared Memory DSM).

Les travaux sur les DSM essaient, soit d'étendre les systèmes d'exploitation, soit de proposer de nouveaux environnements de programmation. Ceci n'est pas sans incidence sur la structuration de l'espace partagé. Ainsi, les systèmes d'exploitation proposerons un espace mémoire virtuelle distribué, alors que les environnements de programmation préféreront un découpage plus fin selon la taille des entités partagées.

L'implémentation efficace d'une DSM introduit le problème de cohérence des données, et par conséquent de la cohérence de la mémoire partagée. Différentes solutions ont été proposées, sous la forme de modèles de cohérence, qui se divisent en deux catégories. Dans la première, on retrouve des solutions similaires dans le principe à celles employées dans le partage physique. Dans la seconde, le programmeur "aide" le logiciel de la DSM en appelant une opération de synchronisation de la mémoire ou de l'entité partagée. Bien sûr, cette synchronisation vise à améliorer encore l'efficacité de la DSM.

Nous commencerons donc par présenter les deux directions possible pour la réalisation des DSM, en nous intéressant plus particulièrement à la partie logicielle. Nous étudierons ensuite l'influence de la granularité du partage, ce qui nous amènera au problème du faux partage, et aux solutions actuelles permettant de l'éviter. Puis nous présenterons les principaux modèles de cohérence. La dernière partie du chapitre sera consacrée aux interfaces de programmation des DSM.

## 2.1 Partage matériel ou partage logiciel

La notion de mémoire partagée diffère selon le type de la machine cible, machines multi-processeurs ou réseau de stations. Dans une machine multi-processeur, le partage est matériel, et l'on parle de mémoire partagée. Dans un réseau de station, le partage est logiciel, et l'on parle de mémoire partagée répartie.

### 2.1.1 Mémoire partagée, partage matériel (Hardware)

Une solution naïve pour réaliser le partage matériel d'une mémoire consiste à interconnecter plusieurs processeurs et la mémoire à l'aide d'un même bus d'adressage et de données (figure 2.1 a).

Un processeur voulant lire un mot mémoire place l'adresse sur le bus d'adressage, envoie un signal sur une ligne de contrôle, puis lit le contenu de ce mot venant de la mémoire sur le bus de données.

Un problème d'accès concurrents se pose quand deux processeurs accèdent simultanément à la mémoire. Dans ce cas, la logique de contrôle (circuits logiques) suspend l'exécution de l'un des processeurs jusqu'a ce que l'autre ait fini son accès. Comme en général, un processeur accède à la mémoire à chaque instruction, ce problème se pose constamment, ce qui entraîne une baisse significative de la vitesse des processeurs.

La solution retenue consiste à associer un cache mémoire à chaque processeur (figure 2.1 b). Ce cache contient une copie des données mémoire accédées le plus récemment. Lors d'un accès mémoire, le processeur regarde si la donnée demandée est déjà présente dans le cache. Si oui, la donnée du cache est utilisée immédiatement, si non, la donnée est transférée de la mémoire vers le cache et le processeur. Une telle machine est appelée une machine multi-processeurs.

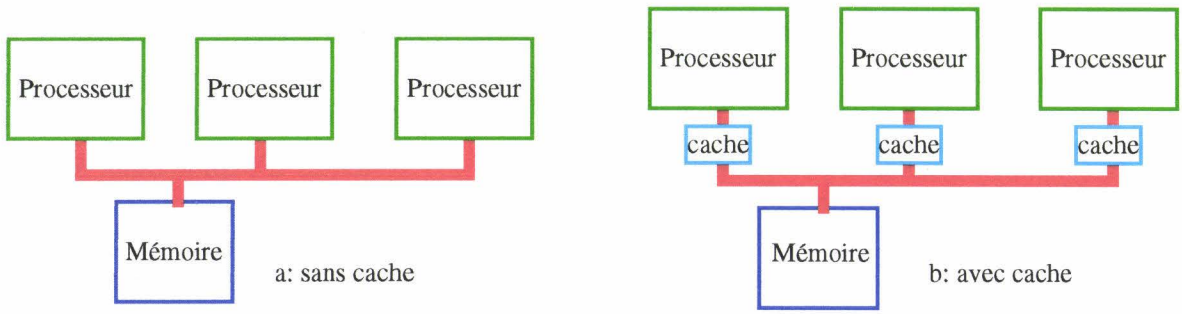



**figure 2.1** Mémoire partagée, partage matériel

L'ajout d'un cache introduit cependant un autre problème: celui de la cohérence des informations contenues dans le cache.

Des solutions matérielles ont été développées afin de maintenir la cohérence des caches, mais, en raison de la complexité des circuits ces solutions sont coûteuses et complexes. De plus, les limites actuelles de la technologie ne permettent pas d'interconnecter un très grand nombre de processeurs (actuellement une centaine) à l'aide d'un même bus d'adressage et de données.

### 2.1.2 Mémoire partagée répartie (DSM), partage logiciel

Dans une architecture distribuée sans mémoire physique commune mais dans laquelle chaque processeur possède sa propre mémoire, et communique avec les autres processeurs par échanges de messages, il est possible de réaliser par logiciel une mémoire partagée répartie (DSM). Cette DSM met à la disposition de chaque processeur un espace d'adressage partagé alors que l'espace mémoire global est physiquement distribué et éclaté.

La figure 2.2 illustre l'architecture d'une DSM. Chaque processeur (ou site) possède sa propre mémoire, et est relié au réseau de communication. La DSM, représentée par la zone grisée, est formée par les mémoires, qui sont distribuées sur les différents processeurs, le réseau de communication et le logiciel de la DSM.

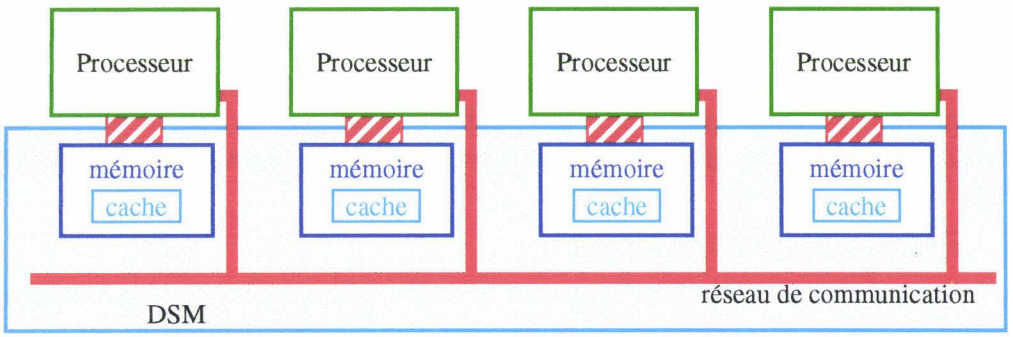



**figure 2.2** Mémoire partagée répartie (DSM)

Une implémentation de la DSM consiste à diviser l'espace de mémoire partagé et à répartir les morceaux dans des caches logiciels répartis entre les processeurs, un peu à l'image des caches mémoire des machines à mémoire physique commune vues précédemment. Ces caches contiennent généralement les morceaux accédés le plus récemment, et sont maintenus par le logiciel. L'accès par un processeur à une zone mémoire qu'il ne possède pas provoque un défaut, et le transfert sous le contrôle du logiciel de la zone requise. Le problème de la cohérence des caches se pose ici aussi, avec cette fois des solutions logicielles qui sont étudiées dans le paragraphe "Modèles et protocoles de cohérences", page 35.

Il en ressort que le logiciel est chargé des différentes fonctions suivantes:

- Découper et structurer l'espace partagé, ce qui détermine la granularité du partage.
- Détecter les accès à l'espace partagé, aussi bien en lecture qu'en écriture.
- Maintenir la cohérence de l'espace partagé. C'est à dire qu'il doit faire en sorte que chaque processeur "voit" le même contenu dans la mémoire partagée répartie.

Le maintien de la cohérence se fait naturellement par échange de messages entre les différents processeurs. Le surcoût introduit par le logiciel, et surtout le temps de propagation des messages font que le temps d'accès à la mémoire partagée est supérieur au temps d'accès à la mémoire locale.

### 2.1.3 Solutions hybrides

Une autre approche retenue dans des machines spécialisées comme la machine Alewife [Alewife95] développée au MIT, ou dans le projet WARPphos [Sieglin96], consiste à reprendre le principe des DSM, mais en utilisant des processeurs spécialisés prenant en charge une partie des fonctions du logiciel.

Ces processeurs se chargent de la détection des accès à l'espace partagé, et du transfert des zones mémoires. Le maintien de la cohérence reste en partie du domaine du logiciel, ce qui permet de l'adapter aux différents types de problèmes.

Dans la suite de cette thèse, nous ne considérerons que les mémoires partagées réparties entièrement réalisées par logiciel. De plus, nous ne parlerons plus de processeurs mais de processus, qui peuvent être vus comme des processeurs virtuels. Bien sûr, plusieurs processus situés sur un même site (une même machine) peuvent avoir de la mémoire physique commune, mais nous ne l'utilisons pas. Nous avons fait ce choix afin qu'il n'y ait pas de différence entre deux processus situés sur des sites différents, ou deux processus situés sur un même site. Nous considérons un processus comme une unité d'exécution individuelle communicant avec les autres processus à l'aide de messages. Ceci nous permet alors de confondre les termes de processus et de site. Un processus s'exécute sur un site, et nous considérons qu'un site contient un processus, même si dans la pratique il en contient plusieurs.

## 2.2 Granularité du partage

La granularité peut se définir à partir du mode de partage mis en oeuvre. Elle est déterminée par la taille de la zone mémoire transférée lors du maintien de la cohérence de la DSM. L'approche système privilégie le transfert d'une page, alors que l'approche programmatique préférera une granularité correspondant à la taille exacte de l'entité partagée, objet ou structure.

### 2.2.1 Pages, approche système

Dans les systèmes actuels, chaque processus peut accéder à un espace mémoire allant de zéro à la taille maximale adressable par le processeur. Cet espace mémoire est virtuel, car bien souvent la capacité de la mémoire physique est inférieure à la taille potentiellement adressable.

L'espace mémoire virtuel est alors divisé en pages dont seules quelques unes sont présentes dans la mémoire physique. Les autres sont stockées dans la mémoire de masse, en général les disques durs.

Le système d'exploitation prend en charge la gestion de la mémoire virtuelle, notamment la détection des accès à une page non présente, la projection d'une page dans l'espace du processus (mapping) ou l'échange de pages entre la mémoire physique et la mémoire de masse (swap). Cette gestion est transparente au programmeur qui ne voit qu'un espace mémoire (virtuel) linéaire.

Il parait alors naturel d'utiliser la gestion mémoire déjà existante, et de l'adapter afin de créer un espace mémoire virtuel partagé. Cet espace mémoire partagé est structuré en pages, et de nouveaux outils sont apportés afin de gérer le partage, outils restant homogènes avec ceux déjà existant.

Ceci permet de structurer l'espace de mémoire partagée de la même façon que l'espace mémoire virtuelle, et ainsi d'utiliser les outils de gestion déjà existant, comme la détection des accès. De nouveaux outils sont apportés afin de gérer le partage, outils restant homogènes avec ceux déjà existant.

Ce type d'approche est plutôt employé par les systèmes d'exploitation qui proposent alors un ensemble d'outils homogènes permettant de gérer l'espace mémoire, qu'il soit local ou partagé. C'est la solution retenue par la mémoire partagée répartie du micro-noyau Chorus [Ortega93].

Une page est une unité de partage assez grande (1K, 4K, 8K), dite à gros grain. Quand la taille des entités à partager est très petite comparée à celle d'une page, se pose le problème classique des DSM: le faux partage.

**Problème du faux partage**

Le problème du faux partage se pose quand différents processus accèdent à une même unité de partage contenant différents objets.

Imaginons que chaque processus accède, de préférence en écriture, à un objet distinct de celui accédé par les autres processus. Lors de l'accès, et afin d'éviter des problèmes de cohérence, l'unité de partage est transférée dans le processus. Si deux processus accèdent régulièrement à des objets distincts, l'unité de partage se met à faire le va et vient (ou ping pong) entre les processus au gré des accès.

C'est un problème de faux partage car les objets accédés n'ont pas de rapport entre eux. Si ils étaient placés dans des unités de partage différentes, il n'y aurait pas de va et vient.

Plus précisément, le problème du faux partage se pose quand différents objets sont rangés dans une seule unité de partage à gros grain, par exemple dans une même page, et que

chaque objet est accédé par un processus différent.

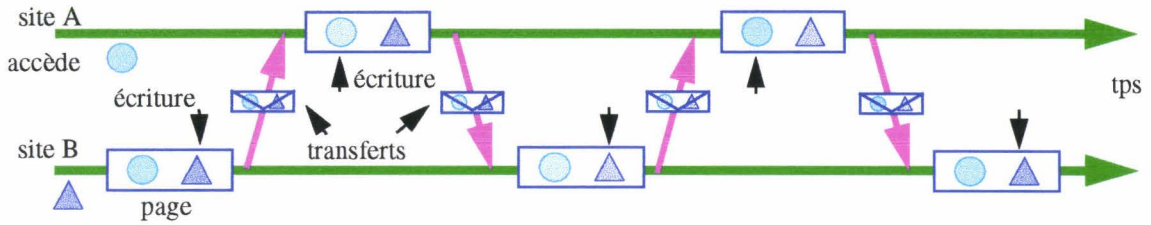



**figure 2.3** Problème du faux partage: ping-pong de la page

Le problème du faux partage peut être évité en plaçant les objets distincts accédés par différents sites dans des unités de partages distinctes. C'est ce que propose le découpage de l'espace partagé en objet ou structure.

### 2.2.2 Objets, approche programmation

Au sein d'une application, le programmeur manipule de préférence des objets ou des structures. Les environnements de programmation proposent alors une unité de partage plus fine, correspondant à la taille exacte de l'objet ou de la structure à partager.

Chaque entité à partager est alors indépendante des autres, et l'accès à deux entités différentes par des sites différents ne provoque plus le problème du faux partage.

Par contre, la détection des accès à un objet est rendue plus difficile. La solution couramment retenue est de demander au programmeur d'encadrer les accès à un objet partagé par l'appel de deux fonctions, l'une précisant le début de l'accès, l'autre la fin.

De même, la gestion de la DSM devient plus complexe. Il est en effet difficile de se servir de la pagination système, et la DSM doit bien souvent gérer elle-même l'occupation de l'espace partagé.

La plupart des DSM proposent un découpage à grain fin. Par exemple le modèle Linda [Carriero89] propose de partager des objets, appelés *tuple*, dans un sac (*tuple space*). Un *tuple* est équivalent à une structure C ou un record Pascal. Il est constitué de différents champs, chacun d'entre eux représentant une valeur d'un des types de base supporté par Linda. Pour le modèle C-Linda, les types de base sont les entiers, les nombres flottants ainsi que des types composés comme les tableaux ou les structures, mais pas d'autre tuple. La figure 2.4 donne des exemples de *tuples*.

```
( "abc", 2, 5 )
( "object name", "field 1", "field 2" )
( "min", 3 )
```

**figure 2.4** Exemple de *tuples* Linda

La tendance est au découpage de l'espace partagé selon la taille des objets à partager plutôt que selon les pages système. Ceci permet non seulement d'éviter le problème du faux partage, mais autorise aussi une meilleur gestion de l'espace partagé. En effet, il est alors possible de gérer les objets indépendamment les uns des autres.

## 2.3 Modèles et protocoles de cohérences

Le problème de la cohérence se pose dès lors que différents sites modifient une même partie de l'espace partagé. En effet, comment assurer que ces modifications soient visibles de tous les sites ?

Prenons l'exemple d'une DSM structurée en pages. Une implémentation simple et naïve consiste à répartir les pages sur les sites (figure 2.5 a) sans les dupliquer. Quand un site distant accède à une page qu'il ne possède pas, un mécanisme détecte l'absence de la page (erreur d'accès mémoire, déroutement du programme) et provoque son transfert vers le site demandeur. Ce mécanisme fonctionne, mais il présente un grave inconvénient: quand une page est accédée par plusieurs sites, elle se met à faire des allers et retours entre ces sites. Les sites passent alors plus de temps à transférer la page qu'à exécuter l'application.

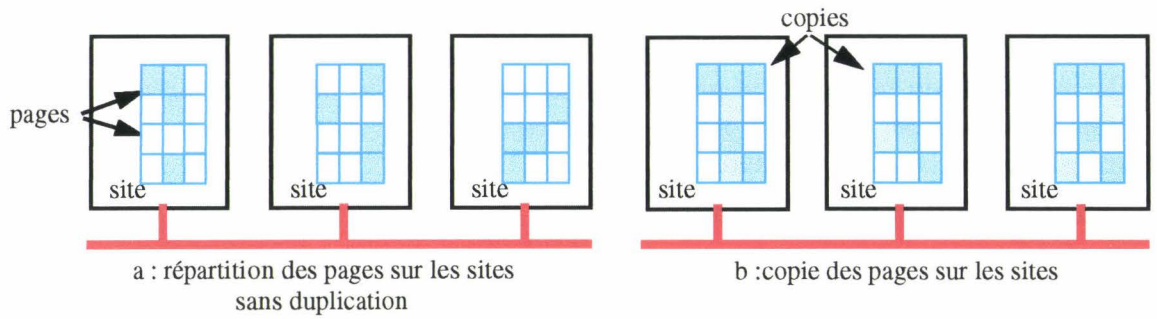



**figure 2.5** Répartition des pages sur les sites

Une autre implémentation consiste à améliorer la première solution en autorisant la duplication d'une page (figure 2.5 b). Une page sera copiée sur chaque site où elle est utilisée. Ainsi, l'accès à une page ne nécessite plus son transfert. Deux sites peuvent consulter (lire) une même page sans qu'il y ait d'erreur d'accès mémoire. Le problème survient quand un site décide de modifier (écrire) une page: les copies de la page ne sont plus à jour. Il faut alors déployer un mécanisme mettant à jour les copies afin de maintenir leur cohérence.

La aussi, il existe une solution naïve pour implémenter ce mécanisme: à chaque modification de la page, un message est envoyé à tous les sites possédant une copie. Ainsi, la modification est propagée. Cette solution présente cependant plusieurs inconvénients:

- Tout d'abord, la modification fréquente d'une zone de mémoire partagée génère l'envoi de messages à chaque modification.
- Ensuite, la modification peut être propagée vers des sites n'utilisant pas ou peu cette page, entraînant autant de messages inutiles.
- Enfin, quelle modification prendre en compte lors d'écritures simultanées d'une même page par plusieurs processus ?

Finalement, le maintien de la cohérence n'est pas aussi simple que cela. De plus, la gestion de cette cohérence entraîne un surcoût (overhead) non négligeable lors de l'exécution du programme. En effet, les accès en mémoire partagée génèrent des échanges de messages, et envoyer un message sur le réseau est toujours plus long que d'effectuer un accès en mémoire locale.

L'objectif des premiers modèles de cohérence, comme la cohérence séquentielle, était de fournir une mémoire dont le comportement était proche de celui d'une mémoire centralisée : Une écriture réalisée dans la mémoire est immédiatement visible par tous les sites, et toute lecture ultérieure retourne la nouvelle valeur.

Les modèles de cohérence plus récents, comme la cohérence relâchée ou la cohérence par entrée, ont pour objectif de diminuer la quantité d'information échangée. Pour cela, ils relâchent les contraintes pesant sur le comportement de la mémoire et introduisent une opération de synchronisation de cette même mémoire. Cette opération permet au protocole de

mieux exploiter les types d'accès à la mémoire

Il est nécessaire, à notre avis, de distinguer le modèle de cohérence, qui est la façon dont réagit la mémoire quand on l'utilise, du protocole pour maintenir cette cohérence, qui est la manière dont est implémentée la mémoire partagée. Les deux sont liés: le modèle induit le protocole, et le protocole implémente le modèle. Mais il faut noter qu'un modèle donné peut avoir plusieurs implémentations.

Nous allons expliquer les principaux modèles de cohérence, d'abord les modèles sans opération de synchronisation, puis les modèles avec opérations de synchronisation.

### 2.3.1 Modèles sans synchronisation

#### Cohérence séquentielle (Sequential Consistency)

Le modèle de cohérence séquentielle a été proposé (initialement) par Lamport [Lamport79] pour des machines multiprocesseurs telles que celles étudiées précédemment (page 32). Ce modèle s'adapte aux systèmes distribués, comme le montre l'implémentation d'IVY [Li89].

Dans le modèle de cohérence séquentielle, le comportement de la mémoire partagée est défini comme suit:

> *Le résultat de toute exécution est le même que celui où les opérations mémoire de tous les processus seraient exécutées dans un ordre séquentiel donné. Dans cet ordre séquentiel, les opérations de chaque processus individuel apparaissent dans l'ordre spécifié par leur programme.*

La séquence, ou ordre séquentiel, est une suite d'opérations atomiques contenant l'ensemble des opérations mémoire effectuées par chaque processus d'une application. Elle peut être obtenue en prenant toutes les opérations mémoire de tous les processus, et en les entrelaçant dans une suite ou séquence (figure 2.6). La définition spécifie qu'il est possible (et même obligatoire) de retrouver dans cette séquence les opérations effectuées par un processus, et de les retrouver dans l'ordre où elles ont été exécutées par le processus.

Pour une application donnée, il existe plusieurs séquences ou entrelacements valides des opérations. La figure 2.6 donne deux séquences possibles de la même application. Chacune de ces séquences ne conduit pas obligatoirement l'application au même résultat (comparer les affichages des exécutions). Mais n'importe laquelle de ces séquences conduit à un comportement correct de l'application. C'est à dire que le résultat de l'application peut être considéré comme juste.

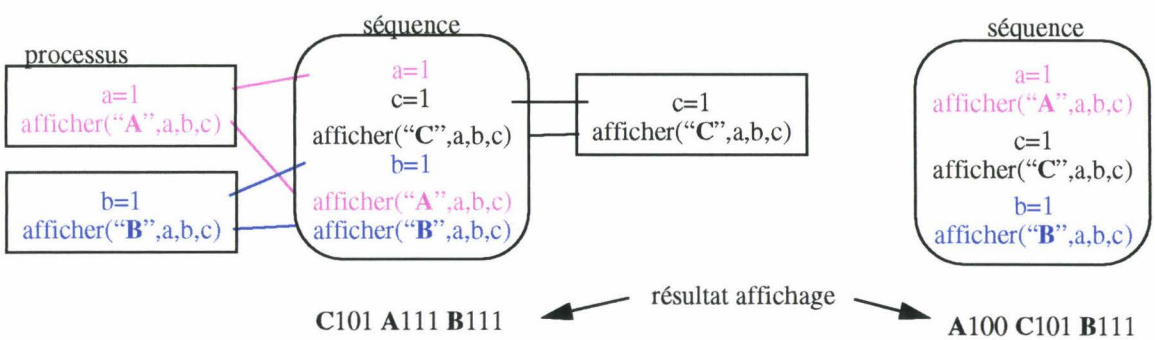



**figure 2.6** Séquences d'exécution possibles pour une application de trois processus

Une mémoire partagée où deux processus d'une application voient deux entrelacements différents n'est pas une mémoire à cohérence séquentielle.

Il est à noter que rien n'est dit à propos du temps: il n'y a pas de référence à "la plus récente écriture", ce qui compte, c'est que tous les processus voient la même suite d'opérations sur la mémoire. La cohérence séquentielle ne garantit pas de voir la toute dernière modification, mais elle garantit que tous les processus verront les opérations sur la mémoire dans le même ordre.

Ainsi la séquence de gauche de la figure 2.6 peut être obtenue indifféremment par l'histogramme des opérations (diagramme temporel) de la figure 2.7 ou par celui de la figure 2.8. Dans le premier histogramme, l'ordre des opérations est le même que dans la séquence, alors que dans le second, la modification de *b* n'est pas propagée immédiatement, et n'est donc pas visible du processus A. Bien sûr, la modification est visible par le processus B dès sa première lecture, mais dans la séquence cette lecture intervient après la lecture de *b* par le processus A.

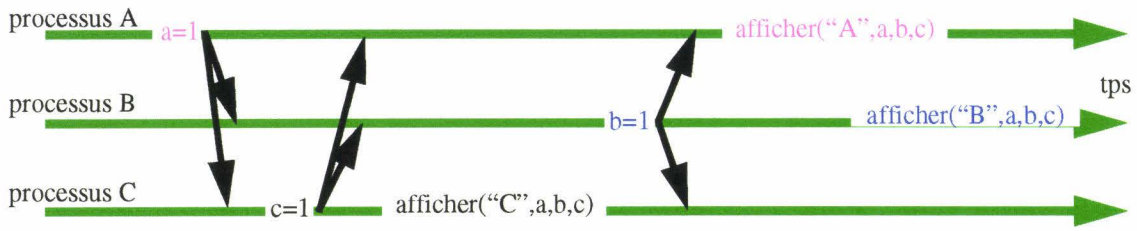



**figure 2.7** Histogramme de la première séquence de la figure 2.6

La différence avec la "cohérence idéale" est que la modification de *b* dans la figure 2.8 n'est pas immédiatement visible par le processus A. Dans une cohérence forte "idéale" cette modification aurait été vue immédiatement par le processus A et le processus B.

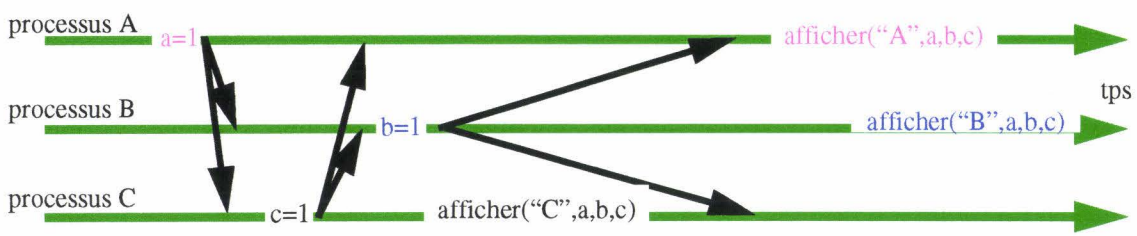



**figure 2.8** Un autre histogramme possible de la première séquence de la figure 2.6

Le comportement d'une application utilisant une DSM à cohérence séquentielle est le même que celui obtenu si les différents processus de l'application sont exécutés en temps partagé sur un seul processeur. Si les processus communiquent uniquement par la mémoire partagée, ils ne peuvent pas déterminer par le biais des interactions avec la mémoire qu'ils n'accèdent pas une mémoire centralisée.

**Cohérence causale (Causal consistency)**

La cohérence causale a été proposée par Hutto et Ahamad [Hutto90] comme alternative à la cohérence séquentielle. Ils ont observé que seules les opérations étant en relation causale devaient apparaître dans le même ordre pour tous les processus. Ainsi, il leur a été possible de relâcher les contraintes imposées sur les opérations d'écriture.

Deux événements sont en relation causale si un événement *B* est causé ou influencé par un événement *A* le précédant. Prenons comme exemple la lecture d'une variable *x* suivie par l'écriture d'une variable *y* (figure 2.9). Si l'écriture de *y* dépend de la valeur de *x* (exemple : $y=x+1$), *x* et *y* sont en relation causale. Quand deux événements ne sont pas en relation causale, comme par exemple deux écritures simultanées dans deux processus différents, ils sont dits concurrents.


**figure 2.9** Relations causales et écritures concurrentes

Dans le cas d'une mémoire partagée, et si on reprend la définition de la séquence d'opérations définie plus haut, la cohérence causale assure que chaque site observe dans sa séquence l'opération de lecture avant l'opération d'écriture.

Une mémoire à cohérence causale doit respecter les conditions suivantes:

- *Les écritures potentiellement en relation causale doivent être vues par tous les processus dans le même ordre.*
- *Les écritures concurrentes peuvent être vues dans des ordres différents sur les différentes machines.*

Une écriture est vue par un processus à partir du moment où une opération de lecture donne la valeur écrite. Entre l'opération d'écriture et l'opération de lecture, la valeur est théoriquement modifiée, mais elle n'est prise en compte (consultée) par personne. L'instant exact où intervient la modification importe peu, du moment qu'elle se passe avant la lecture.

Dans ce modèle de cohérence, chaque processus peut donc observer les opérations d'écriture concurrentes dans un ordre différent, mais tous doivent observer le même ordre sur les opérations d'écriture ayant potentiellement une relation causale.

La figure 2.10 montre les séquences des deux processus de l'histogramme précédent. La cohérence est causale, car il existe une séquence préservant l'ordre causal des écritures pour chaque processus. Remarquez que dans chaque séquence l'opération *x=1* précède l'opération *y=x+1*. Elle n'est pas séquentielle, car chaque processus voit une séquence d'opération différente, notamment au niveau des opérations *a=1* et *b=1*.

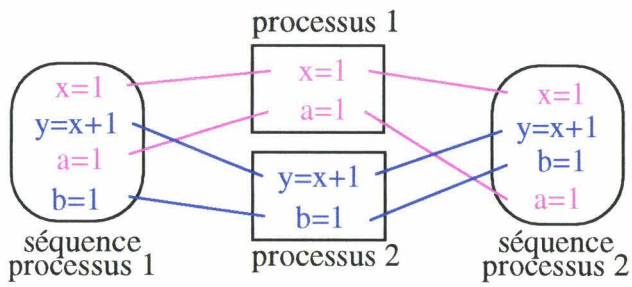



**figure 2.10** Cohérence causale

### Cohérence PRAM (PRAM Consistency)

La cohérence PRAM proposée par Lipton et Sandberg [Lipton88], relâche la contrainte imposée sur les écritures par la cohérence causale. Une mémoire a une cohérence PRAM si elle répond à la condition suivante:

*La séquence des opérations d'écriture effectuées par un processus doit être vue par chaque processus dans l'ordre où ces opérations ont été effectuées, mais l'ensemble des opérations d'écritures effectuées par les différents processus peut être vu par chaque processus dans des ordres différents.*

Autrement dit, chaque processus peut observer une séquence d'opérations différente.

Mais, dans cette séquence, les opérations d'écriture d'un processus sont observées dans l'ordre où elles ont été effectuées.

La figure 2.11 représente trois processus d'une application, et les séquences d'opérations vues par ces processus. La séquence d'un processus n'a pas besoin de faire apparaître les opérations de lecture des autres processus, car ce processus n'est pas (directement) informé de ces opérations de lecture.

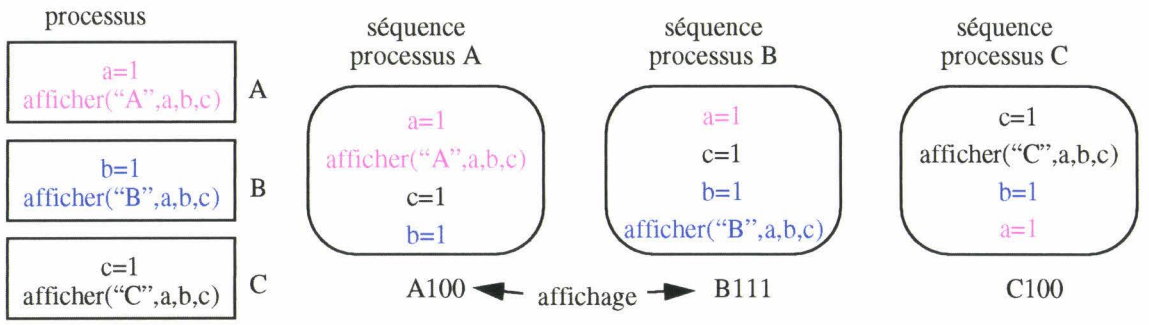



**figure 2.11** Cohérence PRAM: séquence vue par chaque processus

Le nom de cohérence PRAM vient de "Pipelined RAM", car ce modèle permet de "pipeliner" les opérations d'écriture d'un processus. C'est à dire que le processus n'a pas besoin d'attendre que l'opération d'écriture précédente soit terminée pour pouvoir lancer l'opération suivante.

Il existe une variante appelée *Processor Consistency* proposée par Goodman [Goodman89]. Cette variante ajoute une contrainte supplémentaire à la cohérence PRAM: toutes les écritures effectuées sur une variable donnée sont observées dans le même ordre par tous les processus. Les écritures effectuées sur des variables différentes peuvent être observées dans des ordres différents.

### 2.3.2 Modèles avec synchronisation

Il est possible d'aller encore plus loin dans l'allégement des contraintes pesant sur les modèles de maintien de la cohérence. Il existe en effet des situations où un site n'a pas besoin de connaître toutes les modifications intervenues sur un autre site.

Prenons l'exemple (figure 2.12) dans lequel une application est constituée de plusieurs processus s'exécutant sur différents sites. Tous les processus accèdent à deux variables partagées $x$ et $y$. L'un des processus effectue une série de calcul sur la variable $x$ (boucle), puis modifie la valeur de $y$. Les autres processus attendent que la variable $y$ soit positionnée (différente de 0) avant de continuer leur exécution, et de lire $x$.

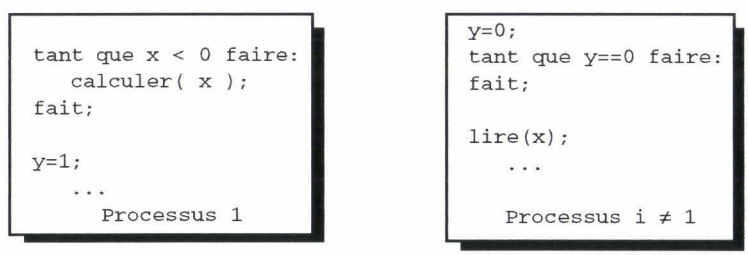



**figure 2.12** Boucle de calcul et variable partagée

Si l'application est réalisée avec un modèle de cohérence tel que ceux vus précédemment, chaque modification de $x$ est propagée à l'ensemble des processus. En étudiant le programme, on s'aperçoit que seule la dernière valeur de $x$ est utile. Plusieurs

synchronisations de la mémoire ont été faites inutilement.

Une solution moins coûteuse en quantité d'information échangée est de laisser le processus finir ses calculs, puis propager le résultat final, sans propager les résultats intermédiaires.

C'est pour prendre en compte des situations de ce genre que de nouveaux modèles de cohérence ont été développés. Ils introduisent une opération de synchronisation permettant au programmeur d'orienter le protocole de maintien de la cohérence.

Ces modèles distinguent deux types d'accès: les accès aux variables partagées, et les accès aux variables de synchronisation. Ces derniers permettent de synchroniser la mémoire après un certain nombre d'opérations effectuées sur les variables partagées. Lors d'une synchronisation, toutes les modifications sont propagées et la mémoire partagée se retrouve dans un état défini par le modèle de cohérence.

Entre deux synchronisations, les modifications effectuées par un processus ne sont pas visibles des autres processus.

Nous allons décrire plusieurs modèles de cohérence avec opérations de synchronisations. Tout d'abord la cohérence faible, puis la cohérence relâchée avec ses variantes: la cohérence relâchée impatiente et la cohérence relâchée paresseuse, et enfin la cohérence par entrée.

**Cohérence faible (Weak Consistency)**

Le modèle de la cohérence faible [Dubois86] [Bisiani90] introduit une variable de synchronisation qui est appelée explicitement par le programmeur avant et après une opération dans la mémoire partagée.

Le modèle de la cohérence faible est basé sur trois propriétés:

- *Les accès aux variables de synchronisation présentent une cohérence séquentielle.*
- *L'accès à une variable de synchronisation ne peut pas se terminer tant que les opérations d'écritures et de lectures ne sont pas terminées sur tous les sites.*
- *Il ne peut pas y avoir d'accès en lecture ou en écriture de la mémoire partagée tant que tous les accès aux variables de synchronisation ne sont pas terminés.*

La première propriété dit que le même ordre est observé par tous les processus sur les accès aux variables de synchronisation. Les deux suivantes garantissent que lorsque un processus effectue une synchronisation, il propage les modifications qu'il a effectuée, et il est informé des modifications intervenues auparavant dans les autres processus.

La figure 2.13 illustre la cohérence faible. On remarque que les accès aux variables de synchronisation se terminent pratiquement en même temps, et qu'il est possible à plusieurs processus d'accéder simultanément à la mémoire partagée.

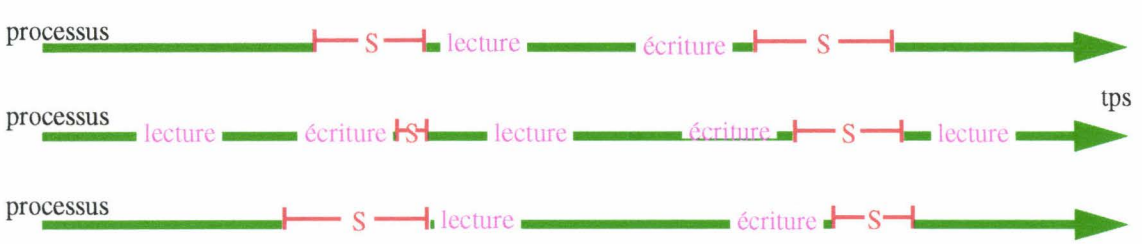



**figure 2.13** Cohérence faible

Ce modèle de cohérence permet de regrouper les modifications intervenues dans la mémoire partagée, et de les diffuser en un seul message lors de l'opération de synchronisation.

Mais la synchronisation nécessite d'autres messages afin de respecter les deux dernières propriétés.

Dans la cohérence faible, aucune distinction n'existe entre le commencement d'un accès en mémoire et la fin de cet accès. Le protocole de maintien de la cohérence doit donc effectuer les actions requises pour les deux cas: il doit s'assurer que toutes les écritures sont terminées, et que toutes les modifications des autres processus ont été prise en compte.

**Cohérence relâchée (Release Consistency)**

Le modèle de cohérence relâchée (aussi traduit par cohérence de libération) utilise deux variables de synchronisation, *acquire* et *release*, distinguant le début et la fin d'un accès en mémoire partagée. Il autorise une meilleure gestion de la mémoire partagée que ne le permet la cohérence faible.

Ces deux variables de synchronisation peuvent être comparées à celle d'une section critique réalisée avec un sémaphore. Un processus voulant accéder à la mémoire partagée effectue un *acquire*. Il réalise son ou ses accès, puis quand il a fini, il effectue un *release* (figure 2.14).

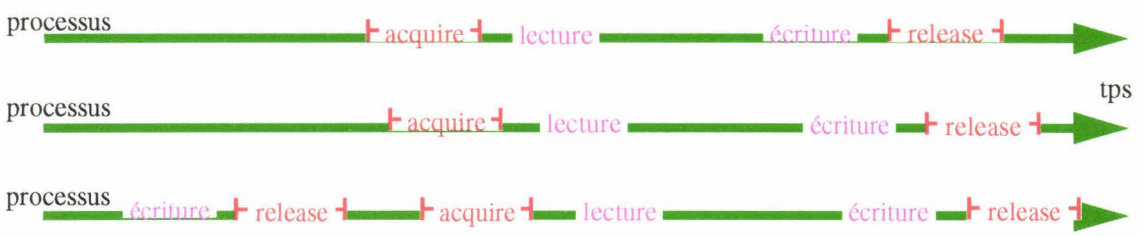



**figure 2.14** Cohérence relâchée

Dans le modèle de la cohérence relâchée, les accès aux variables de synchronisation à l'aide des opérations *acquire* et *release* sont appelés *accès spéciaux*. Les accès aux variables partagées sont appelés *accès ordinaires*. Une mémoire présente une cohérence relâchée si elle remplit les conditions suivantes [Gharachorloo90]:

- *Les accès spéciaux présentent une cohérence séquentielle.*
- *Avant qu'un accès ordinaire ne puisse-t- être accompli en respect avec les autres processus, toutes les acquisitions (acquire) précédentes doivent être terminées.*
- *Avant qu'un relâchement (release) ne puisse être accompli en respect avec les autres processus, tous les accès ordinaire (lectures et écritures) précédents doivent être terminés.*

La première condition assure que toutes les variables de synchronisations présentent une cohérence séquentielle. Tous les processus observent alors le même ordre sur les accès à ces variables de synchronisation. Les deux dernières conditions garantissent que lorsque un processus effectue un *acquire*, il verra les modifications effectuées avant les *release* précédent. De plus, quand un processus effectue un *release*, les modifications qu'il a faites seront vues par tout processus effectuant un *acquire* ultérieur.

Il existe deux variantes du modèle de cohérence relâchée appelées cohérence relâchée impatiente (*eager release consistency*) et cohérence relâchée paresseuse (*lazy release consistency*). Ces variantes se différencient par le protocole utilisé.

Le modèle de cohérence relâchée impatiente a été implémenté dans Munin [Carter91]. Il consiste à propager les modifications intervenues entre un *acquire* et un *release* au moment du *release*. Toutes les modifications intervenues entre ces deux opérations sont envoyées aux

copies en un seul message. Mais ce message peut être envoyé à des processus qui ne se serviront pas de la modification.

Dans le modèle de cohérence relâchée paresseuse, utilisé dans [Amza95], les modifications intervenues sur un site sont propagées au moment de l'*acquire*. Le processus acquéreur demande les dernières modifications intervenues. Ainsi, les modifications sont transmises uniquement aux processus utilisant la mémoire partagée, et uniquement au moment où ils en ont besoin. La cohérence relâchée paresseuse est difficile à mettre en oeuvre. Cependant, Keleher [Keleher95] montre que le surcoût nécessaire au déroulement du protocole est largement contrebalancé par la diminution du nombre de messages et de la quantité d'information échangée.

**Cohérence par entrée (Entry Consistency)**

Les modèles précédents assurent la cohérence de la DSM sans distinguer les variables partagées. Le modèle de cohérence par entrée (aussi traduit par cohérence en entrée), utilisé dans le modèle Midway [Bershad91] [Bershad93], propose d'associer une variable de synchronisation à une ou plusieurs variables partagées. Ainsi, lors des opérations de synchronisation (*acquire* et *release*), seules les variables partagées associées sont à prendre en compte. La gestion de la mémoire partagée est faite plus finement, ce qui permet de réduire la quantité d'information échangée. De plus, ce modèle de cohérence autorise l'existence de plusieurs "sections critiques" mettant en jeu des variables distinctes. Ces sections critiques peuvent alors être parcourues simultanément par deux processus différents.

L'inconvénient est que le programmeur doit explicitement associer les variables partagées aux variables de synchronisation, ce qui augmente le risque d'erreur.

### 2.3.3 Conclusion

Un modèle de cohérence garantit au programmeur un état précis de la mémoire.

Toutes les DSM partagent le même inconvénient : les temps d'accès à l'information partagée est plus long que le temps d'accès à une information non partagée. Ceci est dû à l'échange d'information nécessaire au maintien de la cohérence.

Une cohérence forte a un comportement proche de celui d'une mémoire centralisée. Son utilisation ne présente donc pas de grande difficulté pour le programmeur. Malheureusement, les temps d'accès importants qu'elle présente réduisent les performances de l'application.

Les modèles de cohérence plus faible donnent de meilleurs temps d'accès. Ces améliorations sont obtenues en diminuant le nombre de messages et la quantité d'information échangée. Cette diminution est elle même obtenue en relâchant les contraintes pesant sur la DSM: par exemple en ne garantissant l'état de la DSM qu'en certains points de synchronisation.

Ces modèles forcent la cohérence sur un groupe d'opérations, et non pas sur une lecture ou une écriture individuelle. Ils sont plus efficaces quand les accès isolés aux variables sont rares, et que les accès se font par rafale: beaucoup d'accès sur une courte période, puis plus rien sur une plus longue période.

La contrepartie est que le programmeur doit explicitement indiquer les opérations d'accès à la mémoire partagée, aussi bien le début et la fin d'une série d'opérations (synchronisation) que les accès en lecture ou en écriture. Ces annotations explicites augmentent le risque d'erreur de la part du programmeur.

## 2.4 Interface utilisateur

L'interface utilisateur permet la programmation du modèle de mémoire partagée Cette interface peut être un nouveau langage intégrant les nouvelles fonctionnalités, ou seulement une extension d'un langage déjà existant, ou bien encore une librairie contenant les nouvelles fonctions.

La conception d'un nouveau langage est difficile, et ne se justifie que lorsque l'on veut fournir des concepts de programmation très différents de ceux qui existent déjà. De plus, un nouveau langage implique son apprentissage, et l'impossibilité de réutiliser des programmes écrits pour d'autres langages. Cette approche est peu utilisée, mais il en existe tous de même des exemples comme le langage Orca [Bal92].

La seconde approche consiste à prendre un langage déjà existant, et à lui ajouter des éléments de langage (mots clefs) afin que le programmeur puisse utiliser les nouvelles fonctionnalités. Il est alors possible d'effectuer des contrôles sur la sémantique, de générer des actions en fonction des accès à la mémoire partagée, et aussi de détecter des erreurs dans ces accès. L'inconvénient est qu'il faut adjoindre un préprocesseur qui se charge de transformer le programme contenant les mots clef en un programme plus compréhensible par le compilateur de base. Cette solution est utilisée dans des modèles tels que Midway [Bershad93] et Munin [Carter91].

La dernière approche, qui est aussi la plus employée par les modèles de mémoire partagée, consiste à fournir une librairie contenant les nouvelles fonctionnalités. Le programmeur manipule alors la mémoire partagée à travers un certain nombre de fonctions. Cette solution est la plus simple à mettre en oeuvre: Les nouvelles fonctionnalités prennent la forme de fonctions écrites dans le langage hôte. Il n'est pas nécessaire de réaliser un compilateur ou un préprocesseur traduisant les nouveaux éléments de langage. Elle permet aussi de réutiliser des programmes déjà existant. L'inconvénient de cette solution est qu'elle oblige le programmeur à appeler explicitement les fonctions de la mémoire partagée, ce qui rend l'utilisation de cette dernière moins transparente.

Qu'importe l'approche retenue, l'utilisation de la mémoire partagée commence par la *création et l'identification des objets partagés*. Ensuite, il faut *créer le partage* afin que les objets partagés soient connus de tous les processus les utilisant. Dans certains modèles, cette création nécessite le *choix de la cohérence ou du protocole* utilisé. Enfin, l'*accès aux objets* se fait à travers des mécanismes garantissant la cohérence de la mémoire partagée.

### 2.4.1 Déclaration et identification des objets partagés

Dans tous les langages de programmation, les variables d'un programme sont désignées par un nom symbolique. Ce nom permet au programmeur de manipuler aisément la variable. Dans une application composée de plusieurs processus, chaque processus détient des variables qui lui sont locales, et des variables partagées qui sont globales à l'application. Les variables locales sont manipulées avec des noms ayant uniquement une signification locale, tandis qu'une variable partagée est manipulée avec un nom global désignant la même variable dans tous les processus de l'application. La déclaration et la dénomination des variables partagées diffèrent d'un modèle de mémoire partagée à l'autre.

Une première approche permet de déclarer de façon globale les variables partagées. Cette déclaration consiste à préciser le type et le nom symbolique de la variable, à spécifier qu'elle est partagée, en ajoutant par exemple le mot réservé *shared* et éventuellement le mode de partage comme dans Munin (figure 2.15). Le nom symbolique est global à l'application et permet de manipuler la variable partagée à l'aide du même nom dans chaque processus.

Cette approche nécessite soit un nouveau langage, comme pour Orca (page 51) soit un

préprocesseur, comme dans Munin, afin de prendre en compte les nouveaux éléments de langage.

```
shared read_only int input1[N][N];
shared read_only int input2[N][N];
shared result int output[N][N];
```

**figure 2.15** Déclarations des variables partagées avec Munin (matrices pour effectuer C=A*B)

Dans ces modèles, les variables partagées sont déclarées statiquement dans le programme, c'est à dire avant la compilation. Le compilateur ou le préprocesseur connaît exactement les variables partagées avant l'exécution d'une application. Il n'est pas possible de créer des variables partagées au cours de l'exécution de l'application.

Les langages actuels permettent de manipuler des noms symboliques ayant une signification locale à un processus. Ils ne permettent pas de manipuler des noms globaux à une application composée de plusieurs processus. Par conséquent, les modèles de mémoire partagée dont l'interface prend la forme d'une librairie de fonctions ne permettent pas d'associer directement une variable partagée à un nom global. Au lieu de cela, il faut d'une part déclarer localement au processus la variable qui sera partagée, en précisant son type, et son *nom local,* et d'autre part déclarer le *nom global.* L'association *nom local - nom global* se fera explicitement lors de la phase de création du partage qui sera décrite plus tard (page 46).

La déclaration locale est similaire à la déclaration d'une variable non partagée (figure 2.16). Selon les modèles, il faut déclarer soit la variable elle-même comme pour Adsmith 1.0 ou Phosphorus, soit déclarer une référence sur cette variable comme pour Crl. Dans ce dernier cas, l'allocation de la mémoire se fera lors de la phase de création du partage.

Le nom global prend le plus souvent la forme d'un identificateur numérique entier, et quelque fois celle d'une chaîne de caractères. Cette dernière forme étant la plus agréable à utiliser pour le programmeur.

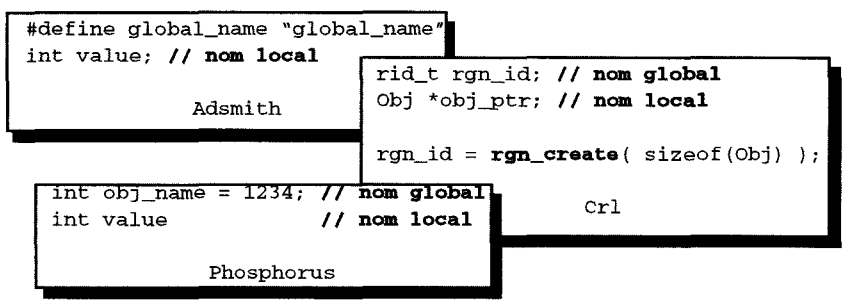



**figure 2.16** Déclaration des variables partagées pour Adsmith, Crl, Phosphorus

Dans le modèle Crl, le programmeur ne peut pas spécifier le nom global. Celui-ci est choisi par le système (*rgn_create()*), et doit être propagé à l'ensemble des processus de l'application. Pour cela, Crl propose des primitives de communication globale servant aussi à la synchronisation (Voir *Synchronisation inter-processus*, page 53).

Une alternative utilisant les noms locaux comme noms globaux est proposée par le modèle TreadMark [Amza95] (figure 2.17). Dans ce modèle, tous les processus sont identiques (SPMD). Cette condition permet de désigner une variable partagée à l'aide du même nom dans chacun des processus de l'application. Ces variables partagées sont déclarées sous la forme de références, puis elles sont allouées dans le même ordre dans chaque processus. La primitive d'allocation ne nécessite pas de nom global, l'association correcte entre le nom local et la

variable partagée est garantie par l'ordre des allocations.

```
Obj *obj1_ptr;
Obj *obj2_ptr;

obj1_ptr = Tmk_malloc( sizeof(Obj) );
obj2_ptr = Tmk_malloc( sizeof(Obj) );

obj2->x = 0;
obj1->x = 0;
```

**figure 2.17** Déclaration et création des variables partagées avec TreadMark

Cette méthode d'identification présente l'inconvénient d'obliger à avoir les mêmes processus, et permet difficilement la création dynamique de variables partagées.

### 2.4.2 Création du partage

La création du partage est la phase consistant à rendre opérationnel un objet partagé. Un processus ayant effectué cette phase peut accéder à l'objet partagé. Le processus crée l'objet à l'aide de son nom global, et l'opération retourne la représentation locale permettant d'y accéder.

Dans les modèles proposant une extension du langage par de nouveaux mots clef, la phase de création est prise en charge par le système lors de l'initialisation de l'application. La création du partage est alors transparente à l'utilisateur.

Dans les autres cas, la création du partage doit être faite avant toute utilisation de l'objet. Cette création comporte deux phases:

- Une phase de création proprement dite, effectuée par un des processus. Cette phase permet au système de créer l'objet ainsi que les structures associées nécessaires à sa gestion.
- Un phase de recherche de l'objet, effectuée par les autres processus. Cette phase permet à un processus de retrouver un objet créé par ailleurs, et de l'attacher afin de pouvoir y accéder.

Dans les modèles TreadMark (figure 2.17) et Phosphorus (figure 2.18), la phase de création et la phase de recherche sont effectuées par la même primitive. Cette primitive crée la variable uniquement si elle n'existe pas encore.

```
adsmobj obj_desc;
obj_desc = adsm_declare( global_name, "int 1", 1, ADSM_DEFAULT );
  ...
obj_desc = adsm_find( global_name );
```

Adsmith

```
rgn_id = rgn_create( sizeof(Obj) );
  ...
obj_ptr = rgn_map( rgn_id );
```

Crl

```
phos_declare( obj_name, INT, 1, CONVENTIONAL);
```

Phosphorus

**figure 2.18** Création du partage dans Adsmith, Crl et Phosphorus

Dans le modèle Crl, les deux primitives sont distinctes. Un processus père crée la zone mémoire (région) qui contiendra la variable (*rgn_create()*), puis crée les autres processus. Tous les processus doivent rechercher la zone mémoire (*rgn_map()*) avant d'utiliser la variable.

Adsmith, quant à lui, propose les deux solutions: une primitive de création/recherche, et

une primitive de recherche uniquement. Ces primitives retournent un descripteur de l'objet partagé, descripteur qui permettra ultérieurement d'accéder à l'objet, et qu'il ne faut pas oublier de déclarer.

### 2.4.3 Choix de la cohérence ou du protocole

Etant donné qu'il n'existe pas de cohérence avec un protocole convenant parfaitement à toutes les situations de partage possible, des modèles comme Munin, Phosphorus ou Adsmith proposent plusieurs types de cohérences ou des variantes de protocole. Le choix de la cohérence ou du protocole dépend des accès qui seront effectués sur la variable.

Ainsi le système peut tirer parti du fait qu'une variable est constante (initialisée puis uniquement lue). De même, pour une variable dont la lecture est toujours suivie d'une écriture, le système peut diminuer significativement le nombre d'échanges de messages en ne propageant pas les modifications, mais en migrant la variable d'un processeur à l'autre.

Munin propose sept protocoles différents (figure 2.19), adaptés à différents cas de partage, contre six pour Phosphorus. Adsmith, quant à lui, permet au programmeur d'orienter le protocole en donnant des indications sur les méthodes à employer. Adsmith propose aussi deux cohérences : cohérence stricte et cohérence relâchée.

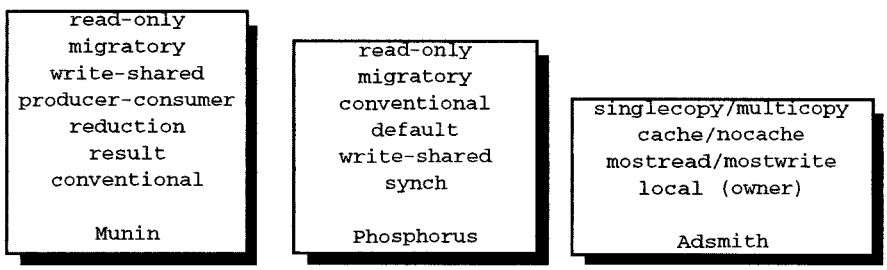



**figure 2.19** Cohérences et protocoles proposés par Munin, Phosphorus et Adsmith

Préciser les types d'accès, ou indiquer le type de cohérence ou de protocole permet d'orienter le système afin d'optimiser la gestion de la mémoire partagée. Si pour certaines variables le choix est trivial (constante, résultat), pour d'autres il est plus difficile (migratoire, conventionnel). Pourtant, ce choix est critique : un mauvais choix peut entraîner une gestion inefficace de la mémoire partagée, et une dégradation des performances de l'application.

Dans les modèles actuels, ce choix se fait à la création de la variable. Il ne peut pas être modifié au cours de l'application, pour par exemple l'adapter à de nouvelles situations.

### 2.4.4 Accès aux objets partagés

L'accès à un objet partagé se fait à l'aide du nom local déclaré par le programmeur. Mais cet accès ne se fait pas directement. Il doit être accompagné d'appels de fonctions destinés à garantir la cohérence de la mémoire partagée. Deux méthodes prédominent: la copie de l'objet, et la section critique.

#### Accès par copie

La copie consiste à rendre la vue locale de l'objet partagé cohérente avec la vue qu'en ont les autres processus. Il existe deux fonctions différentes, selon que l'objet est lu ou écrit: la première est appelée avant la lecture alors que la seconde est appelée après l'écriture (figure

2.20).

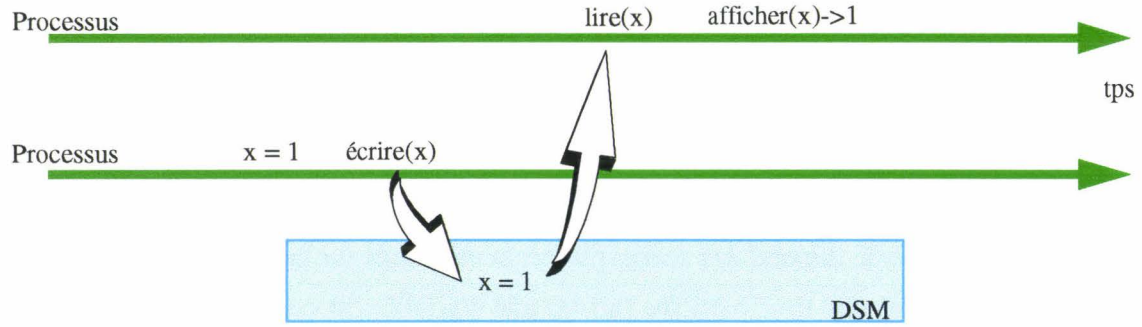



**figure 2.20** Accès par copie

Prenons l'exemple du modèle Linda [Carriero89] où la mémoire est considérée comme un sac (tuple space) contenant les objets partagés. Un processus voulant partager un objet le range dans le sac (*out()*), où un autre processus peut aller le retirer (*in()*) (figure 2.21). Les accès à l'objet se font donc par des fonctions copiant l'objet du ou vers le sac. Avec Linda, la recherche des objets partagés se fait de manière associative. Ainsi, l'appel à *in( "abc", ?value)* provoque la recherche du tuple ayant comme premier paramètre *"abc"*, et comme second paramètre un entier qui sera copié dans la variable *value*.

Les modèles Adsmith et Phosphorus utilisent aussi un accès par copie aux variables partagées. Ici, la recherche de la variable se fait avec le descripteur ou le nom de l'objet.

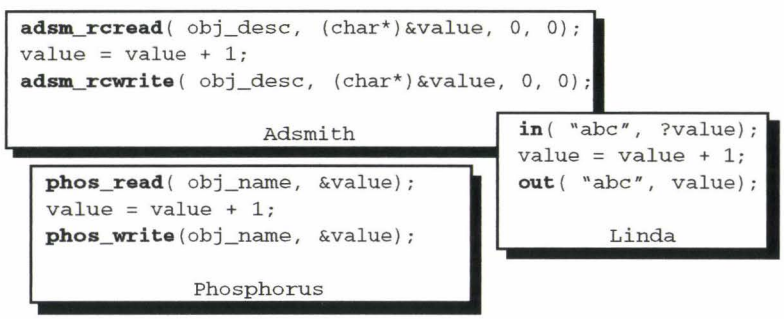



**figure 2.21** Accès a un objet partagé avec Adsmith, Linda et Phosphorus

### Accès dans une section critique

Dans une section critique (s.c.), l'accès à un objet est encadré par une entrée et une sortie de cette section critique (figure 2.22). Cette entrée et cette sortie permettent alors de synchroniser les accès à l'objet partagé. La section critique est gardée par une variable qui peut soit être l'objet lui même, la section critique garantit alors uniquement les accès à l'objet, soit être indépendante. Dans ce dernier cas, il faut associer la section critique à un ou plusieurs objets partagés, et n'accéder à ces objets que dans la section critique correspondante.

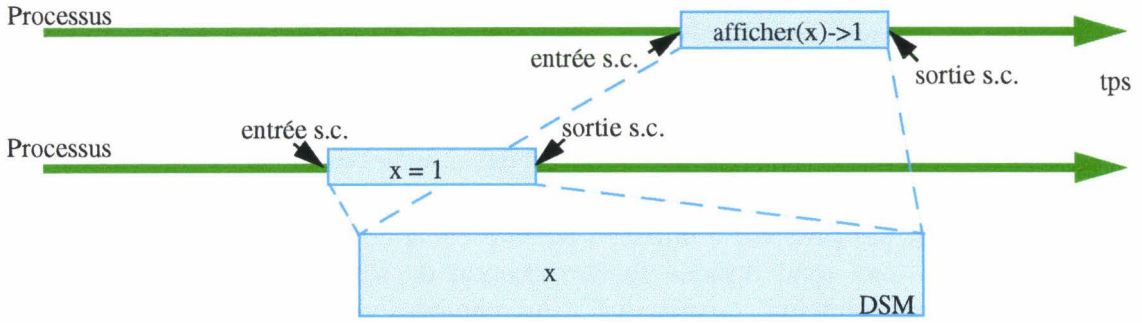



**figure 2.22** Accès dans des sections critiques

Les fonctions d'entrée et de sortie de la section critique peuvent être différentes, comme dans Crl (figure 2.23), selon que le processus entre ou sorte de la section, et selon que l'opération est une lecture ou une écriture.

```
rgn_start_write( obj_ptr );
obj_ptr->x = 3;
rgn_end_write( obj_ptr );

Crl
```

**figure 2.23** Accès à un objet partagé avec Crl

La section critique permet au gestionnaire de mémoire partagée d'intervenir sur un objet aussi bien avant que après un accès, alors que la copie ne permet d'intervenir qu'une fois. Il faut noter que la modification d'un objet est bien souvent précédée d'une consultation de cet objet, il faut donc, pour la méthode par copie, appeler les deux fonctions, l'une avant, l'autre après.

Une section critique permet de limiter le nombre de processus accédant à un objet, et par là synchroniser les accès à cet objet. Cette synchronisation est souvent d'un processus en section critique lors d'un accès en écriture, et de plusieurs processus lors d'accès en lecture.

### 2.4.5 Adresse d'un objet partagé

Dans une application distribuée, chaque processus possède son propre espace d'adressage. Un objet partagé entre ces processus possède alors une adresse dans chacun de ces espaces. Cette adresse peut être différente d'un processus à l'autre, ou être la même dans tous les processus.

Si l'adresse d'un objet partagé est différente d'un processus à l'autre, la référence sur un objet devient locale au processus et n'a pas de signification dans les autres processus. Cette référence ne peut alors pas être échangée entre les processus, ce qui interdit la construction de structures partagées complexes se référençant entre elles, comme les listes chaînée.

Si maintenant on choisit de donner à un objet la même adresse dans tous les processus, d'autres problèmes se posent. Tout d'abord il faut trouver une adresse libre dans tous les processus, et s'assurer qu'elle ne sera pas utilisée pour autre chose par un processus. En effet, que faire si un processus veut partager un objet et qu'il s'aperçoit que l'adresse est déjà utilisée ?

Une solution est de réserver une partie de l'espace d'adressage des processus pour la mémoire partagée. Chaque processus se voit affecter une sous-partie de cet espace dans laquelle il crée ses nouveaux objets à des adresses qui ne sont pas utilisées par d'autres car ce sous-espace lui est réservé. Le partage de l'objet par d'autre processus se fait à l'adresse où

l'objet a été créé. Ce mécanisme est très gourmand en quantité d'adresses, mais il est de plus en plus réaliste, notamment avec l'arrivé des grands espace d'adressage de 64 bits [Carter92] [Mossiere94] comme par exemple sur les stations Alpha.

La maîtrise de l'espace d'adressage d'un processus est difficile, et nécessite de bonnes connaissances du système. C'est pourquoi la plupart des modèles de DSM existant donnent des adresses différentes. L'identification d'un objet partagé ne peut alors se faire que par son identificateur, il n'est pas possible d'utiliser l'adresse en la communiquant ou la partageant entre les processus. Ceci rend difficile la construction de structures complexes partagées utilisant la notion de pointeurs, comme les listes chaînées ou les graphes. Pourtant, ces structures sont très employées en programmation.

### 2.4.6 Des exemples

Nous allons étudier deux modèles de DSM, Crl et Phosphorus, et un langage de programmation Orca, proposant des objets partagés.

Nous avons retenu ces exemples car il en existe une implémentation opérationnelle sur réseaux de stations, ce qui n'est pas le cas de tout les modèles. De plus, ces modèles proposent des exemples d'applications que nous avons repris afin d'évaluer et de comparer Dream.

#### Crl

Crl [Johnson95] propose de partager des données à l'aide de *régions*. Une région est une zone mémoire de taille arbitraire définie par le programmeur.

Une région doit être créée par un des processus, et projetée (*mapped*) par les autres. L'accès à la zone mémoire d'une région se fait alors dans des sections critiques délimitant le début puis la fin de l'accès, ainsi que le type d'accès (lecture ou écriture).

La figure 2.24 donne le code correspondant à la recherche du minimum vue page 28.

```
int x, *min_ptr;
rid_t min_id;

if (my_name == MASTER)
  {
  min_id = rgn_create( sizeof(int) );
  rgn_bcast_send( sizeof(min_id), &min_id );
  }
 else
  rgn_bcast_recv( sizeof(min_id), &min_id );

min_ptr = rgn_map( min_id );

calculer_x();

rgn_start_write( min_ptr );
  if( x < *min_ptr )
    *min_ptr = x;
rgn_end_write( min_ptr );
```

**figure 2.24** Code de la recherche du minimum avec CRL

Avec Crl, chaque processus d'une application possède un nom (en réalité un numéro). Le processus de nom MASTER crée la région, puis diffuse l'identificateur de la région. Les autres processus reçoivent l'identificateur, puis chacun projette la région, ce qui donne l'adresse de la zone de mémoire partagée.

L'accès à la variable partagée se fait dans la section critique *rgn_start_write()* - *rgn_end_write()*. Le modèle Crl garantit que, à un moment donné, seul le processus dans la

section critique peut accéder en écriture ou en lecture à la région. Ainsi, la valeur minimum ne peut pas être modifiée simultanément par des processus différents.

Il existe une section critique équivalente, *rgn_start_read() - rgn_end_read()*, n'autorisant que la lecture de la région correspondante. Dans ce cas, Crl permet à plusieurs processus d'entrer simultanément dans la section critique, et de consulter la région.

Crl a été porté sur une machine à mémoire commune expérimentale du MIT (Alewife [Alewife95]), et sur la Thinking Machine CM-5. Il existe aussi une version pour réseau de stations, version qui ne fonctionne que sur des stations Sun avec SunOs 4.1.

**Phosphorus**

Phosphorus [Demeure95] permet de partager des données, ou des tableaux de données entre les processus d'une application. Les données sont obligatoirement d'un type de base (entier, caractère, ...).

Une donnée partagée doit être déclarée à Phosphorus par chaque processus voulant l'utiliser. L'accès à la donnée se fait ensuite soit en demandant une copie à Phosphorus (lecture), soit en donnant la dernière valeur de la donnée à Phosphorus (écriture).

Au sein de Phosphorus, une donnée est caractérisée par son identificateur, qui est un nom arbitraire fixé par le programmeur.

A chaque donnée partagée est associée un protocole. Celui-ci indique à Phosphorus comment il doit maintenir la cohérence de la donnée entre les différent processus. Le choix du protocole est fait par le programmeur lors de la déclaration de la variable, il dépend des accès qui y seront fait (Voir *Choix de la cohérence ou du protocole*, page 47).

La figure 2.25 donne un code possible pour la recherche d'un minimum avec Phosphorus. La paire *lock-unlock* est nécessaire afin de garantir qu'un seul processus à la fois modifie le minimum.

```
int x, min;
int min_id = 1234;

phos_declare( min_id, INT, 1, protocol_type);

calculer_x();

phos_lock( min_id );
phos_read( min_id, &min);
  if( x < min )
    phos_write( min_id, &x );
phos_unlock( min_id );
```

**figure 2.25** Code de la recherche du minimum avec Phosphorus

Phosphorus a été développé au dessus de PVM, et de ce fait fonctionne sur tous les réseaux de stations supportant PVM.

Phosphorus ne fonctionne que sur des variables de type simple. Il ne permet pas de partager des structures complexes composées de différents types de variables.

L'espace partagé de Phosphorus n'est pas directement accessible par les processus. Ceux-ci manipulent des copies des données partagées, et effectuent des synchronisation (lecture, écriture) quand nécessaire.

**Orca**

Orca est un langage de programmation orienté objet développé par Henry Bal et son

équipe [Bal90]. Orca propose de programmer des applications distribuées en utilisant des objets partagées entre les processus.

Le modèle utilise des processus pour exprimer le parallélisme, et des objets partagés pour la communication et la synchronisation de ces processus. Les données contenues dans un objet ne peuvent être accédées que par les méthodes, appelées opérations, définies pour cet objet.

Les processus sont créés dynamiquement à l'aide de l'instruction *fork nom_de_la_procédure()* : un processus peut créer de nouveaux processus qui peuvent à leur tour créer d'autre processus (figure 2.26).

```
PROCESS worker(minimum: SHARED IntObject)
    x : integer;
BEGIN
    x = rand();
    IF x < minimum$value() THEN
      min$assign(x);
END;
```

```
PROCESS OrcaMain();
   minimum: IntObject;
BEGIN
   FORK worker(minimum);
   FORK worker(minimum);
   WriteLine("minimum = ", minimum$value());
END;
```

**figure 2.26** Création de nouveau processus avec Orca

Les objets sont créés en déclarant des variables ayant le type de la classe de l'objet. Les objets ne peuvent être déclarés que dans une procédure, il ne peut pas y avoir de déclaration globale d'un objet.

Quand un processus crée un nouveau processus, il peut passer n'importe lequel de ses objets en tant que paramètre partagé. Le nouveau processus peut lui aussi passer l'objet à des nouveaux processus, ce qui permet de distribuer et de partager l'objet entre les différents descendants du processus l'ayant créé.

Les modifications apportées à l'objet par un des processus sont visibles des autres processus. Ainsi un objet partagé sert de canal de communication entre les processus.

Chaque appel d'une méthode est exécuté de manière indivisible : le processus ne voit pas les états intermédiaires de l'objet.

La définition d'une classe d'objet consiste en une *partie spécification* définissant les méthodes de l'objet, et en une *partie implémentation* contenant les données de l'objet et le code des opérations (figure 2.27).

Une opération peut être non bloquante, elle contient alors simplement une séquence d'instruction, ou être bloquante, elle contient dans ce cas une ou plusieurs *commande gardée*. Une commande gardée consiste en une expression booléenne suivie d'une séquence d'instructions. Une opération contenant des gardes bloque jusqu'à ce que une de ces gardes soit évaluée à *"vrai"*. Ensuite, la séquence d'instructions correspondante à cette garde est exécutée. Les commandes gardées peuvent être utilisées pour, par exemple, transférer des données entre les processus, ou synchroniser ces processus.

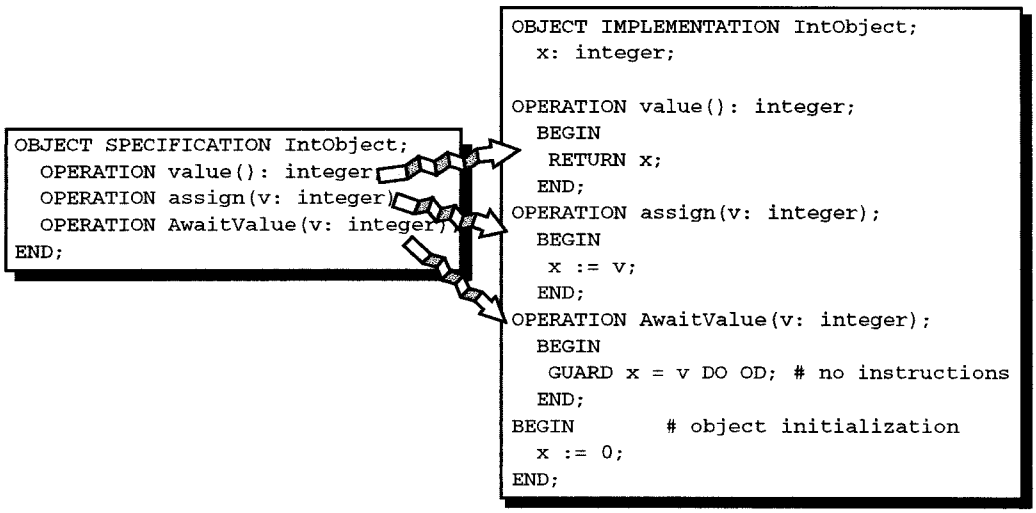



**figure 2.27** Spécification et implémentation avec Orca d'une classe d'entiers partagés

Le langage Orca est constitué d'un compilateur et d'un run-time. Le compilateur est chargé de vérifier la syntaxe des programmes sources et de générer un programme intermédiaire en langage C. Le run-time implémente les types de base (entiers, caractères, mais aussi tableaux, graphe, ensemble) et prend en charge la gestion des processus et du partage.

## 2.5 Synchronisation inter-processus

La mémoire partagée est accessible par différents processus, et il se pose donc le problème des accès concurrents, et de la synchronisation globale inter-processus.

La synchronisation des accès à un objet commun est un problème connu des systèmes centralisés. Les conflits y sont résolus à l'aide d'outils de synchronisation comme les sémaphores ou les variables conditionnelles. Ces outils de synchronisation sont bien maîtrisés et une solution pourrait être d'essayer de les implémenter à l'aide de la mémoire partagée. Malheureusement, cette implémentation uniquement à l'aide de mémoire partagée est inefficace à cause du grand nombre d'échanges de message nécessaires pour maintenir la cohérence [Dubois88] [Carter90].

Alors, les modèles de mémoire partagée fournissent leurs propres outils de synchronisation dont l'implémentation est cachée au programmeur. Parmi ces outils, on trouve la *barrière*, la *section critique*, et le *broadcast*.

La figure 2.28 montre le squelette général de la synchronisation d'un processus d'une application, ainsi que les fonctions correspondantes pour différents modèles. Dans ce squelette, on trouve une phase d'initialisation, une phase de calcul et une phase d'exploitation/ affichage des résultats. Chaque phase est séparée par une synchronisation par barrière. La phase de calcul comporte des accès aux variables partagées dans des sections critiques, dont un seul accès est représenté.

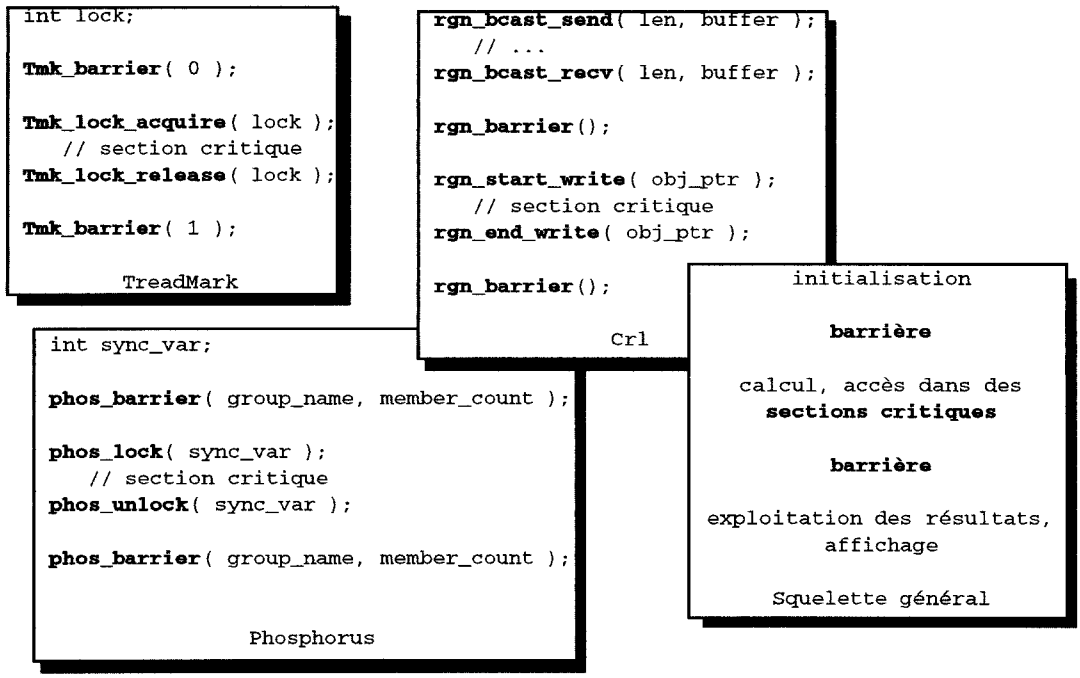



**figure 2.28** Outils de synchronisation proposés par TreadMark, Crl et Phosphorus

Une barrière permet à un ensemble de processus de se synchroniser (synchronisation tous vers tous). La synchronisation est effective après que chaque processus ait appelé la barrière. L'utilisation des barrières peut nécessiter un nom identifiant la barrière (TreadMark, Phosphorus), et parfois le nombre de processus participants à la barrière (Phosphorus).

Les sections critiques permettent de protéger l'accès à une variable en restreignant le nombre de processus y accédant. Une section critique est gardée soit par une variable indépendante, soit par la variable partagée elle-même. En général, les mémoire partagées proposent les deux possibilités.

Dans les modèles comme Crl, la section critique fait partie intégrante de l'accès à la variable (*start_write-end_write*). Il n'est d'ailleurs pas possible d'accéder à une variable partagée en dehors de sa section critique.

La synchronisation par *broadcast* se rencontre dans Crl. Elle permet au processus ayant créé les régions contenant les objets partagés de diffuser les identificateurs de ces régions (synchronisation un vers tous). Dans ce modèle, cette synchronisation est rendue nécessaire car au départ, seul le créateur des régions possède les identificateurs de celles-ci (voir figure 2.6).

## 2.6 Conclusion

Nous avons montré dans ce chapitre les difficultés et la complexité rencontrées dans la réalisation d'une DSM. Cette réalisation nécessite des échanges de messages afin de maintenir la cohérence des données.

Le nombre de messages échangés est gplus important avec une DSM qu'avec une solution directement à base d'échanges de messages car la DSM doit pouvoir répondre à différents cas de partage. Ceci entraîne bien sûr des performances sensiblement inférieures avec une DSM.

Cependant, la conception d'une application parallèle est généralement facilitée par la présence d'une DSM. En particulier, la DSM permet un partage et une mise en commun aisée

de l'information.

C'est pourquoi différents modèles de DSM ont été développés. L'évolution actuelle tend à favoriser une granularité variable, ceci afin d'éviter le problème du faux partage. En ce qui concerne la cohérence, différents modèles sont proposés, mais il est généralement admis qu'il n'existe pas de "cohérence universelle". Un modèle de cohérence est adapté à un problème donné, mais ne convient pas à un autre. Certaines DSM proposent plusieurs types de cohérences, mais le choix d'une d'entre elle doit être fait au début de l'application, et ne peut plus être modifié.

Pour ce qui est de l'interface d'utilisation de la DSM, celle-ci doit être la plus transparente possible, afin de préserver la facilité de programmation. L'idéal serait que la manipulation d'un objet partagé soit identique à celle d'un objet non partagé. Malheureusement, la recherche de bonnes performances oblige à introduire des opérations de manipulation supplémentaires.

La synchronisation inter-processus, par exemple pour la synchronisation des accès à un objet partagé, est plus efficace quand elle est réalisée avec des échanges de messages. Mais la plupart des modèles actuels de DSM ne permettent pas l'accès à l'interface d'échange de messages. Ces modèles proposent alors différents outils de synchronisation prenant eux-même en charge les échanges de messages.

Dans ce contexte, nous proposons Dream, un modèle de programmation hybride composé de processus communicants (messages) et d'une mémoire partagée répartie (DSM). Cette dernière est destinée aux applications telles que celles abordées dans le chapitre 1: répartition de charge, récolte de trace de déverminage, ramasse-miettes. Ces applications ne sont pas au niveau du système, mais elles ne sont pas non plus du ressort du programmeur. Elles se situent à un niveau intermédiaire. Leur point commun est de nécessiter une prise de connaissance de l'état global, qui est facilitée par l'utilisation de la DSM. De plus, ces applications peuvent fonctionner avec des données "un peu anciennes". Les contraintes portant sur la cohérence peuvent donc être relâchées, ce qui nous permet de proposer une cohérence faible que nous définirons plus précisément. Pour les applications ayant besoin d'une cohérence plus forte, le modèle Dream propose un contrôle original de la cohérence, contrôle pouvant s'exercer, et donc varier, à tout moment durant l'application.

L'interface de programmation de Dream sera composée d'une part de l'interface de programmation du modèle de processus communicant retenu, et d'autre part de l'interface de programmation de la DSM. Cette dernière tendra vers une utilisation la plus transparente possible. La synchronisation se fera par les processus communicants, Dream mettant à disposition les moyens de communiquer avec les processus partageant une même donnée.

# 3 Le modèle DREAM

Dans ce chapitre, nous présentons le modèle de mémoire partagée Dream. L'objectif de Dream est de donner aux applications l'illusion du partage d'information par la notion de mémoire partagée. L'implémentation d'une mémoire partagée est réalisée par une couche logicielle qui simule le partage physique. Le principe est que chaque processus réserve une portion de son espace mémoire pour la mémoire partagée, et Dream se charge ensuite d'assurer une certaine cohérence de l'ensemble.

Le modèle de programmation Dream consiste à étendre le modèle de processus communicants par des possibilités d'accès à une mémoire partagée. Nous nous sommes intéressés dans le cadre de cette thèse à des modèles de processus communicants à gros grain. L'environnement que nous avons retenu est PVM qui fournit des fonctionnalités pour la création distante de processus et des primitives de communication. Dream fournit quant à lui le concept de partage d'information. Ceci permet d'offrir un modèle hybride (figure 3.1) composé de processus communicants et d'une mémoire partagée (DSM). Ainsi, les avantages des deux modèles sont disponibles: les possibilités de communication et de synchronisation par messages, et la facilité de partage d'informations par une mémoire partagée.

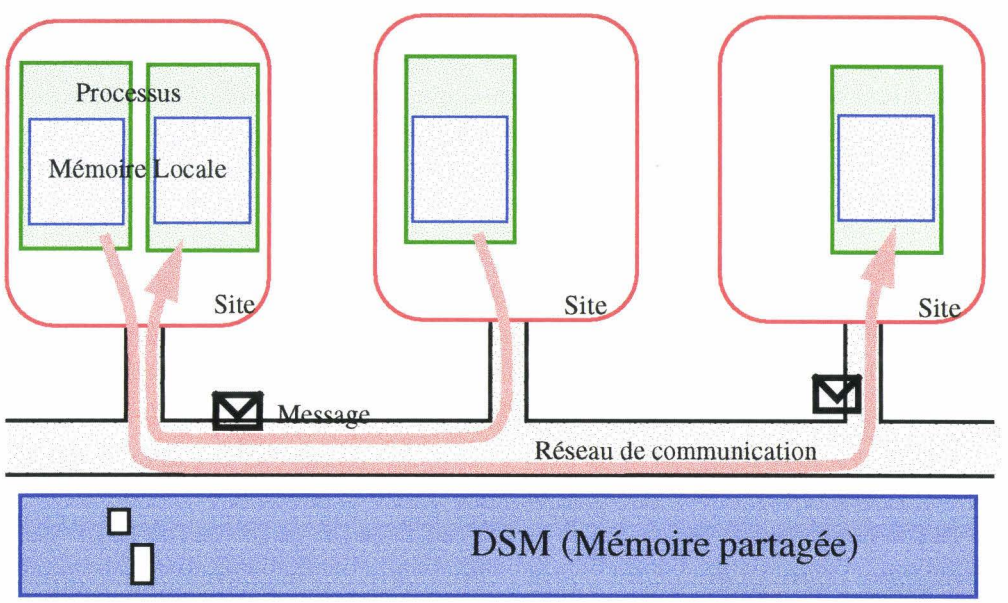



**figure 3.1** Dream: processus communicant et mémoire partagée

L'espace de mémoire partagée est structuré en entités appelées régions. Les régions sont créées dynamiquement par les processus de l'application. La taille d'une région est fixée lors de sa création.

Pour pouvoir accéder en lecture à une région, le processus doit projeter cette région dans son espace mémoire: c'est l'opération d'attachement. Une région de l'espace partagé peut être

attachée et donc projetée dans plusieurs processus. Mais seul le propriétaire, qui est en général le processus créateur, a le droit de modifier le contenu d'une région. Cette restriction élimine les problèmes de synchronisation et d'écritures multiples.

Se pose alors le problème classique de la cohérence. Nous proposons un mécanisme de cohérence faible que nous définirons précisément dans ce chapitre[1]. Schématiquement, notre modèle de cohérence induit un certain délai entre la modification d'une région par le processus propriétaire et sa perception par les autres processus.

Ce mode de cohérence, valable pour certaines classes d'application, n'est bien sûr pas applicable pour toute classe d'applications. Dream fournit aussi des mécanismes permettant au programmeur de renforcer le niveau de cohérence.

Dans la première partie de ce chapitre nous développons la notion de région en tant que structure de partage. Pour ce faire, nous introduisons et classifions la notion d'objet partagé, que nous relions à la notion de région. Ensuite nous étudions le modèle de cohérence proposé par Dream, ainsi que les méthodes proposées pour contrôler cette cohérence. Une troisième partie est consacrée à l'interface d'accès à la mémoire partagée. La quatrième partie aborde le problème de la synchronisation inter-processus, notamment lors des accès à un objet commun. C'est dans cette partie que nous détaillons la notion de propriétaire d'une région.

## 3.1 Objets, régions et caches

Une application développée dans l'environnement Dream est composée d'un ensemble de processus communiquant par messages et se partageant des objets dans un espace commun. Par objet, nous entendons des données simples (entiers, booléens) ou des données plus complexes comme par exemple des structures (enregistrements), des tableaux ou encore des listes chaînées.

L'espace partagé de Dream contient les objets partagés de l'application. Ces objets partagés peuvent être classés en trois catégories selon la fréquence des consultations et des modifications. Ainsi on trouve les objets constants, les objets faiblement variables et les objets fortement variables. Toutefois, la frontière entre ces trois catégories n'est pas stricte. Un objet peut se trouver à cheval entre deux catégories, et osciller de l'une à l'autre au cours de l'exécution de l'applications.

Cette classification est liée à la cohérence de l'objet. Elle assure au programmeur une "bonne" cohérence de l'objet, à condition que ce dernier réponde aux critères de la catégorie. Le choix d'une catégorie n'est pas irréversible: le programmeur peut toujours contrôler ou modifier la cohérence d'un objet si celui-ci change de comportement en cours d'exécution.

Même si généralement le programmeur manipule des *objets partagés*, ces derniers sont en réalité contenus dans des *régions* qui sont l'unité de partage de Dream. Ces régions sont elles même regroupées dans des *caches*, chaque cache contenant l'ensemble des régions utilisées localement par un processus.

### 3.1.1 Objets partagés

Dream comprend un espace de mémoire virtuellement partagé appelé la Dsm (Distributed Shared Memory) (figure 3.2). Cet espace est virtuel, car il n'a pas de réelle existence physique. Il donne aux processus de l'application l'illusion de l'existence d'une mémoire partagée. Le programmeur voit la Dsm comme un espace de stockage des objets qui ont besoin d'être partagés par plusieurs processus. Un objet qui n'est utilisé que par un processus, reste local à ce processus. Cet objet est naturellement stocké dans l'espace mémoire

1. Rappel : la cohérence faible de Dream n'est pas la *weak consistency*.

local (ou privé) du processus.

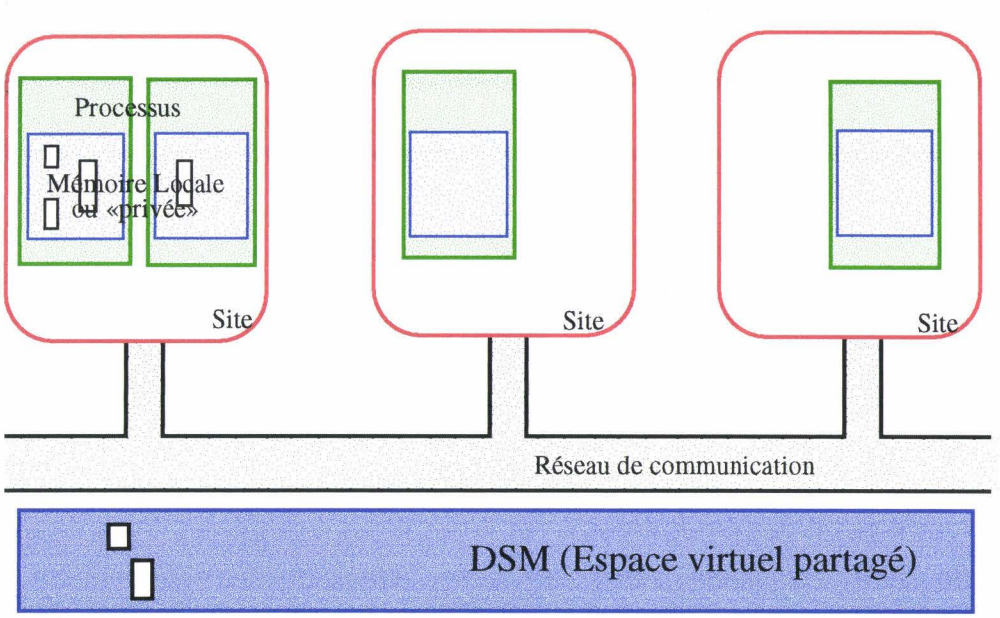



**figure 3.2** La Dsm

Un des objectifs de Dream a été de rendre l'utilisation d'un objet partagé la plus proche possible de l'utilisation d'un objet non partagé (local). Exception faite de l'opération de création ou d'attachement qui correspond à une phase de localisation de l'objet dans l'espace partagé, l'accès en lecture ou en écriture sur un objet se fait de la même manière que pour un objet local. C'est à dire qu'au niveau langage de programmation, il n'y a ni de nouveaux opérateurs, ni de nouvelles fonctions à écrire ou à utiliser pour accéder aux objets partagés.

Nous avons classé les objets partagés en fonction de la fréquence de modification ou de consultation de ces objets. Dans une première classe, appelée classe des *objets partagés constant,* on trouve des objets dont le contenu est évalué une seule fois par un des processus, les autres processus de l'application ayant simplement besoin d'accéder en lecture au contenu de cet objet.

On trouve ensuite la classe des objets variables, c'est à dire des objets qui peuvent être modifiés à plusieurs reprises. Dans cette dernière catégorie il est possible de distinguer les objets plus souvent lus que modifiés, que nous appelons *faiblement variables*, et les objets fréquemment lus et écrits dits *fortement variables*.

Etudions maintenant l'utilisation de Dream pour représenter ces objets.

**Objet partagé constant**

Un objet partagé constant est donc un objet dont le contenu est évalué une seule fois. Ce type d'objet est créé et initialisé par un des processus de l'application. Par la suite, l'objet peut être consulté par les autres processus, mais son contenu ne sera plus jamais modifié (figure 3.3).

Avec Dream, ce type d'objet est partagé à l'aide de l'espace virtuel. Lors de la phase d'initialisation, l'un des processus crée l'objet dans la Dsm et en devient propriétaire. Il peut par conséquent initialiser l'objet. Les autres processus ont potentiellement le droit de lire cet objet. Ils ne peuvent effectivement le lire qu'après l'avoir projeté dans leur espace d'adressage à l'aide de l'opération d'attachement. Une fois attaché par un processus, l'objet reste accessible jusqu'à ce qu'il soit explicitement détaché par ce processus.

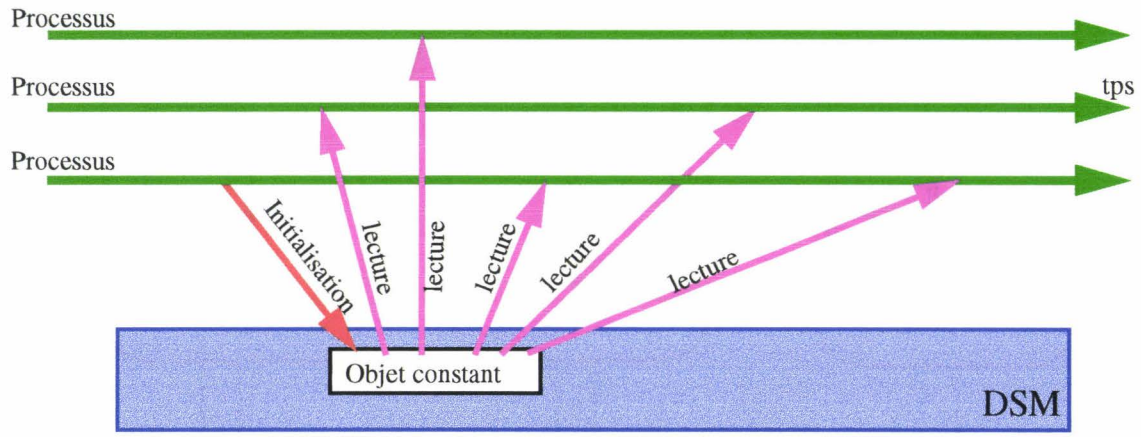



**figure 3.3** Accès à un objet constant

### Objet partagé faiblement variable

Un objet faiblement variable est un objet qui est, comme pour un objet constant, initialisé par un processus et qui est accédé en lecture par d'autres processus de l'application. Mais à la différence des objets constants, le contenu de l'objet pourra être modifié au cours de l'exécution (figure 3.4). Ces modifications seront peu fréquentes et seront généralement effectuées par le processus propriétaire de l'objet, c'est à dire le processus qui a été à l'initiative de la création.

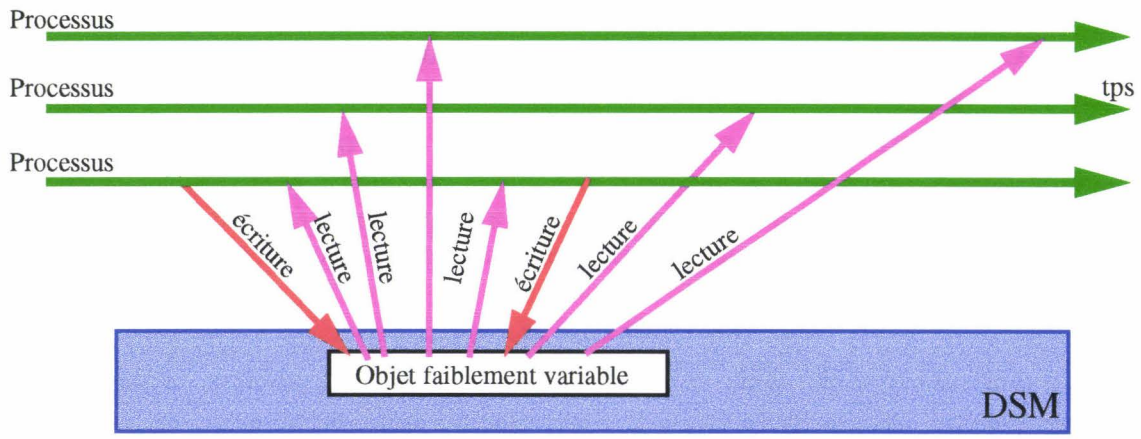



**figure 3.4** Accès à un objet faiblement variable

Avec Dream, la réalisation d'un objet partagé faiblement variable ressemble à celle d'un objet constant : l'objet est créé dans l'espace virtuel partagé, ce qui le rend potentiellement accessible de tous les processus de l'application.

L'un des processus crée l'objet et en devient le propriétaire. Il peut alors, à ce titre, le modifier. Un autre processus de l'application désirant consulter le contenu de cet objet, doit l'attacher avant la première utilisation; Une fois cette opération d'attachement effectuée, le processus peut accéder autant de fois que nécessaire à l'objet.

L'objet est ainsi virtuellement partagé par l'ensemble des processus de l'application: quand le processus propriétaire modifie l'objet, les autres processus «voient» la modification au bout d'un temps fini.

Dream se charge du maintient de la cohérence de l'objet. Le programmeur n'a pas à s'en

soucier. Ainsi, il accède à un objet partagé comme il accède à un objet non partagé. Il n'est même pas nécessaire de protéger les accès par des sections critiques, ni de les faire précéder ou suivre par une synchronisation de l'objet.

Cette facilité d'utilisation a toutefois une contrepartie: les modifications intervenues sur un objet ne sont pas toujours immédiatement vues par l'ensemble des processus. Ainsi, la cohérence proposée par Dream est "faible", par opposition à la cohérence dite "forte" (La cohérence "faible" proposée par Dream ne correspond pas à la *weak consistencies* tel que définies par Dubois, et décrite page 41).

Toutes les applications ne peuvent pas fonctionner avec ce type de cohérence. Néanmoins, elle convient parfaitement à certains types d'applications dont nous verrons quelques exemples au paragraphe 5.2 *Applications "systèmes"*, page 120.

Pour les autres applications, il est toujours possible de renforcer la cohérence. Celle-ci peut être contrôlée à tout moment par le programmeur, et ce individuellement pour chaque objet (voir 3.2 *Cohérence*, page 65).

**Objet partagé fortement variable**

Les objets fortement variables sont des objets qui sont fréquemment accédés en lecture ou en écriture (figure 3.5). De plus, pour cette famille d'objet, nous considérons que tout processus a potentiellement le besoin de modifier l'objet. Pour ce type d'objet, le ratio lecture/ écriture est proche de 1, c'est à dire qu'un processus donné effectue sur l'objet autant de lectures que d'écritures.

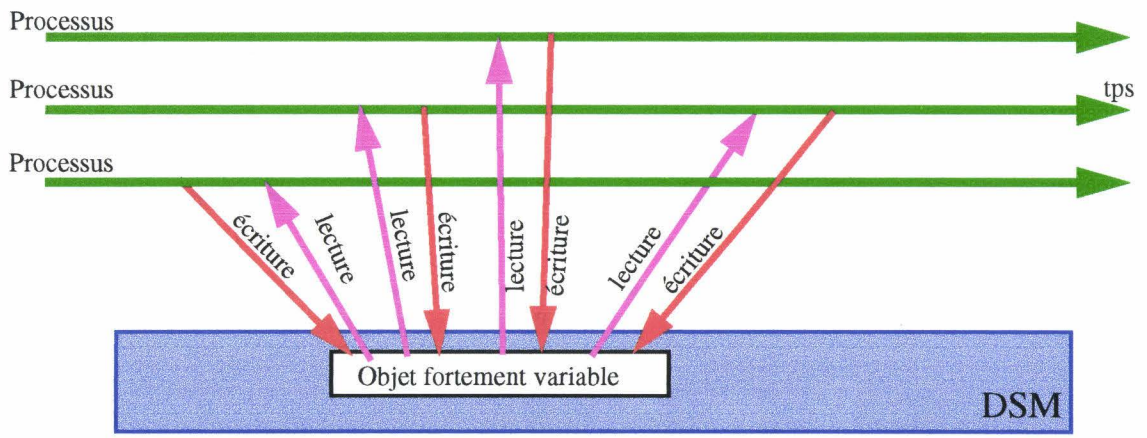



**figure 3.5** Accès à un objet fortement variable

Avec Dream, la conception d'application manipulant des objets partagés fortement variables se fait sur un schéma similaire à ceux utilisés pour les autres classes d'objets partagés. Un des processus de l'application crée l'objet et en devient le propriétaire. Les autres processus voulant accéder à cet objet doivent le projeter dans leur espace d'adressage à l'aide de l'opération d'attachement.

L'objet ne leur est alors accessible qu'en lecture. En effet, dans Dream, un objet ne peut être modifié que par son propriétaire, et il ne peut y avoir qu'un seul propriétaire à un moment donné. Si le processus veut modifier l'objet, il doit au préalable en devenir le propriétaire. Dream fournit des primitives pour changer le propriétaire d'un objet. Le mécanisme de changement de propriétaire est étudié plus en détail au paragraphe 3.4.1 *Propriétaire d'une région*, page 84.

Nous savons déjà que les processus clients (ceux qui n'ont qu'un accès en lecture) ne

voient pas immédiatement les modification effectuées par le propriétaire, ce qui se traduit par une cohérence «faible» des données. Cette cohérence est d'autant plus faible que la fréquence de modification est importante. Le programmeur peut agir sur le degré de cohérence afin de l'adapter à ses besoins et d'obtenir une cohérence plus forte (voir 3.2 *Cohérence*, page 65).

Bien sûr, une cohérence plus forte entraîne une participation plus importante de Dream, et par conséquent une augmentation des temps d'accès aux données partagées.

### 3.1.2 Régions

Les objets partagés ne sont pas directement créés dans la Dsm. En réalité, l'espace virtuel partagé est structuré en régions destinées à contenir les objets. Une région est une portion de mémoire partagée créée dynamiquement par un processus d'une application. Elle est assimilable à une page d'une mémoire virtuelle mais à la différence decelle-ci, toutes les régions n'ont pas la même taille, et sont créées à la demande explicite des processus de l'application.

Toutes les régions sont contenues dans la Dsm qui sert d'hôte (figure 3.6). Une région sert d'unité de partage, et est destinée à recevoir un ou plusieurs objets devant être partagés. Le partage d'un objet se fait uniquement à l'aide d'une région, il n'est pas possible de partager des objets en dehors d'une région.

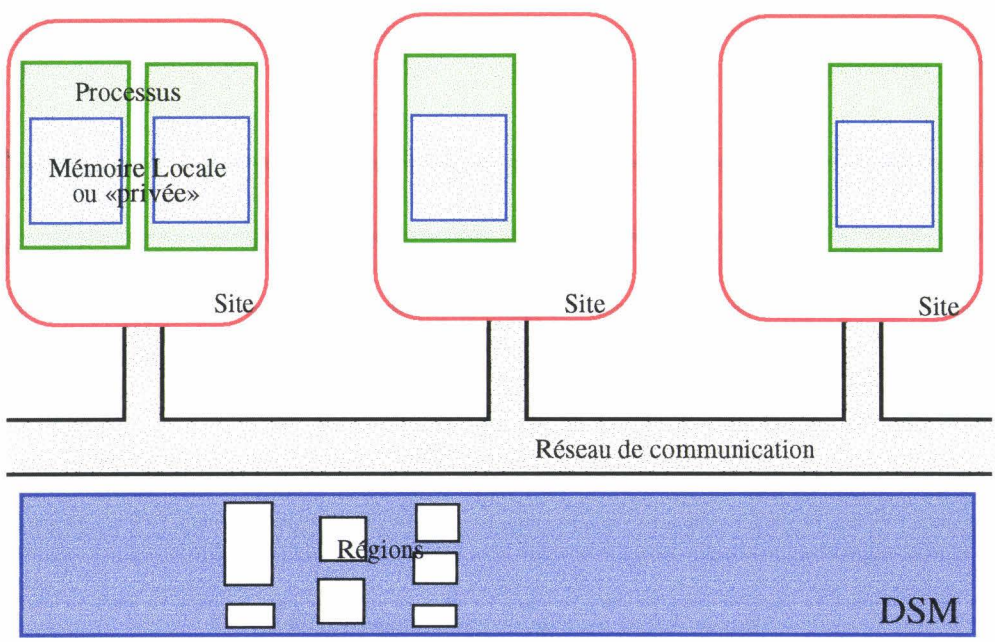



**figure 3.6** Les régions

#### Régions et objets partagés

Une région sert d'unité de partage. Tous les objets se trouvant dans une même région ont en commun les propriétés de la région. Notamment, l'attachement de la région induit l'attachement des objets qu'elle contient. De même, tous ces objets présente la même cohérence.

La création de la région, quant à elle, n'implique pas toujours la création d'objets partagés. En effet, c'est au programmeur de créer son ou ses objets dans la région, soit statiquement en projetant une structure composée d'un ou plusieurs objets, soit dynamiquement en créant des objets dans la région au fur et à mesure des besoins de l'application. Il choisira une création statique quand le nombre d'objets à partager dans la

région est bien connu lors de la conception de l'application. Dans le cas contraire, il choisira une création dynamique.

Il existe trois possibilités pour répartir les objets dans les régions:

1) placer plusieurs objets dans une région

2) placer un objet par région

3) diviser un objet pour le répartir dans plusieurs régions

Etudions ces solutions en fonction de la classification des objets faite précédemment.

Des objets constants sont initialisés par un processus de l'application. Ensuite, ils sont consultés par les processus de cette application. Il est conseillé de placer le maximum d'objets constants dans une région. Ainsi l'attachement de cette région permettra l'attachement simultané de ces objets constants.

Les objets faiblement variables sont modifiés par un processus et lus par les autres. Afin d'éviter le problème du faux partage (Voir *Problème du faux partage*, page 34), il est préférable de placer les objets modifiés par des processus différents dans des régions différentes. Ainsi chaque objet est modifiable indépendamment des autres objets.

Par contre, il est préférable de placer les objets «liés» dans un même région. Deux objets sont liés quand l'accès à l'un entraîne souvent l'accès à l'autre, et que le risque de faux partage est réduit, voir inexistant. Placer les objets liés dans une même région permet de garantir le même comportement, et notamment la même cohérence pour tous ces objets.

Quant aux objets fortement variables, ils sont souvent lus et modifiés. La remarque faite à propos des objets faiblement variables s'applique aussi: il est conseillé de placer les objets liés dans une même région, et les objets modifiés par des processus différents dans des régions différentes.

Certains gros objets se décomposent en parties non liées, ou modifiées par des processus différents. Dans ce cas, il est conseillé de diviser l'objet selon les parties indépendantes, et de les répartir dans des régions différentes. Prenons l'exemple d'une matrice résultat. Elle peut être divisée en colonnes, chacune calculée, et donc modifiée, par un processus différent. Il est alors préférable de répartir les colonnes dans des régions différentes.

**Propriétaire unique**

Il faut rappeler que seul le processus propriétaire d'une région, qui est en général son créateur, a le droit de modifier le contenu de la région, et par conséquent le ou les objets. Les autres processus n'ont que le droit de consulter la région et ce qu'elle contient.

Le fait que seul le propriétaire d'une région puisse la modifier peut paraître un peu restrictif. En fait, il n'en est rien car une région peut être modifiée par différents processus, mais à des moments différents. Il suffit pour cela de transférer le titre de propriétaire d'un processus à l'autre. Ce mécanisme de changement de propriétaire est détaillé plus loin (3.4.1 page 84).

Le mécanisme de propriétaire unique interdit donc plusieurs processus d'écrire simultanément dans une même région. Ainsi les problèmes de synchronisation des écritures concurrentes sont implicitement éliminés.

Le mécanisme de propriétaire unique n'interdit pas plusieurs processus d'écrire en même temps dans différentes régions. Comme la granularité des régions est fine, il est possible de diviser un objet et de le partager à l'aide de plusieurs régions. Ainsi plusieurs processus peuvent modifier simultanément un même objet. Cette possibilité est parfois appelée "multiple

writer protocol" ou "write shared protocol" [Carter93], [Demeure95]. Nous trouvons que ce terme n'est pas tout à fait adapté car dans ce protocole deux processus ne peuvent jamais modifier simultanément la même case mémoire. Tout comme Dream ne permet pas de modifier simultanément la même région.

**Régions miroirs**

Jusqu'à présent nous avons vu que les objets partagés sont placés dans des régions, et que ces régions sont contenues dans la Dsm qui est un espace de mémoire virtuellement partagé par les processus. Nous savons aussi que du point de vue matériel (hardware) un processus ne peut qu'accéder à la mémoire physique (locale) qui lui appartient. Dans ces conditions, quand un processus accède à la portion mémoire correspondant à une région, il faut que cette mémoire soit locale au processus. C'est pourquoi chaque processus ayant attaché une région en possède une copie locale appelée région miroir (figure 3.7).

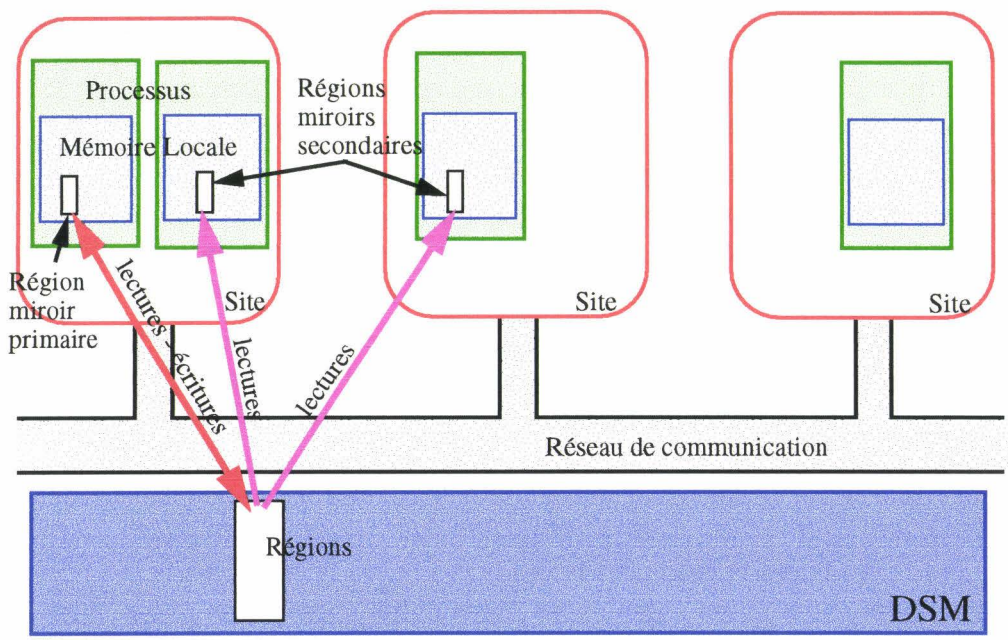



**figure 3.7** Régions miroirs

Cette copie est créée lors de l'opération d'attachement ou lors de la création. Une région miroir peut être considérée comme étant la vue locale du processus sur la région.

Une région de la Dsm est donc représentée par un ensemble de régions miroirs reflétant plus ou moins fidèlement son contenu. Parmi ces régions miroirs, on distingue la région miroir primaire, détenue par le propriétaire, des régions miroirs secondaires détenues par les autres processus.

Nous verrons que la région miroir primaire dispose de droits privilégiés sur la région. Ainsi nous avons déjà vu que le processus propriétaire est le seul à pouvoir modifier la région. La région miroir primaire est donc la seule région miroir à pouvoir être modifiée.

**Régions et espace virtuel partagé (Dsm)**

La Dsm, représentant l'espace virtuel partagé, sert d'hôte à l'ensemble des régions. Elle propose un certain nombre d'outils pour consulter cet ensemble. Il est ainsi possible de parcourir la liste des régions, ou de vérifier qu'une région est présente dans la Dsm.

Le fait qu'une région soit dans la Dsm signifie que cette région existe, et qu'elle est

potentiellement accessible par les processus. Cela ne signifie pas qu'un processus donné puisse accéder à la région. Il ne peut y accéder effectivement qu'après l'avoir projetée dans son espace d'adressage à l'aide de l'opération d'attachement.

Les régions contenues dans la Dsm doivent toujours être utilisées par au moins un processus. La Dsm ne contient pas de régions non utilisées: celles-ci sont systématiquement détruites lorsqu'elles ne sont plus attachées à aucun processus.

### 3.1.3 Caches

Nous venons de voir que Dream propose un espace virtuellement partagé entre différents processus. Cet espace est structuré en régions elles aussi virtuellement partagées. L'existence physique d'une région est quant à elle matérialisée dans un processus par une copie locale appelée région miroir. Toutes les copies locales d'un processus sont regroupées au sein d'une entité que nous appelons un Cache (figure 3.8).

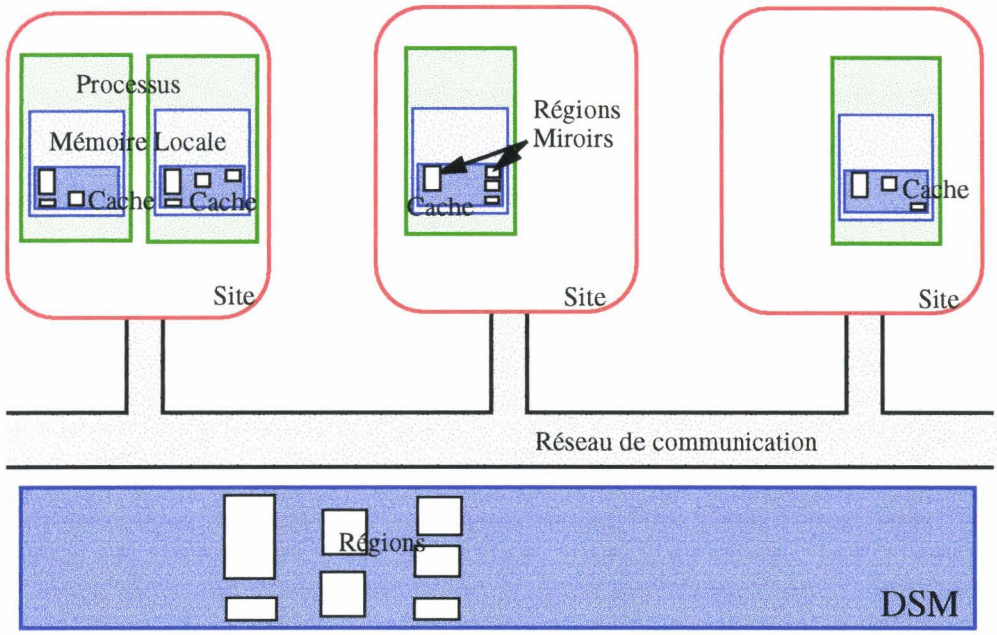



**figure 3.8** Caches

Chaque processus possède son propre cache qui lui est local, et qui diffère de celui des autres processus. Quand un processus attache une région, il crée une copie locale, ou région miroir, qui est ajoutée au cache.

Une région miroir est donc contenue dans le cache et est directement accessible par le processus tant qu'il ne la détache pas, c'est à dire tant qu'il ne l'enlève pas du cache. Il est donc inutile qu'un cache contienne plusieurs copies d'une région. Pour cela toute opération supplémentaire d'attachement d'une région déjà attachée par un processus donne la région miroir du premier attachement.

Le cache peut être considéré par un processus comme une représentation partielle et locale de la Dsm. Le cache est nécessaire afin de matérialiser la Dsm et ses régions au niveau des processus. La Dsm est virtuelle et n'a d'existence physique qu'à travers les caches qui contiennent des copies des régions.

## 3.2 Cohérence

Dans les modèles actuels de DSM, le programmeur a rarement le choix de la cohérence.

Celle-ci lui est imposée par le modèle. Quand, cependant, il a ce choix, il doit alors décider quelle cohérence utiliser. Ce choix se fait en fonction de l'utilisation des objets partagés. Il est fait par le programmeur lors du développement de l'application : le programmeur associe une des cohérences proposées à un objet partagé. La cohérence de l'objet ne peut pas être modifiée durant l'exécution de l'application.

Le modèle de cohérence proposé par Dream est différent. C'est une cohérence que nous qualifions de "faible", par opposition aux cohérences dites "fortes" dans lesquelles une modification de la mémoire partagée est vue par l'ensemble des processus au terme d'une relation complexe entre les opérations d'écritures et de lectures effectuées dans cette mémoire.

Dans Dream, la cohérence "faible" signifie tout simplement que lorsqu'un processus lit dans une région, il n'est sûr ni de trouver la toute dernière modification effectuée dans la région, ni qu'il y ait de relation entre les opérations d'écritures et de lectures.

Il faut noter que notre définition de la cohérence faible est différente de celle vue en 2.3 page 35 dans laquelle "faible" est synonyme d'un relâchement des contraintes sur l'ordre des opérations effectuées entre deux opérations de synchronisation.

### 3.2.1 Caractéristiques de la cohérence faible

Le modèle de cohérence faible de Dream est caractérisé par les propriétés suivantes:

- *La modification d'une région est vue par les autres processus au bout d'un temps fini.*
- *Un accès en lecture à une région ne retourne pas toujours la plus récente modification.*
- *Un accès mémoire à une région attachée est toujours immédiat.*

La première propriété signifie que les modifications effectuées dans une région ne sont pas immédiatement propagées aux processus attachant la région. Elles le seront plus tard, au bout d'un temps indéterminé mais fini. Ceci permet à Dream de regrouper plusieurs modifications et de les propager toutes en une seule fois.

La deuxième propriété signifie que la valeur lue dans une région peut ne pas être la dernière valeur écrite par le processus propriétaire de la région. Cette propriété est complémentaire de la première: si l'information modifiée n'est pas immédiatement propagée, l'information lue peut ne pas être la plus récente.

Les deux premières propriétés rompent le lien entre les écritures et les lectures: dans Dream, contrairement aux autres modèles de cohérence, il n'est pas nécessaire de déployer un mécanisme de maintien de la cohérence entre la modification et la consultation d'un objet.

L'absence de maintien systématique de la cohérence, associée à la structuration des régions en régions miroirs permettent, pour chaque processus, d'accéder immédiatement à un objet. C'est ce que traduit la troisième propriété: lors de l'accès à un objet, il n'y a pas d'attente due à la synchronisation de la mémoire. De ce fait, les temps d'accès aux objets partagés sont similaires aux temps d'accès aux objets non partagés.

Cette troisième propriété donne aussi la possibilité d'écrire et de lire simultanément dans une région partagée. En effet, les accès à une région étant immédiats, rien n'interdit à deux processus d'accéder au même moment à une région qu'ils ont l'un et l'autre attachée, l'un en lecture, l'autre en écriture. Ceci ne pose pas de conflit d'accès car, tandis que le processus écrivain modifie la région, le processus lecteur en consulte une autre version. Les conflits d'accès en écriture sont impossibles car il n'existe qu'un et un seul processus propriétaire d'une région à un moment donné. Le problème qui peut se poser est une mise à jour de la région par Dream pendant la lecture par un processus. Les données sur lesquelles travaille le processus lecteur sont alors modifiées, ce qui peut entraîner des dysfonctionnements. Ce

problème est évité en interdisant les modifications de la copie locale de la région à l'aide du mécanisme de gel d'une région (Voir *Gel d'une région*, page 70).

Par défaut, Dream maintient simplement une cohérence faible sur les régions. Celle-ci est obtenue par une mise à jour automatique des régions.

### 3.2.2 Mise à jour automatique

La mise à jour automatique des régions est le comportement par défaut de Dream. Dans ce cas, la cohérence garantit que les modifications d'une région sont vues par les autres processus au bout d'un temps fini. Elle garantit aussi qu'un processus ayant attaché une région peut toujours y accéder directement sans indiquer les opérations qu'il effectue, et que ces accès sont immédiats, sans latence. En contrepartie, le degré de cohérence est alors une notion approximative.

En effet, la mise à jour étant effectuée automatiquement par le système à une fréquence dépendant de l'intervalle de mise à jour, il s'ensuit que la cohérence de l'objet est fonction de cet intervalle, mais aussi de la fréquence de modification de l'objet. Quoiqu'il en soit, il est possible d'avoir une notion plus précise du degré de cohérence à l'aide du degré instantané de cohérence.

#### Intervalle de mise à jour

La cohérence faible de Dream est assurée automatiquement par le système, sans intervention de la part du programmeur. Les régions sont périodiquement synchronisées (i.e. mises à jour) à intervalle régulier (figure 3.9). Ainsi, toutes les modifications intervenues sur une région durant l'intervalle sont propagées vers les copies. Afin d'éviter des échanges inutiles d'information, cette propagation ne se fait que si la région a été effectivement modifiée.

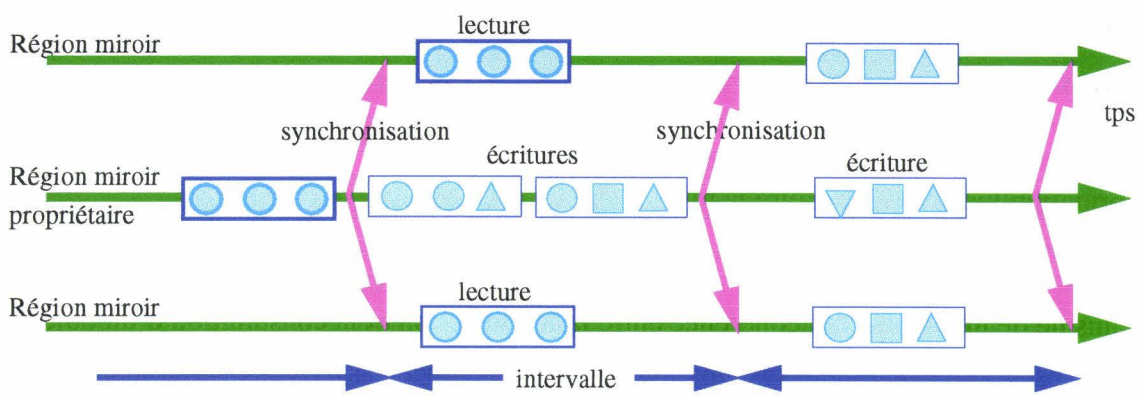



**figure 3.9** Propagation des modifications à intervalle régulier

L'intervalle séparant deux synchronisations automatiques n'est pas figé. Il peut être modifié à tout moment par le programmeur, et ce pour chaque région individuellement. Il est alors possible d'adapter la cohérence de chaque région au fur et à mesure de l'évolution de l'application.

Plus l'intervalle est grand, plus le nombre de messages nécessaire pour maintenir la cohérence est réduit, et plus le degré de cette cohérence est faible. Plus l'intervalle est court, plus le nombre de messages est important, et meilleur est le degré de cohérence.

C'est au programmeur de choisir la durée de l'intervalle de mise à jour en fonction du type d'objet qu'il place dans la DSM, et du degré de cohérence désiré. Examinons ce qui peut

guider son choix.

**Intervalle de mise à jour et cohérence**

Pour un objet constant, l'intervalle de mise à jour n'a pas d'influence : une fois l'objet initialisé et diffusé pour la première et unique fois à tous les processus, il n'est plus modifié, et par conséquent il n'y a pas de mise à jour.

Pour un objet faiblement variable, il faut comparer la fréquence de modification de l'objet, à la fréquence des mises à jour. Si ces deux fréquences sont similaires, la région est mise à jour pour pratiquement chaque modification. Les autres processus voient les modifications au fur et à mesure de leur venue. Le nombre de messages échangés est alors proche du nombre de modifications.

Si la période de mise à jour est plus grande que la période séparant deux modifications, les processus ne verront pas toutes ces modifications. La cohérence est alors "moins bonne" que précédemment, mais le nombre de messages nécessaires à la synchronisation est inférieur au nombre de modifications.

Le cas des objets fortement variables ressemble à celui des objets faiblement variables, mais comme la fréquence des modifications de la région est plus grande, il faut augmenter la fréquence des mises à jour (diminuer l'intervalle entre deux synchronisations) afin d'obtenir une "bonne cohérence".

D'une façon générale, plus l'intervalle de mise à jour est grand, plus la cohérence d'une région est faible. De même, plus l'intervalle est petit, meilleure est la cohérence. Malheureusement, la réduction de l'intervalle augmente la participation de Dream, et donc le surcoût induit.

**Degré instantané de cohérence**

Le degré de cohérence est une notion approximative. Les mises à jour automatiques de façon asynchrone par le système permettent uniquement de dire que la cohérence est "bonne" ou "moins bonne", ce qui est une notion floue. Nous venons de voir que Dream permet au programmeur de contrôler ce degré en modifiant l'intervalle de mise à jour. Ce contrôle donne une moyenne de la cohérence sur une longue période regroupant plusieurs mise à jour.

Si maintenant on s'intéresse au degré de cohérence juste après une mise à jour, on peut dire que celui-ci est "fort", et qu'il décroît avec le temps depuis cette dernière mise à jour (figure 3.10). Dream permet de connaître ce degré de cohérence instantané, qui est proportionnel au temps écoulé depuis la dernière mise à jour. En fait, Dream autorise la consultation de la date de la dernière mise à jour d'une région. Cette date est une notion locale à un processus, car il n'existe pas d'horloge globale.

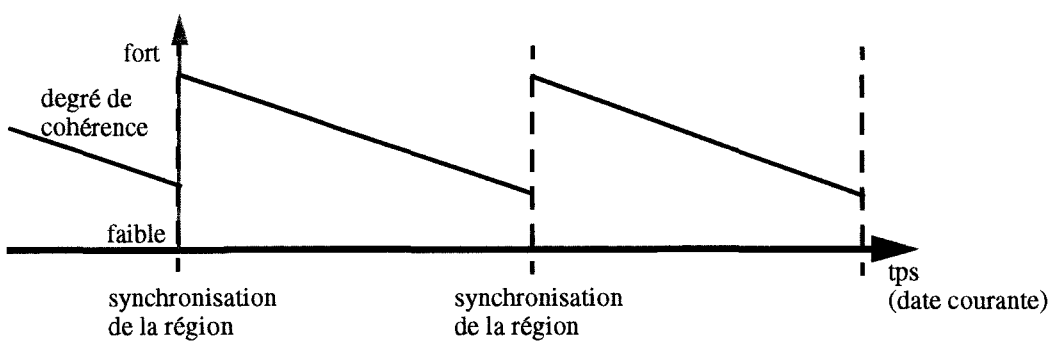



**figure 3.10** Degré instantané de cohérence

Ainsi, à partir de la date de mise à jour et de la date courante (figure 3.11), le

programmeur peut déduire depuis quand la région n'a pas été mise à jour, et ainsi se faire une idée du degré de cohérence de la région. En fonction de ce degré, il peut choisir d'effectuer une action ou une autre. Par exemple, il peut décider d'effectuer lui même une nouvelle mise à jour (voir *Mise à jour explicite*, page 69), ou attendre encore.

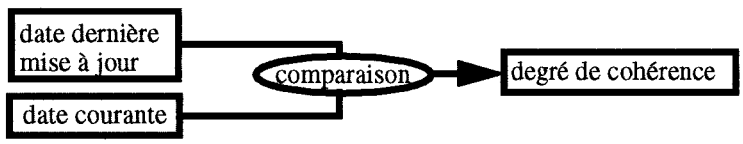



**figure 3.11** Obtention du degré instantané de cohérence

Prenons l'exemple d'un algorithme d'aide au placement. L'objectif est de répartir les nouveaux processus créés sur les sites les "moins chargés". Chaque site dispose d'un indicateur reflétant sa charge. Cet indicateur est en mémoire partagée. Il peut alors être consulté par tous les processus. La recherche d'un site se fait en comparant les indicateurs. Si deux indicateurs présentent la même valeur, il est possible de les départager en comparant leur degré de cohérence.

### 3.2.3 Programmation de la cohérence

La cohérence faible vue précédemment convient à certaines applications, mais pas à toutes. Pour les applications nécessitant une cohérence plus forte, Dream permet de contrôler plus précisément le degré de cohérence. Ceci se fait à l'aide d'opérations de synchronisations "programmées" (appelées) explicitement par le programmeur. Ces opérations permettent de contrôler explicitement les mises à jour, et ainsi de définir plus finement la cohérence d'une région

Elles peuvent être utilisées conjointement à la synchronisation automatique: le programmeur les appelle ponctuellement quand il veut une cohérence "forte". Ainsi il est possible de renforcer la cohérence à certain moment crucial de l'application, pour obtenir par exemple la plus récente valeur d'une région.

Les opérations de contrôle peuvent aussi être utilisées après avoir inhibé la mise à jour automatique d'une région. Le programmeur doit alors prendre lui-même en charge la gestion de la cohérence. Il programme les mises à jour, ce qui lui permet de définir son propre modèle de cohérence.

Les opérations possibles sont: la mise à jour explicite de la région, le gel de la région, l'inhibition des mise à jour automatiques, et la propagation automatique de l'état modifié d'une région.

Pour chacune des deux premières opérations, il faut distinguer le cas où l'opération est appliquée par le processus propriétaire de la région de celui où elle est appliquée par un autre processus. Dans les deux cas, la sémantique est la même, mais les effets sont sensiblement différents du fait que le processus a le droit ou non de modifier la région.

Les deux dernières opérations ne peuvent, quant à elles, être exécutées que par le processus propriétaire de la région.

**Mise à jour explicite**

La mise à jour explicite permet au programmeur de propager les modifications d'une région. Il faut distinguer la mise à jour de la région primaire, effectuée par le propriétaire, des mises à jour effectuées par les autres processus.

Quand une région est mise à jour par le processus qui en est propriétaire

(*synchronisation totale* figure 3.12), toutes ses régions miroirs sont mises à jour. Les modifications de la région sont propagées à chacune des copies qui détiennent alors la dernière version de la région. La région est complètement synchronisée.

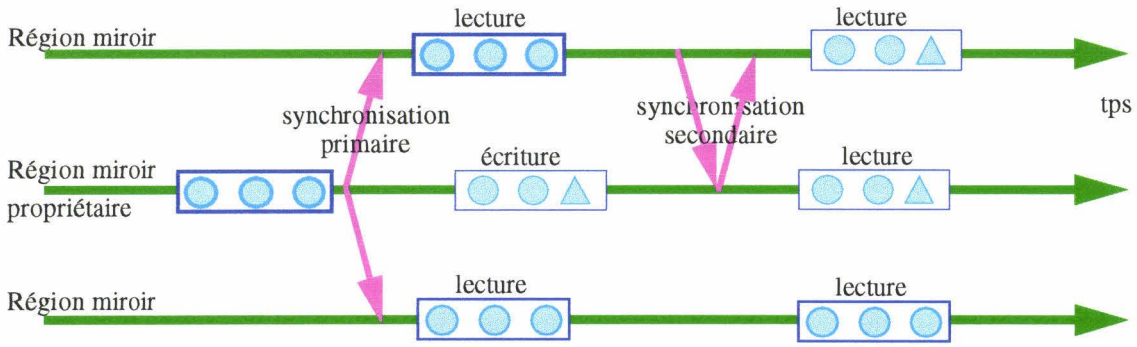



**figure 3.12** Mises à jours explicites

En appliquant cette mise à jour après chaque écriture dans une région, on propage les modifications qui viennent d'être effectuées. Ceci permet à un processus venant de modifier une région de diffuser immédiatement l'information aux autres processus. Ceux-ci voient alors la toute dernière modification. On obtient ainsi une cohérence qui peut être qualifiée de forte.

L'inconvénient de ce mécanisme est que la modification est transmise même à un processus ne consultant que sporadiquement la région.

Quand la mise à jour est effectuée par un processus autre que le propriétaire, seule la région miroir de ce processus est mise à jour (*synchronisation partielle* figure 3.12). Cette région miroir voit alors les dernières modifications intervenues dans la région miroir primaire du propriétaire. La région est partiellement synchronisée.

Cette mise à jour permet au programmeur d'obtenir les dernières modifications d'une région. Appliquée avant une lecture, elle assure d'obtenir les valeurs les plus récentes possibles (au temps de propagation de l'information près). Avec ce type de synchronisation, chaque processus peut réclamer les dernières modifications d'une région juste avant d'y accéder. L'inconvénient est que cette demande nécessite une interrogation du processus propriétaire (envoi d'une demande et attente d'une réponse).

**Gel d'une région**

L'idée derrière le gel d'une région est de permettre à un processus de consulter ou modifier une région sans être perturbé par une mise à jour.

Le gel d'une région n'agit que sur la région miroir du processus demandeur. Il permet d'isoler la région miroir du demandeur des régions miroir des autres processus. Ainsi, les actions effectuées sur la région par les autres processus n'atteignent plus la région miroir. Seul le processus demandeur peut encore agir sur sa région miroir

L'effet du gel d'une région est différent selon que la demande émane du processus propriétaire ou d'un autre processus.

Quand le processus propriétaire gèle sa région, il l'isole des autres processus (figure 3.13). Les mises à jour automatiques de la région sont suspendues, et les demandes de synchronisation venant des autres processus sont différées. Le processus propriétaire peut quant à lui modifier le contenu de la région, ou effectuer explicitement des mises à jour.

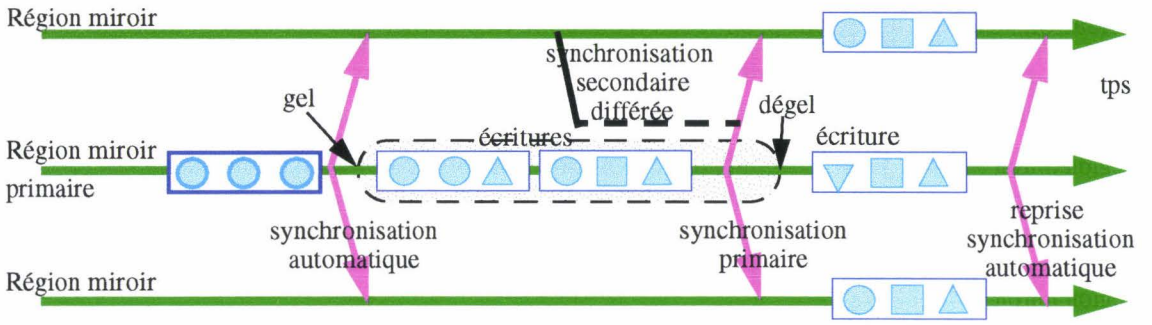



**figure 3.13** Gel d'une région miroir primaire

Le gel d'une région permet au processus propriétaire d'effectuer toute une série de modifications tout en étant certain que personne ne verra la région durant son état incohérent. Quand il a fini ses modifications, il dégèle la région, rendant ainsi les modifications accessibles par les autres processus.

Si besoin est, le processus propriétaire peut effectuer explicitement une synchronisation alors que la région est gelée. Les régions miroirs sont alors synchronisées, et les éventuelles demandes de synchronisation en attente sont honorées.

Le gel d'une région, effectué par un processus autre que le propriétaire, fige le contenu de la région pour ce processus (figure 3.14). Les mises à jour venant de l'extérieur, c'est à dire venant du propriétaire, sont alors interdites. Le processus est sûr que sa vue sur la région ne sera pas modifiée par un événement extérieur.

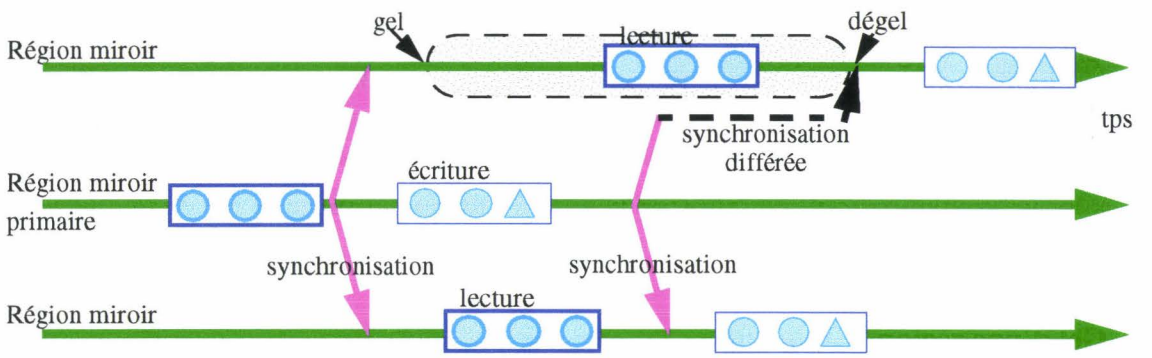



**figure 3.14** Gel d'une région miroir secondaire

Le processus peut alors consulter la région tout en étant certain que le contenu ne changera pas. Ceci est très important, notamment lors de la consultation de structures complexes. Prenons l'exemple d'une liste chaînée: imaginez les conséquences de la modification des liens de la liste alors qu'un processus est en train de la parcourir.

Le processus reste libre d'effectuer lui-même une synchronisation pendant le gel, celle-ci s'exécutera alors normalement.

Pendant la période de gel d'une région miroir, toutes les demandes extérieures de synchronisation sont différées. Elles sont honorées dès que la région est dégelée. Ainsi, il n'y a pas de perte d'information concernant la région.

Une région peut être gelée par un ou plusieurs des processus l'attachant, au gré du programmeur.

### Inhibition de la mise à jour automatique

L'inhibition de la mise à jour automatique d'une région interdit sa synchronisation périodique par le système. L'inhibition se fait individuellement pour chaque région. Elle ne peut être demandée que par le processus propriétaire de la région.

Quand la mise à jour automatique est inhibée, le programmeur prend alors lui-même en charge la synchronisation de la région. Il peut ainsi programmer son propre modèle de cohérence, en se servant des différents outils mis à sa disposition. L'inhibition évite des synchronisations parasites durant l'exécution d'un algorithme personnel de maintient de la cohérence.

Dans l'exemple de la figure 3.15, le programmeur a choisi de prendre connaissance des dernières modifications avant chaque lecture. Il inhibe les mises à jour automatiques, et effectue une synchronisation partielle avant une lecture. Cette façon de faire permet d'avoir une cohérence forte.

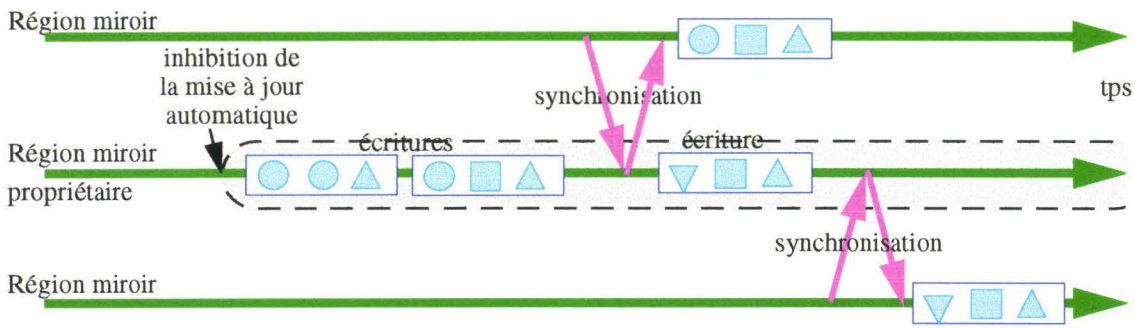



**figure 3.15** Inhibition de la mise à jour automatique

L'inhibition de la mise à jour automatique est implicite lorsqu'un processus gèle sa vue de la région. Elle diffère du gel par le fait que seul le propriétaire peut effectuer l'inhibition, et surtout par le fait que les demandes de synchronisation ne sont pas bloquées.

Pour le programmeur, l'inhibition de la mise à jour automatique se fait en demandant un intervalle de mise à jour «infini».

### Propagation de l'état modifié

Après la synchronisation d'une région par le processus propriétaire, toutes les régions miroirs voient les dernières modifications. Quand par la suite la région miroir du propriétaire est modifiée, son contenu diffère. On dit alors que la région est dans l'état modifié ou incohérent

Par défaut, seul le propriétaire sait que la région est incohérente. Les autres processus ne sont pas informés de la modification de la région, et ils ne savent pas que leur région miroir n'est plus à jour.

Le programmeur peut demander à Dream de maintenir l'état des régions miroirs: celles-ci sont alors averties de la modification de la région miroir primaire. Le programmeur peut alors consulter cet état, et savoir si il a affaire aux toutes dernières modifications, ou à une version un peu ancienne.

L'état indique non modifié dès qu'il est possible de dire que la région miroir reflète le contenu de la région (figure 3.16). Ceci arrive après une synchronisation par le propriétaire: toutes les régions miroir sont synchronisées. Cela arrive aussi après une synchronisation partielle: dans ce cas seul l'état de la région miroir synchronisée indique non modifié. La

région miroir primaire indique l'état réel de la région: si une des régions miroir n'est pas à jour, l'état est modifié.

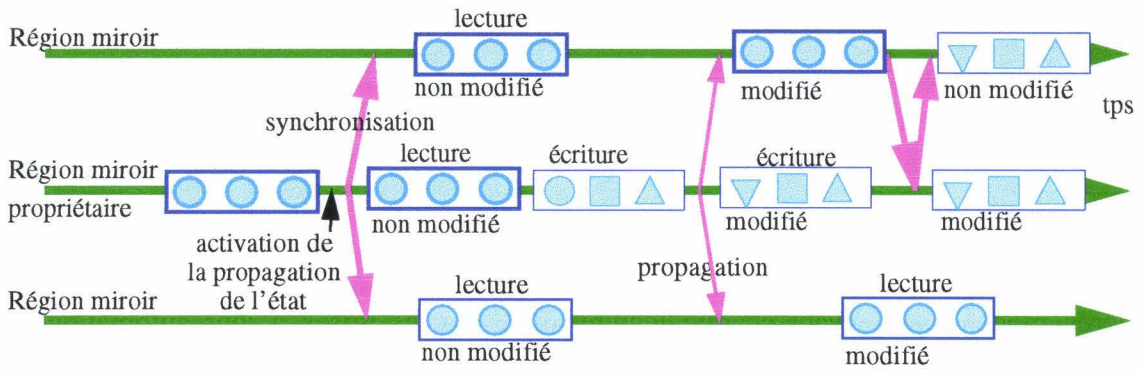



**figure 3.16** Propagation de l'état modifié

Avec la propagation de l'état modifié le programmeur peut contrôler précisément la cohérence d'une région. Il peut, selon l'état de la région, décider si une région miroir doit être mise à jour.

Notamment il est possible de réaliser un protocole d'invalidation: l'état modifié d'une région miroir correspond à une copie invalide. Quand un processus veut consulter une région, il vérifie sa validité, et effectue une mise à jour si nécessaire. Quand la région est modifiée, toutes les régions miroir passent dans l'état invalide.

L'inconvénient de la propagation de l'état modifié est qu'elle nécessite des échanges d'information supplémentaires. Néanmoins, J.B. Carter [Carter93] a montré dans sa thèse que pour certaines applications cet échange d'information supplémentaire permet de réduire significativement le nombre de messages nécessaire au maintien de la cohérence de la mémoire partagée.

## 3.3 Interface d'accès à la mémoire partagée

Dream se présente sous la forme d'une librairie proposant un ensemble de fonctions de manipulation de la mémoire partagée. Cette manipulation comporte trois phases:

- Une phase d'initialisation durant laquelle le programmeur détermine la structure de son application (la disposition des processus), identifie et déclare les régions à partager, puis crée ces régions, et ainsi le partage.
- Une phase d'utilisation qui correspond à l'exécution proprement dite de l'application. Durant cette phase le programmeur accède aux régions partagées par l'intermédiaire de leurs adresses (références) ou de leurs noms (identificateurs). Il peut ainsi rechercher une région, et y projeter un ou plusieurs objets partagés. Il peut aussi choisir de partager un objet à l'aide de plusieurs régions. Enfin, il peut consulter un certain nombre d'attributs liés à une région, comme sa taille, son nom, son adresse, ses droits d'accès, ...
- Une phase de terminaison où les régions partagées sont soit détruites soit détachées.

Etudions plus en détails ces différentes phases.

### 3.3.1 Initialisation

#### Structure d'une application

Une application type est constituée d'un ensemble de processus coopérant. La coopération consiste en la mise en commun d'informations et se fait naturellement par la mémoire partagée de Dream.

Les processus formant l'application sont indépendants les uns des autres. Ils n'ont pas besoin d'avoir un lien de relation, ni au niveau de la structure, ni au niveau de la parenté.

Chaque processus peut donc présenter une structure différente, c'est à dire avoir un code différent des autres processus. Ceci permet de ne pas suivre le modèle SPMD utilisé par beaucoup de DSM, et dans lequel chaque processus a la même structure. Avec Dream, chacun des processus contient uniquement les fonctionnalités dont il a besoin.

Les processus n'ont pas non plus besoin d'avoir un ancêtre commun. Chacun peut être lancé individuellement par le programmeur, à partir de n'importe quel site de l'architecture. Un processus peut bien évidemment aussi être lancé à partir d'un autre processus.

Comme les processus sont indépendants, ils peuvent être ajoutés ou supprimés dynamiquement durant l'exécution de l'application. Ainsi le programmeur ajoute des processus quand il en a besoin, et les détruit quand ils deviennent inutiles

Ceci donne au programmeur une grande liberté d'action quant à la structure de son application. Il peut utiliser le modèle maître - esclave dans lequel un processus maître lance plusieurs esclaves et attend le résultat de ceux-ci. Il peut aussi décider de lancer un processus qui lance d'autres processus, qui lancent à leur tour d'autre processus, formant ainsi une arborescence de processus. Il peut encore choisir de lancer chaque processus un par un, au fur et à mesure des besoins de l'application.

Prenons l'exemple d'une application dans laquelle plusieurs utilisateurs dialoguent par l'intermédiaire de zones de dialogue (figure 3.17). Chaque utilisateur dispose de sa propre zone de dialogue, et voit l'ensemble des zones des autres utilisateurs. Une zone de dialogue sera constituée d'une région partagée. Chaque utilisateur écrit dans sa zone-région, rendant ainsi son texte visible des autres. Ici l'application est constituée d'un ensemble de processus lancés individuellement et n'ayant par conséquent aucun ancêtre commun.

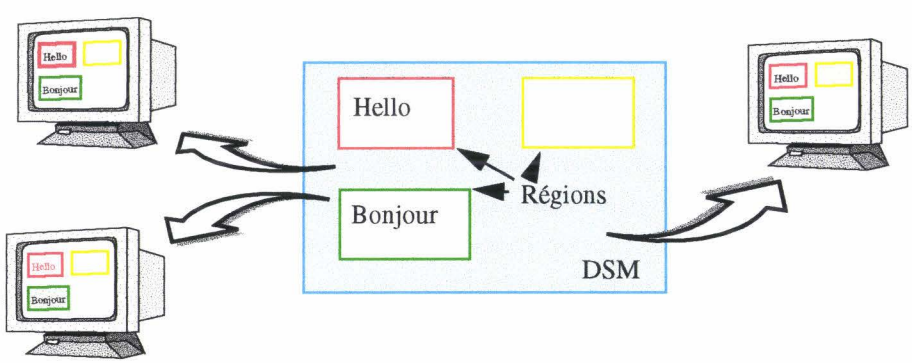



**figure 3.17** Application de dialogue

#### Identification d'une région

Dans Dream chaque région est identifiée par un nom unique dans toute l'application. Ce nom permet de désigner la région, sans erreur possible. Le programmeur se sert de ce nom pour attacher et manipuler la région.

Une région est aussi identifiée par les adresses qu'elle contient. Nous verrons plus loin que l'adresse d'une région est la même dans tous les processus. Ainsi cette adresse permet de désigner la région, là aussi sans erreur possible. Il en est de même de toutes les adresses de la zone de mémoire partagée de la région: toutes désignent la région.

Le programmeur peut alors utiliser aussi bien le nom de la région que l'une de ses adresses. Les adresses ont l'avantage sur le nom d'être implicitement associées à l'objet partagé. Il n'est donc pas nécessaire de les mémoriser quelque part dans l'objet.

Il n'y a pas double emploi entre l'identification par nom et l'identification par adresse, car seul le nom peut être défini par le programmeur. Le choix du nom permet de créer des processus indépendant attachant des régions communes. Le nom est alors choisi statiquement lors du développement de l'application, et n'a pas besoin d'être communiqué dynamiquement aux processus.

Si on reprend l'exemple de l'application de dialogue, chaque nouvel utilisateur ne connaît pas les utilisateurs déjà présent. Il prend connaissance de ces utilisateurs par l'intermédiaire d'une région commune dont le nom a été choisi par le programmeur. Cette région contiendra par exemple l'adresse ou le nom des régions servant au dialogue, régions qui pourront alors être attachées.

**Déclaration d'un objet partagé**

La déclaration d'un objet partagé est identique à celle d'un objet non partagé, mais la création d'un objet partagé ne peut être que dynamique: il faut déclarer une référence sur cet objet. L'objet lui même sera alloué par le programme lors de l'exécution.

L'exemple de la figure 3.18, montre la déclaration des deux principaux objets partagés de l'application de dialogue: *participants*, contenant la liste des participants, et *ma_fenetre*, représentant la fenêtre de dialogue. Les types *ListParticipants* et *FenetreDialogue* sont des types définis par le programmeur, ne contenant aucune spécification de partage concernant Dream.

```
ListParticipants *participants;
FenetreDialogue *ma_fenetre;
```

**figure 3.18** Déclaration d'objets partagés

La déclaration de référence sur les entités partagées présente l'avantage sur la déclaration statique de ne pas avoir à introduire de nouveaux éléments de langage, et de ne pas avoir besoin de faire appel à un préprocesseur ou à un nouveau compilateur pour traduire ces éléments de langage. Le type «référence sur un objet» est la plus part du temps présent dans les langages de programmation, et ne demande pas la création d'un nouveau type. L'inconvénient est que la déclaration d'un objet partagé, c'est à dire de sa référence, doit être suivie de son allocation avant la toute première utilisation.

**Création du partage**

La création du partage se fait à l'aide des opérations de création et d'attachement (figure 3.19). La première opération permet de créer une région potentiellement partageable par l'ensemble des processus. L'opération d'attachement permet, quant à elle, à un processus de projeter cette région dans son espace d'adressage, et ainsi d'y accéder effectivement. L'attachement est fait automatiquement pour le processus créateur.

La création et les attachements d'une région se font dynamiquement durant l'exécution de l'application. De plus, une région n'a pas besoin d'être attachée par tous les processus de l'application. Ainsi, un processus crée ou attache une région uniquement quand il en a besoin.

```
create( taille, [nom] )
attach( nom | référence )
```

**figure 3.19** Fonctions de création et d'attachement

Lors de la création d'une région, le programmeur doit préciser sa taille. Dream crée alors la zone de mémoire partagée ayant la taille demandée, et ajoute la nouvelle région à l'espace virtuel partagé. La taille est au minimum d'un mot mémoire, et n'a pas de limite supérieure, si ce n'est la limite de la capacité mémoire de la machine. Il est donc possible de créer des régions de la taille exacte des données à partager. La taille n'est plus modifiable après la création de la région. Il faut prendre garde à ce qu'elle soit suffisante pour que la région puisse accueillir tout ce que le programmeur veut y mettre.

Le créateur devient le propriétaire de la région, et comme tel a le droit de la modifier. Le programmeur profite en général de ce droit afin d'initialiser son ou ses objets dans la région.

Par défaut Dream fournit un nom à toute nouvelle région. Ce nom peut alors être communiqué par message ou partage, aux autres processus. Prenons l'exemple d'une application où les processus sont lancés par un processus racine. Ce processus crée les régions qui seront utilisées par l'application. Il lance ensuite les autres processus en leur passant les noms choisis par Dream, par exemple dans une région, un message ou dans la ligne d'arguments. Les processus attachent les régions avec leurs objets partagés, puis commencent les calculs.

Le programmeur peut aussi imposer le nom d'une région. Dans ce cas Dream vérifie que le nom n'est pas déjà utilisé. La région est créée seulement si elle n'existe pas déjà. Si elle existe déjà, une erreur est retournée. Il faut donc faire attention à ne pas utiliser un nom employé par ailleurs. De plus, le choix d'un nom interdit l'exécution simultanée de deux applications indépendantes utilisant un même nom pour une région.

L'attachement d'une région ne nécessite que son identification, soit par son nom, soit par une de ses adresses. Il n'est pas nécessaire de rappeler la taille, Dream se charge de retrouver ce paramètre, ainsi que d'autres nécessaire à la gestion de la région.

Le programmeur peut donc créer un objet partagé, et communiquer sa référence aux autres processus. Ces processus peuvent alors projeter l'objet dans leur espace d'adressage grâce à cette référence.

La figure 3.20 donne l'algorithme correspondant à la création du partage dans l'application de dialogue. Un processus, correspondant à un nouvel utilisateur, commence par essayer de créer la région commune. Si il y arrive, il l'initialise. Si il n'y arrive pas, il l'attache et se sert des informations qu'elle contient. Ensuite, le processus crée sa région qui servira au dialogue, et attache les régions de dialogue déjà existantes. On remarque que dans la version actuelle de Dream, le nom d'une région est un numéro. Ceci est discuté dans le chapitre 4 *Implémentation de DREAM*, page 91.

```
RgnMirror      rgn_participants, ma_rgn;     // descripteurs locaux des régions

rgn_participants.create( taille(ListParticipants), nom_rgn_participants );

si la création a réussi alors
   initialiser la liste des participants;
 sinon
   rgn_participants.attach( nom_rgn_participants );
   consultation de la liste des participants ...;
fin si

ma_rgn.create( taille(FenetreDialogue) );
attacher les régions de dialogue des autres utilisateur ...;
```

**figure 3.20** Création du partage

### 3.3.2 Utilisation

#### Adresse d'une région

Dans Dream, une région à la même adresse dans tous les processus d'une application. Ainsi un objet projetté dans une région aura la même adresse ou référence dans tous les processus ayant attaché la région.

Les avantages sont que les références peuvent être utilisées directement, sans qu'il soit nécessaire de les convertir vers une "adresse locale". De plus, la référence d'un objet étant la même dans chaque processus, il est possible de la communiquer directement de l'un à l'autre: La référence peut être placée directement soit en mémoire partagée, soit dans un message, sans avoir à invoquer un mécanisme de conversion. Ainsi il est possible de construire des structures complexes partagées, structures référençant d'autres structures elles-même partagées.

Nous avons retenu l'accès direct par des adresses identiques sur tous les processus, à cause de la plus grande souplesse d'utilisation et de sa transparence d'utilisation. Le problème du choix de l'adresse est pris en charge par Dream lors de la création d'une région, et n'apparaît nullement au programmeur. Ce problème est étudié page 104.

#### Recherche d'une région

L'opération d'attachement d'une région permet au programmeur de rechercher cette région dans la Dsm, et de la projeter dans l'espace d'adressage du processus. Une fois attachée, la région reste dans l'espace d'adressage du processus jusqu'a ce qu'elle soit explicitement détruite par le programmeur. Les accès à la région, et par conséquent à l'objet (ou aux objets) qu'elle contient, ne nécessitent pas d'autres attachements.

Cependant, il arrive que le programmeur perde la «trace» de l'attachement, souvent parce qu'il ne veut pas se soucier de mémoriser cette «trace». Il peut alors redemander l'attachement de la région. Cette nouvelle opération d'attachement fournit alors la même région miroir que celle obtenue lors du premier attachement, et n'entraîne pas de trafic de messages.

L'opération d'attachement permet donc non seulement de projeter une région dans l'espace mémoire d'un processus, mais elle permet aussi de retrouver une région déjà projetée. Ainsi un processus peut attacher plusieurs fois une même région sans se soucier de savoir si l'opération a déjà été faite.

Il existe aussi une fonction permettant de rechercher une région miroir localement dans le processus (figure 3.21). Bien sûr, la région miroir n'est trouvée que si la région a déjà été attachée par le processus. Cette fonction est utile quand par exemple un processus dispose

d'une liste de régions dont seule quelques unes sont attachées. Le processus veut alors détacher ces régions. Si seul l'opération d'attachement existait, le processus serait obligé d'attacher chaque région de la liste pour ensuite la détacher. Avec l'opération de recherche, seules les régions effectivement attachées sont retrouvées, évitant ainsi des attachements inutiles.


**figure 3.21** Fonctions de recherche d'une région

L'attachement ou la recherche d'une région se fait à partir du nom de la région, ou à partir d'une des adresses de la zone de mémoire partagée de la région. Ainsi, il est possible de rechercher une région à partir d'un des objets qu'elle contient. Les objets d'une région n'ont alors pas besoin de mémoriser le nom de la région les contenant.

Reprenons l'exemple de l'application de dialogue de la page 74. Dans cet exemple, la liste des régions des utilisateurs est contenue dans une région commune. Chaque nouveau processus consulte la liste afin d'attacher les régions déjà existantes, puis il y ajoute le nom de sa région. Mais quand un ancien processus consulte la liste, il doit attacher la nouvelle région. Une solution simple est d'attacher périodiquement toutes les régions de la liste partagée, seules les nouvelles régions entraînent un trafic de messages. Les régions déjà attachées sont retrouvées par Dream.

**Accès à un objet partagé**

Dans Dream, l'accès à un objet partagé se fait par sa référence, ou pointeur, sur l'objet. Cette référence est identique à une référence sur un objet non partagé. Le programmeur manipule donc l'objet partagé de la même manière qu'il manipule un objet non partagé.

Pour projetter un objet partagé dans l'espace d'adressage d'un processus, il faut projetter cet objet dans une région. Le plus simple est alors de faire coïncider la zone de mémoire partagée (la région) avec la référence sur l'objet. Ceci se fait en affectant la référence de l'objet avec le début de la région, grâce à la fonction retournant l'adresse d'une région (figure 3.22).


**figure 3.22** Adresse d'une région

Cette fonction peut aussi servir à retrouver un objet à partir une région, à condition que cet objet ait été placé au début de la région.

Une fois la région attachée, l'objet est toujours directement accessible. Il n'est pas nécessaire de spécifier le début ou la fin des accès à l'objet.

La figure 3.23 donne l'algorithme correspondant à la création de la région commune pour l'application de dialogue. Cette création est effectué par le premier processus utilisant la région qui se charge alors d'initialiser l'objet *participant* contenu dans la région.

```
rgn_participants.create( taille(ListParticipants), nom_rgn_participants);
   ...
   // Projection de l'objet et initialisation
participants = rgn_participants.startAddr();
participants->nb_participant = 0;
```

**figure 3.23** Création et accès à un objet

La figure 3.24, elle, donne l'algorithme correspondant à l'attachement qui est effectué par les autres processus. Une fois la région attachée, un processus peut y accéder en lecture, afin de consulter la liste des participants.

```
rgn_participants.attach( id_rgn_participants );
   ...
   // Projection de l'objet et consultation
participants = rgn_participants.startAddr();

pour chaque utilisateur dans la liste faire :
   afficher( utilisateur->noms );
fait
```

**figure 3.24** Attachement et accès à un objet

La référence d'un objet peut être partagée sans aucune difficulté. A partir de cette référence, n'importe quel processus peut attacher la région associée à l'objet, et ainsi accéder à cet objet.

Il est alors possible de créer des structures partagées se référençant les unes les autres. Notamment, il est possible de créer des structures partagées telles que des listes chaînées dont chaque maillon est partagé dans une région, ou bien encore des arbres dont les noeuds sont eux aussi partagés.

**Récupération des erreurs d'accès**

Quand un processus accède à une région qu'il n'a pas attachée, Dream génère une erreur d'accès. De même, un processus essayant d'écrire dans une région sur laquelle il n'a pas le droit d'écriture génère aussi une erreur d'accès.

Par défaut, ces erreurs d'accès provoquent la terminaison du processus. Dream donne alors le nom de la région fautive et indique quel genre d'erreur d'accès vient de ce produire, région non attachée ou absence du droit d'écriture.

Dream permet de récupérer ces erreurs d'accès, et d'exécuter une action. Le programmeur choisira en général de réparer l'erreur, soit en attachant la région, soit en demandant le droit d'écriture. Après la récupération de l'erreur et exécution de l'action, le programme reprend l'instruction ayant déclenché l'erreur.

La récupération d'une erreur d'accès permet au programmeur de ne pas se soucier de l'attachement d'une région avant le premier accès. Ainsi il est possible de parcourir une liste chaînée partagée ou un arbre partagé sans avoir à attacher systématiquement les régions associées. Quand le processus accède à un maillon ou un noeud dont la région n'est pas attachée, une erreur est déclenchée. Celle-ci est récupérée, ce qui permet d'attacher la région.

**Plusieurs objets dans une région**

Nous avons déjà vu qu'une région peut accueillir plusieurs objets à condition que sa

taille soit suffisante. Une région contenant plusieurs objets peut alors être vue comme un tas.

Dream fournit une fonction d'allocation et une fonction de libération d'objets dans ce tas. Ces fonctions gèrent l'espace partagé de la région, dans les limites de la taille spécifiée lors de la création. Le programmeur est libre d'utiliser ces fonctions ou de gérer lui-même l'espace de mémoire de la région.

L'application de dialogue fournit ici encore un exemple. La région commune contient une liste des noms ou références de toutes les régions participant au dialogue. Pour faciliter l'insertion, la suppression ou le parcours de ces noms, il est possible d'implémenter cette liste sous la forme d'une liste chaînée qui est créé dans la région commune (figure 3.25). Chaque maillon représente un participant. La création ou la destruction d'un nouveau maillon est aisée grâce à l'allocation ou la libération d'une sous-zone dans la région. La tête de liste sert de point d'entrée pour le parcours des objets dans la région. Elle est facilement retrouvée par un processus si le programmeur prend garde de la placer à une adresse bien connue de la région, comme l'adresse de début.

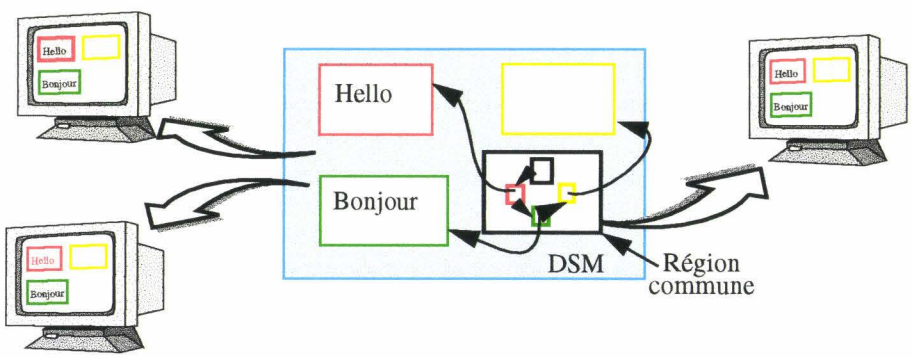



**figure 3.25** Liste partagée dans une seule région

### Plusieurs régions pour un objet

Plus un objet est volumineux, plus le problème du faux partage risque de se poser. Prenons l'exemple d'une matrice résultat dont chaque colonne est calculée par un processus différent. Chaque processus calcule case après case, et il y a une forte probabilité pour que deux processus veulent écrire dans la matrice au même moment.

Une solution est de partager chaque colonne à l'aide d'une région (figure 3.26). Chaque processus peut alors modifier sa colonne sans interférer avec les autres. Le problème est que dans la représentation mémoire d'une matrice, les colonnes sont placées les unes à la suite des autres, c'est à dire à des adresses contiguës. Or Dream ne permet pas de choisir l'adresse d'une région.

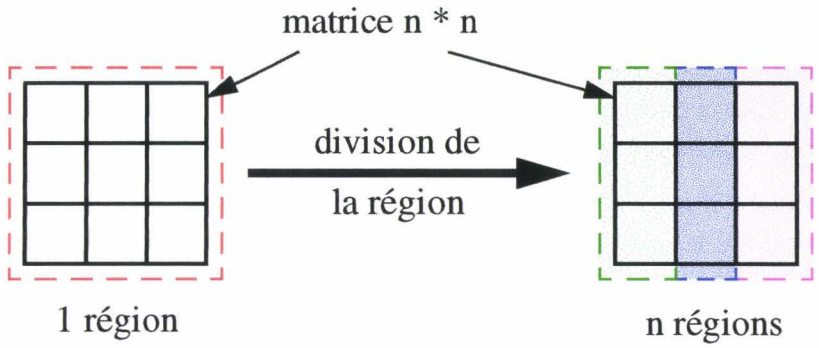



**figure 3.26** Division d'une région

Par contre, Dream permet de diviser une région (figure 3.27), formant ainsi plusieurs régions à part entière, mais dont les adresses sont contiguës. La région de départ existe toujours, mais sa taille a changé. Ainsi il est possible de diviser un objet et de le partager dans plusieurs régions. De plus cette division n'entraîne pas un morcellement de l'objet: chaque partie reste contiguë aux autres. L'accès à l'objet se fait toujours de la même façon.

**split( offset [, nouveau nom] )**

**figure 3.27** Fonctions de division d'une région

Pour résoudre le problème constitué par la matrice résultat, le programmeur crée une région de la taille de la matrice, puis il divise cette région en autant de régions que nécessaire. Il peut maintenant modifier chacune des colonnes individuellement et simultanément. Il peut aussi consulter l'ensemble de la matrice à partir d'un autre processus.

L'utilisation de plusieurs régions pour partager un objet permet donc d'éviter le problème du faux partage. Elle permet aussi à plusieurs processus de modifier simultanément un même objet.

**Attributs**

Une région n'est pas uniquement constituée d'une zone de mémoire partagée. Elle possède aussi un certain nombre d'attributs consultables par le programmeur. Certains de ces attributs sont constants et ne changent pas tout au long de la vie de la région. D'autres sont variables et servent à prendre connaissance de l'état de la région. Ces attributs variables sont souvent différents d'un processus à l'autre, et ce pour une même région. Ainsi, si un processus voit le droit d'accès en lecture-écriture pour une région, les autres processus voient ce droit comme étant en lecture seule.

Les attributs consultables par le programmeur sont les suivants:

- nom de la région: C'est le nom de la région qui permet de la différencier de toutes les autres régions au sein d'une application. Quand le nom est choisi par le système, la consultation de cet attribut est le seul moyen de connaître ce nom.
- taille de la région: La taille de la région telle qu'elle a été fixée par le programmeur lors de la création de la région.
- adresse de base de la zone partagée: Donne l'adresse de base de la région. La dernière adresse de la région est calculable en ajoutant la taille de la région à l'adresse de base.

- droit d'accès: Donne le type d'accès autorisé sur la région. Il peut être en lecture-écriture (Read-Write), auquel cas le processus a le droit de lire et d'écrire dans la région. Seul le propriétaire peut avoir ce droit. Il peut être en lecture seule (Read Only), auquel cas le processus n'a que le droit de lire dans la région. Si il essaie d'écrire, il provoquera une erreur mémoire qui entraîne en général la terminaison du processus.
- type de région: Primaire ou Secondaire. Le droit d'accès sur une région ne suffit pas pour reconnaître le propriétaire de la région. En effet, le propriétaire peut abandonner son droit d'écriture (voir 3.4.1 page 84), et n'avoir plus que le droit de lecture. Pour différencier le processus propriétaire des autres processus, il faut regarder le type de la région miroir. La région miroir du propriétaire est appelée région miroir primaire, et est de type primaire, alors que les régions miroirs des autres processus sont appelées régions miroirs secondaires, et sont par conséquent du type secondaire.
- date de la dernière synchronisation: C'est la date de la dernière synchronisation de la région miroir. Si la région miroir est primaire, c'est la date où toutes les régions miroir ont été synchronisée. Si la région miroir est secondaire, c'est la date où cette région à été synchronisée avec celle du propriétaire. Cette date est modifiée aussi bien par les synchronisations automatiques que par les synchronisations explicites. Elle permet d'avoir une idée du degré instantané de cohérence de la région.
- indicateur de modification: Il indique si une région miroir reflète exactement le contenu de la région. Cet indicateur ne fonctionne que si la propagation de l'état modifié de la région est en service. Le fonctionnement de l'indicateur à été décrit auparavant (Voir *Propagation de l'état modifié*, page 72).

### 3.3.3 Terminaison

Si le programmeur peut créer des régions dynamiquement, il peut aussi les détruire quand il n'en a plus besoin.

La destruction d'un région qui ne sert plus est nécessaire afin de libérer les ressources qu'elle occupe. En effet, une région n'est pas seulement constituée d'un ensemble de régions miroirs, mais aussi de structures annexes destinées à la gestion du partage.

Une région est constituée d'une ou plusieurs régions miroirs, détenue chacune par un processus. Un processus n'ayant plus besoin d'une région peut vouloir soit détruire toutes les ressources de la région, dans tous les processus et nous disons alors qu'il y a destruction de la région, soit détruire uniquement les ressources locales de la région sans s'occuper des ressources utilisées dans les autres processus et nous disons qu'il y a détachement de la région.

Le programme doit libérer les ressources des régions qui ne servent plus, et qui ne serviront plus. C'est à lui de choisir entre la destruction totale de la région ou le simple détachement de la région miroir.

Lors de la terminaison d'un processus, les régions qui ont été attachée sont automatiquement détachées. Ainsi, si le programmeur a oublié des régions, ou s'il n'a pas voulu se soucier de leur détachement, Dream s'en charge pour lui.

**Destruction**

La destruction d'une région entraîne, comme signalé précédemment, la destruction de toutes les ressources qu'elle occupe sur tous les processus.

Elle entraîne notamment la destruction des régions miroirs détenues par chaque processus et par conséquent, la destruction des objets partagés qu'elle contient. Il est de la

responsabilité de l'application de faire attention à ce qu'un processus n'accède plus à un objet partagé qui a été détruit. En effet, comme Dream autorise des accès directs aux objets partagés, il ne lui est pas possible de contrôler les accès réalisés par le programmeur et de lui interdire l'accès à une région qui n'existe plus.

Afin d'avertir un processus de la destruction d'une de ses vues sur une région, Dream permet, et ce pour chaque région, de spécifier une fonction qui sera appelée lors de la destruction de la région miroir. Cette fonction est entièrement définie par le programmeur, qui peut s'en servir afin de détruire les références détenues par le processus sur la région.

La destruction d'une région est une action brutale et autoritaire. Elle n'est autorisée que pour le processus propriétaire de la région. Elle doit être utilisée quand le programmeur est sûr que plus aucun processus de l'application n'accède à la région. Le programmeur lui préférera le détachement individuel des régions, qui est une méthode plus douce.

**Détachement**

Le détachement d'une région par un processus permet à ce processus de détruire les ressources occupées par la région dans ce processus. Les ressources utilisées dans les autres processus ne sont pas affectées. Ainsi, seule la région miroir du processus est détachée, c'est à dire détruite. Comme pour la destruction, c'est au programmeur de s'assurer que le processus n'accède plus à la région détruite ou à un objet qu'elle contient.

Le détachement d'une région par le processus qui en est propriétaire pose un problème supplémentaire. En effet, que faire du titre de propriétaire? La solution employée par Dream consiste à transférer le titre à l'un des processus auquel est encore attachée la région. Ce processus se voit confié le titre de propriétaire, mais pas le droit d'écriture. Ce mécanisme garantit qu'il y a toujours un processus, le propriétaire, pour gérer la région.

Le nouveau propriétaire n'hérite pas du droit d'écriture afin que son comportement ne change pas vis à vis des accès à la région. Ainsi le processus ne se retrouve pas soudainement avec la possibilité d'écrire dans la région. Pour pouvoir le faire, il doit comme normalement en demander explicitement le droit (voir 3.4.1 page 84).

Si la région n'est plus attachée que par un seul processus, et que celui-ci la détache à son tour, la région est alors totalement détruite. Cela revient à détruire la région, mais cette destruction est plus douce que celle vue précédemment : chaque processus détache lui-même sa vue sur la région, et quand la région n'est plus visible par aucun processus, elle est complètement détruite. Il n'est plus possible ni d'y accéder, ni de l'attacher.

Ainsi, quand un processus est sûr de ne plus se servir d'une région, il peut la détacher, libérant ainsi les ressources qu'elle occupe au sein de ce processus. Quand une région n'est plus attachée par aucun processus, elle est automatiquement détruite, et elle n'existe plus pour Dream.

Le détachement des régions d'un processus se fait automatiquement à la fin d'un processus, mais le programmeur peut détacher ou détruire lui-même une région quand il est certain de ne plus en avoir besoin.

## 3.4 Synchronisation inter-processus

Les régions, et les objets qu'elles contiennent, sont accédées par les différents processus de l'application. Se pose alors l'inévitable problème de la synchronisation des accès aux objets, ou synchronisation inter-processus.

Dream ne propose pas, à proprement parler, d'outils de synchronisation des accès à une région. Ceux-ci ne sont en général pas nécessaires. La structuration d'une région en région

miroir permet à plusieurs processus d'accéder simultanément à une même région, et comme chaque processus accède à sa propre copie, il n'y a pas de conflits d'accès. De plus, le propriétaire unique évite les conflits lors des écritures.

Cependant, le programmeur peut avoir besoin de synchroniser les accès à une région. Par exemple, il peut vouloir synchroniser les écritures et les lectures. La notion de propriétaire peut, dans une certaine mesure, servir à la synchronisation. Nous allons donc commencer par étudier cette notion. Puis nous aborderons les possibilité de synchronisation des accès à partir de la synchronisation des régions. Enfin nous allons voir qu'il est possible d'utiliser à tout moment la synchronisation implicite des échanges de messages.

### 3.4.1 Propriétaire d'une région

Dans le modèle Dream, seul le propriétaire d'une région a le droit de la modifier. Tous les autres processus ont potentiellement le droit d'accéder à cette région, mais seulement en lecture. De plus, une région ne peut avoir, à un moment donné, qu'un seul propriétaire. Cette contrainte permet d'éviter les conflits d'accès à une région lors d'une écriture et réduit les problèmes de synchronisation rencontrés lors de la réalisation de la DSM. Cependant, cette contrainte rend impossible la modification de la région par un autre processus que le propriétaire.

Un certain nombre d'applications peuvent se contenter de toujours modifier une région à partir du même processus. C'est à dire avoir un seul processus rédacteur pour une région. Cependant, il est des cas où le programmeur préfère modifier un objet à partir de différents processus. Pour cela, Dream lui fournit un mécanisme de changement de propriétaire permettant le transfert du droit d'écriture.

Le titre de propriétaire et le droit d'écriture sur une région sont intimement liés. Quand un processus a le droit d'écriture sur une région, il est aussi propriétaire de la région. Mais un processus peut être propriétaire et ne plus avoir le droit d'écriture sur la région.

Pour Dream, un processus propriétaire d'une région est chargé de la gérer: c'est à lui de traiter les requêtes concernant cette région. Ce processus est assuré d'avoir une vue cohérente de la région, c'est à dire de voir les plus récentes modifications, y compris lorsque le processus vient juste d'acquérir le titre de propriétaire.

Lors de la création d'une nouvelle région, le processus créateur en devient automatiquement le propriétaire et a directement le droit d'écriture dans cette région. Il ne perdra ce droit d'écriture et son titre de propriétaire que par un abandon explicite.

#### Changement

Le changement de propriétaire est basé sur un mécanisme d'abandon-acquisition du droit d'écriture (figure 3.28). Un processus propriétaire d'une région abandonne son droit d'écriture sur la région qui peut alors être acquis par un processus en faisant la demande. L'acquisition du droit d'écriture inclut automatiquement l'acquisition du titre de propriétaire.

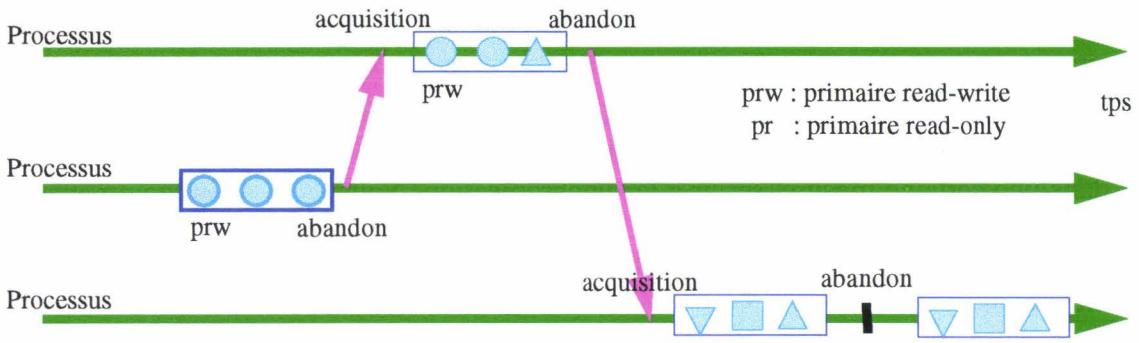



**figure 3.28** Changement de propriétaire

L'abandon et l'acquisition sont réalisés explicitement par le programmeur. Quand un processus demande le droit d'écriture sur une région, cette demande est satisfaite uniquement si ce droit n'est plus détenu par un processus. Si le droit d'écriture est encore détenu par un processus, le demandeur est bloqué jusqu'à ce que sa demande soit satisfaite. Le blocage du processus peut être évité en spécifiant une durée maximale d'attente (time out).

Plusieurs processus peuvent demander le droit d'écriture alors que celui-ci est encore détenu par un processus. Ces demandes sont enregistrées et sont honorées les unes à la suite des autres dans l'ordre de leur arrivée au processus propriétaire. Cet ordre n'est pas obligatoirement celui dans lequel les requêtes ont été faites car il n'existe pas d'horloge globale permettant de les ordonner globalement.

Un processus détenant le droit d'écriture, le garde jusqu'à ce qu'il l'abandonne explicitement. Il ne peut pas perdre ce droit d'écriture, même si d'autres processus essaient de l'acquérir. Les demandes d'acquisition ne sont satisfaites que si le droit d'écriture est explicitement abandonné par le processus.

Quand le propriétaire d'une région abandonne son droit d'écriture et qu'aucun processus ne veut l'acquérir, ce processus reste propriétaire «intérimaire» de la région. Il reste propriétaire dans le sens où il continue de gérer la région. Cependant, il n'a plus le droit d'écriture sur la région. Dès qu'un processus demande le droit d'écriture, ce processus devient le nouveau propriétaire de la région et l'ancien propriétaire redevient un processus quelconque pour la région.

L'acquisition peut bien sûr se faire par l'ancien propriétaire, qui redevient propriétaire à part entière. Dans ce cas, Dream ne génère pas d'échange de messages. Un processus peut acquérir le droit d'écriture juste le temps de modifier un objet, et l'abandonner aussitôt. Si le même processus reprend plusieurs fois de suite le droit d'écriture, Dream ne fait pas d'échange de messages inutiles.

**Section critique**

Le mécanisme d'abandon-acquisition du droit d'écriture permet de construire une section critique dans laquelle un processus est sûr d'être le seul à accéder à une région ou à ses objet. Pour cela, les processus s'entendent à ne consulter ou modifier la région que lorsqu'ils ont le droit d'écriture sur cette région. Ce droit est abandonné dès qu'ils n'accèdent plus à la région. Ainsi, quand un processus est dans la section critique, il est sûr d'être le seul à accéder à la région, car il est le seul à pouvoir détenir le droit d'écriture.

Une section critique réalisée de la sorte permet d'obtenir des objets fortement variables ayant une cohérence "très forte". Cette cohérence est "très forte" du fait que le processus

propriétaire voit toujours les plus récentes modifications. Ces objets sont comparables aux variables migratoires rencontrés dans Munin [Carter91] et Phosphorus [Demeure95].

Prenons l'exemple d'un compteur servant à dénombrer les processus d'une application. Le compteur est un objet partagé contenu dans une région créée par l'un des processus. Ce processus a automatiquement le droit d'écriture sur le compteur. Il initialise la valeur à un, puis abandonne le droit d'écriture pour que les autres processus puissent l'acquérir. Quand un nouveau processus est créé, il attache la région contenant le compteur, demande le droit d'écriture, incrémente la valeur, puis abandonne à son tour le droit d'écriture sur la région pour que les autres puissent l'acquérir. Ce processus est devenu propriétaire de la région le temps de modifier la valeur du compteur. Ainsi, chaque nouveau processus peut modifier le compteur, tout en étant sûr d'être le seul à pouvoir le modifier. Un processus qui se termine, demande le droit d'écriture, décrémente le compteur, puis abandonne aussitôt le droit d'écriture.

L'inconvénient de ce mécanisme est qu'il ne peut y avoir qu'un seul processus à la fois dans la section critique et que le mécanisme utilisé n'est pas performant.

**Notification de changement**

L'acquisition du titre de propriétaire peut venir soit d'une demande explicite de la part d'un processus, soit du détachement de la région miroir du précédant propriétaire (Voir *Terminaison*, page 82).

Dans ce dernier cas, la vue de la région (région miroir) détenue par le processus change de statut sous l'action d'un événement extérieur au processus. Le droit d'écriture n'est, quant à lui, pas positionné, afin de perturber le moins possible le comportement du processus recevant le titre de propriétaire.

Le processus peut demander la notification de l'événement et installer une fonction qui est appelée immédiatement après l'acquisition effective du titre de propriétaire. Le processus peut alors entreprendre n'importe quelle action requise, comme par exemple acquérir le droit d'écriture.

Dans la fonction installée, le processus peut différencier l'acquisition «normale» de l'acquisition due à un détachement en vérifiant le droit d'accès à la région. Si le droit est en écriture, le changement est dû à une acquisition normale de la part du processus. Si le droit d'accès est en lecture seule, le titre de propriétaire a été transféré autoritairement suite au détachement de la région miroir primaire.

La fonction notifiant un changement de propriétaire est aussi appelée lorsque le propriétaire perd effectivement son titre (il peut s'écouler un certain délai entre l'abandon du droit d'écriture et la perte du titre). Ceci permet au processus perdant effectivement le titre d'en être averti, et d'entreprendre des actions concernant la région. Il peut ainsi réacquérir le droit d'écriture afin de modifier une dernière fois la région. Ce droit est automatiquement perdu à la fin de la fonction et les modifications sont transmises au nouveau propriétaire.

### 3.4.2 Synchronisation d'une région

La synchronisation d'une région (mise à jour) peut servir à synchroniser les processus, ou les accès à la région. La synchronisation d'une région a déjà été étudiée dans le paragraphe consacré à la cohérence (3.2 page 65). Nous allons voir maintenant comment se servir des mises à jour pour synchroniser les processus.

Tout d'abord, un processus peut être bloqué en attente d'une modification dans une région. La mise à jour de la région débloque le processus qui est alors synchronisé.

Ensuite, un processus peut demander à être notifié des mises à jour qu'il subit. Ceci lui

permet de se synchroniser sur ces mises à jour.

**Attente d'une mise à jour**

Imaginons le problème suivant: un drapeau, initialement à faux, est partagé par plusieurs processus. L'un des processus modifie le drapeau, tandis que les autres attendent sa modification. La manière la plus simple d'attendre est de scruter le drapeau jusqu'à ce que sa valeur change.

L'inconvénient de cette méthode est que l'attente est active; le processus regarde constamment la valeur du drapeau. Dream permet d'éviter cette attente active, en bloquant un processus, jusqu'à ce qu'une région soit mise à jour.

Notre problème se résout en bloquant le processus sur l'attente de la mise à jour de la région contenant le drapeau. Dès que cette région est synchronisée, le processus est débloqué et il peut consulter la valeur du drapeau.

Si la région est gelée, elle ne peut normalement pas recevoir de mise à jour venant de l'extérieur. Si le programmeur se met malgré tout en attente d'une mise à jour, celle-ci risque de durer indéfiniment. Afin d'éviter ce problème, Dream «dégèle» temporairement la région autorisant ainsi l'arrivée des mises à jour extérieures.

Avec Dream, il est possible d'attendre une mise à jour plus récente qu'une date précise. Cette date est une notion locale au processus car il n'existe pas d'horloge globale. Si cette date est antérieure à la date courante, Dream regarde si il y a eu une mise à jour de la région depuis cette date, si oui, l'attente se termine immédiatement, si non, elle dure jusqu'a la prochaine mise à jour venant de l'extérieure. Si la date est postérieure à la date courante, L'attente durera jusqu'a ce qu'il y ait une mise à jour plus récente que la date demandée.

Prenons l'exemple d'une application dans laquelle un des processus est chargé d'afficher la valeur d'un objet partagé. Ce processus attend la modification de la région contenant l'objet, et affiche les données. Dès que l'affichage est fait, il attend la prochaine modification. Si l'affichage prend beaucoup de temps, et que la région est mise à jour durant cette période, le processus va se mettre en attente d'une modification, alors que celle-ci à déjà eu lieu. Pour éviter ceci, le processus mémorise la date à laquelle il consulte la valeur de l'objet, effectue l'affichage, et attend une mise à jour plus récente que celle qu'il vient de lire.

**Notification de mise à jour**

Un processus peut demander à être notifié des mises à jour subie par une région. Une fonction, écrite par le programmeur, est alors appelée à chaque mise à jour de sa vue sur cette région (mise à jour de la région miroir).

Pour le processus propriétaire de la région, la fonction est appelée juste avant la synchronisation effective de la région, c'est à dire après la demande de synchronisation. Cette demande peut venir d'une synchronisation automatique, d'une synchronisation explicite par le processus propriétaire, ou encore de l'aboutissement d'une demande de synchronisation venant d'un autre processus.

Pour les autres processus, la fonction est appelée immédiatement après la synchronisation de la région. Le programme normal est dérouté, et la fonction de l'utilisateur est exécuté. Le cours normal du programme est repris dès la fin de la fonction.

Prenons l'exemple de la figure 3.29: deux processus partagent une région contenant des objets. Chaque processus met en place une fonction lui permettant d'être notifié de la mise à jour de la région. Le propriétaire modifie un objet, puis synchronise la région. Sa fonction est alors appelée, et lui permet de modifier un autre objet. La région est ensuite synchronisée, et le

processus propriétaire continu son cours. Quand le second processus "reçoit" la synchronisation, son cours normal est interrompu, et la fonction mise en place est appelée. La consultation des objets donne les dernière valeurs écrites. A la fin de la fonction, le cours normal du processus est repris.

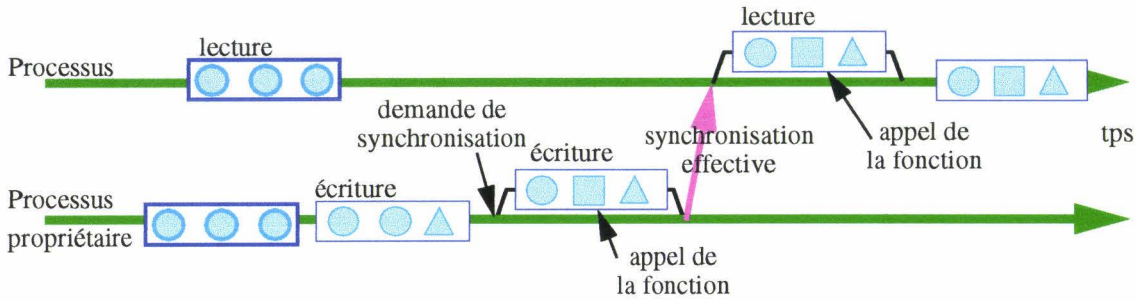



**figure 3.29** Notification de mise à jour

### 3.4.3 Synchronisation par messages

Dream est un modèle hybride étendant les possibilités des processus communicants par des possibilités d'accès à une mémoire partagée. Dream fournit la partie mémoire partagée, qui est elle-même bâtie sur les processus communicants. Mais les fonctionnalités de communication restent accessibles. Ainsi, il est possible d'utiliser les messages, ou tout autre outil fournit par la couche de communication afin de synchroniser les processus.

De plus, Dream donne la possibilité d'obtenir les ports des processus attachant une région. Ces ports permettent d'envoyer un message a ces processus. Un processus propriétaire peut obtenir l'ensemble des ports des processus attachant la région, tandis qu'un processus quelconque peut obtenir le port du propriétaire.

Ceci permet, à l'aide de messages, de synchroniser un ensemble de processus attachant une même région.

## 3.5 Conclusion

L'essentiel de ce chapitre a été consacré à la présentation du modèle Dream. Cette présentation s'est globalement articulée autour des trois points suivants:

**Structuration de l'espace partagé**

Une des particularités du modèle Dream est de proposer une structuration de l'espace partagé en regions. La region est l'unité élémentaire de partage du modèle; elle est créée dynamiquement et explicitement pendant l'exécution des programmes et la taille des regions est dynamique. De ce point de vue, le modèle Dream ressemble un peu à la structuration fournies par les gestionnaires de mémoire virtuelle.

Dream propose un mécanisme complémentaire de structuration en terme d'objets. C'est à dire que Dream laisse au programmeur la possibilité d'utiliser un espace mémoire partagé pour y stocker ses objets ou variables. Cette possibilité est similaire à celle fournie par les *tuples* du modèle Linda (mais contrairement à Linda, Dream permet d'accèder aux objets dans cet espace). Ce type de fonctionnalité est clairement proposée pour offrir aux programmeurs des facilités de développement.

Le chapitre suivant présentera les principaux choix d'implémentation que nous avons retenus pour Dream. Les nombreux travaux sur les mémoires partagées ont largement mis en évidence les difficultés de réalisation de tels modèles. On peut citer par exemple les difficultés

importantes d'implémentation du modèle Linda tournant autour de la réalisations du *Tuple Space*.

**modèle de cohérence**

L'originalité principale du modèle Dream réside très certainement dans le fait que la cohérence qui y est définie est faible, certains iront même jusqu'à dire qu'il n'y a pas de cohérence du tout. Notre approche a été de fournir un modèle extrêmement rustique et simple, et ensuite de fournir des mécanismes aux programmeurs pour la renforcer. L'approche couramment utilisée est de proposer un modèle de cohérence assez fort et d'en optimiser l'efficacité en relâchant certaines contraintes. Notre approche est complètement opposée à l'approche traditionnelle.

Un des intérêts de notre approche est finalement d'avoir réussi à proposer une implémentation relativement simple à mettre en oeuvre à partir des outils et environnements existant. La simplicité d'implémentation du modèle sera présentée dans le chapitre suivant.

Il reste également à montrer, et c'est ce que nous ferons dans le chapitre applications, que ce modèle de cohérence est bien adapté à certaines classes d'applications comme les applications où il est nécessaire d'avoir une vision générale et globale (état d'activités de sites par exemple).

**Interface de programmation**

L'interface de programmation est également un facteur très important dans la conception des modèles. Tout modèle, aussi beau soit-il, perdra de son intérêt si son interface d'utilisation n'est pas naturelle et accessible à la communauté des programmeurs Dream. Nous avons donc essayé de proposer une interface de programmation simple. Nous illustrerons aussi dans le chapitre applications la simplicité d'utilisation de Dream.

# 4 Implémentation de DREAM

Un premier prototype de Dream à été implémenté. Pour cette implémentation, nous avons privilégié les aspects "génie logiciel" en termes de facilité de mise en oeuvre et de lisibilité, au détriment de la recherche de hautes performances.

L'implémentation d'une DSM nécessite la mise en commun d'information. Nous avons alors utilisé, à chaque fois que cela été possible, la DSM elle-même afin de partager ces informations.

Dans ce chapitre nous ne décrirons pas tous les détails de l'implémentation de Dream, mais seulement ce qui nous semble intéressant. Tout d'abord, nous exposerons le *cadre de développement*. Puis nous décrirons l'architecture générale de Dream qui s'articule autour de la notion de *région*. Ensuite, nous étudierons, du point de vue implémentation, ce qui se cache derrière la notion de propriétaire. Enfin, nous aborderons les solutions retenues pour la *gestion mémoire* de Dream.

## 4.1 Cadre de développement

Lors de l'implémentation de Dream, nous avons été amené à faire différents choix concernant : l'architecture matérielle cible, le modèle de processus communicant, et le langage de programmation.

### 4.1.1 Architecture

L'architecture cible de Dream est un ensemble de stations de travail reliées entre elles par un réseau de communication rapide. Ce type d'architecture est de plus en plus fréquent dans les laboratoires et les entreprises. Le système d'exploitation retenu est le système d'exploitation de réseau (NOS) dans lequel chaque machine possède son propre système d'exploitation pouvant différer des autres machines. L'intégration des fonctionnalités liées à la distribution est certes moins bonne que sur un véritable système d'exploitation distribué (DOS), mais ces fonctionnalités sont présentes. On retrouve au minimum les possibilités de communication et de création à distance de processus. De plus, nous pensons que si Dream fonctionne sous un système NOS, il pourrait fonctionner sous un DOS.

Le choix de l'architecture et du système d'exploitation est aussi lié à la disponibilité du matériel: Dans notre laboratoire, nous avons à disposition un réseau de plus d'une centaine de stations de travail, fonctionnant chacune avec son propre système d'exploitation. Ces machines sont réparties en quatre grandes familles:

- Un groupe de stations Sun fonctionnant avec le système d'exploitation SunOs, qui est une version *BSD* d'Unix.
- Un autre groupe de stations Sun fonctionnant lui avec le système d'exploitation Solaris, dont le noyau Unix est apparenté *system V.*
- Une ferme d'ALPHA fonctionnant sous OSF/1 qui est aussi un noyau Unix apparenté *system V.* Ces Stations sont en plus reliées entre elles par un réseau haut débit *FDDI*

appelé *giga-switch.*

• Des P.C. sous Linux.

Un des objectifs est de développer Dream de façon à ce qu'il puisse être porté sur ces différentes architectures, mais aussi sur d'autre architectures existantes.

### 4.1.2 Choix du modèle de processus communicants

L'implémentation de la mémoire partagée de Dream nécessite des communications inter-processus, et dans une moindre mesure le contrôle de ces processus. Nous avons cherché un modèle de communication qui soit à la fois simple d'utilisation, indépendant de l'architecture, largement diffusé, et fiable.

Nous avons écarté les solutions utilisant les communications de bas niveau comme les sockets ou la couche TLI (Transport Layer Interface). Ces solutions sont difficiles d'utilisation, et restent dépendantes de l'architecture.

Nous avons également écarté les produits de recherche comme $PM^2$ [Namyst95] dont la diffusion était encore restreinte quand nous avons commencé l'implémentation de Dream.

Notre choix s'est alors porté sur MPI et PVM. au moment de ce choix, MPI était plus un standard sur papier qu'un modèle à part entière. Peu de prototype existait, et aucun n'était pleinement satisfaisant. De plus, MPI s'adresse plus à des machines parallèles qu'à des réseaux de machines. Notre choix s'est finalement porté sur PVM.

PVM (Parallel Virtual Machine) est une bibliothèque de fonctions permettant de créer et de contrôler des processus qui communiquent entre eux à l'aide d'échanges de messages. Cette bibliothèque propose une interface stable et fiable. Elle est distribuée gratuitement, et fonctionne sur un grand nombre d'architectures.

Le principal inconvénient de PVM est qu'il n'est pas possible d'avoir plusieurs flots d'exécution différents dans un seul processus (multithreading). Les versions futures devraient cependant offrir cette possibilité.

### 4.1.3 Choix d'un langage orienté objet

Le langage de programmation choisi pour l'implémentation de Dream est C++, inspiré du langage C. Ce langage est un langage orienté objet. Il en fournit les mécanismes de base comme l'héritage multiple ou le contrôle fort des types.

C++ présente l'avantage de fournir une compatibilité vers le langage C. Celui-ci est le langage privilégié de l'interface du système d'exploitation dont toutes les fonctionnalités peuvent alors être utilisées. Ceci est important pour Dream qui utilise intensément les mécanismes de contrôle de la mémoire d'un processus.

Un programmeur connaissant le langage C peut utiliser Dream sans trop de difficultés. En effet, les structures de ces deux langages se ressemblent, à condition d'utiliser la norme ANSI de C. Le programmeur habitué au langage C a besoin d'apprendre comment instancier un objet C++, et comment appeler les méthodes sur cet objet (figure 4.1). L'instanciation d'un objet se fait statiquement en déclarant l'objet, ou dynamiquement en déclarant un pointeur sur l'objet et en allouant l'objet (méthode *new*). L'appel d'une méthode se fait en écrivant le nom de l'objet suivi du nom de la méthode. Les deux noms sont séparés par un point '.' (objet statique) ou un flèche '->' (objet dynamique).

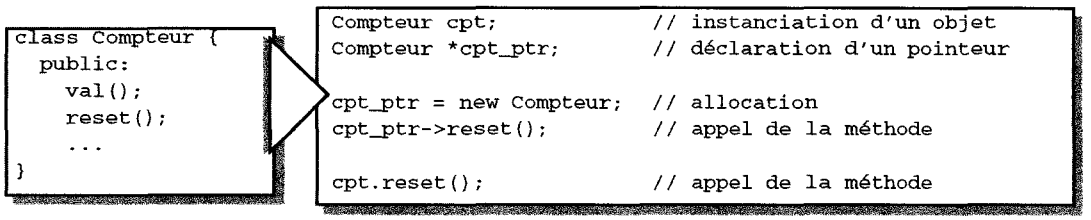



**figure 4.1** Instanciation d'un objet et appel de méthode

En C++, le contrôle des types des objets est fort. Le compilateur vérifie la conformité des opérations intervenant sur un objet, et indique, par une erreur, les accès illicites. Ce contrôle permet d'éviter des erreurs de conception en obligeant le programmeur à une utilisation stricte des objets. Dream ne remet pas en cause ce contrôle: La manipulation d'un objet partagé se fait à l'aide de sa référence qui est elle-même typée.

Les langages à objets permettent de créer des composants logiciels réutilisables. Ces composants proposent en général une interface d'utilisation la plus indépendante possible de l'implémentation réelle (principe d'encapsulation). Ainsi il est possible de modifier cette implémentation sans modifier l'interface, et donc sans modifier le reste de l'application. En C++, un composant prend la forme d'une classe.

Les langages à objet proposent aussi un mécanisme d'héritage permettant de créer des composants héritant de toutes les fonctionnalités d'un autre composant. Ce nouveau composant peut modifier des fonctionnalités déjà existantes, ou en proposer de nouvelles.

Ainsi, il est possible de créer une classe de base prenant en charge le partage, puis de créer de nouvelles classes d'objets partagés en les faisant hériter de la classe de base. Dans ces nouvelles classes le programmeur ne s'occupe que de la manipulation de l'objet, la classe de base prenant en charge son partage. La création de classes d'objets partagés à partir de classes déjà existantes est tout aussi simple: Il suffit de créer une classe héritant et de la classe des objets partagés et de la classe existante (figure).

## 4.2 Architecture générale de Dream

La structure retenue pour l'implémentation de Dream est similaire à la structure du modèle. On retrouve le concept de région, de cache et d'espace virtuel partagé.

Du point de vue implémentation, une région est formée d'un ensemble de régions miroirs réparties entre les processus. A chaque région miroir est associée un descripteur local, contenant toutes les données utiles à la gestion locale, et un descripteur global contenant les données globales de la région (figure 4.2).

Les descripteurs locaux sont associés au cache, qui contient aussi l'ensemble des régions miroirs utilisées localement par un processus. Quant aux descripteurs globaux, ils sont associés à l'espace virtuel partagé, appelé Dsm, qui permet d'accéder à l'ensemble global des régions.

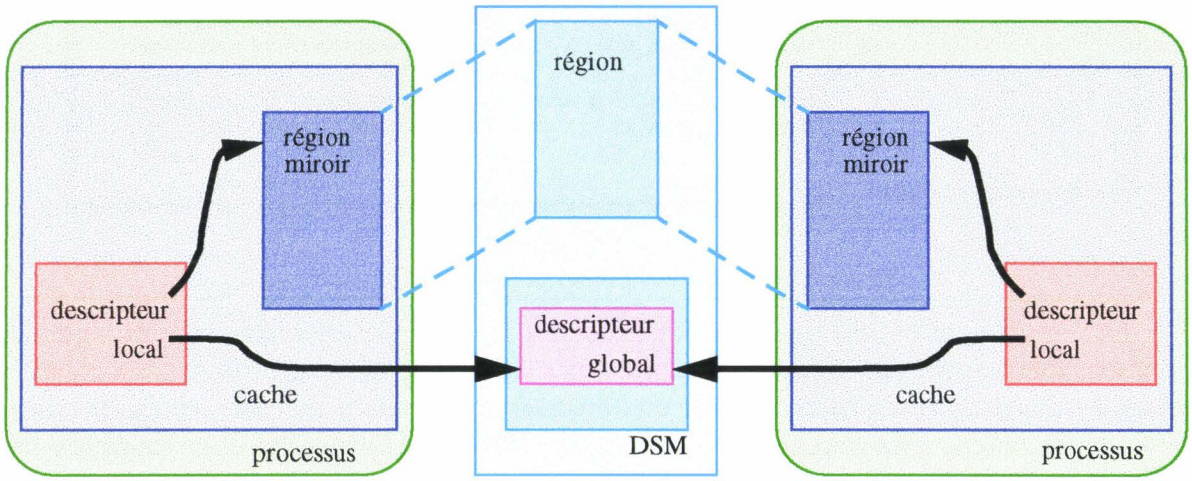



**figure 4.2** Structure générale d'une région

### 4.2.1 Région

Du point de vue du programmeur, une région est représentée par la classe *RgnMirror* (figure 4.3). Cette classe fournit toutes les fonctions permettant au programmeur de manipuler une région. Ces fonctions ont déjà été décrites dans *Interface d'accès à la mémoire partagée*, page 73. Il existe aussi un manuel de ces fonctions (en annexe).

```
class RgnMirror {
  public:
        RgnMirror( );
        RgnMirror( RgnMirrorDesc *rgn );

  int   create( Size size, RgnId ident = 0 );
  int   attach( Addr addr );
  int   attach( RgnId ident );
  int   split( Size offset, RgnId ident=0 );
  int   findAgain( Addr addr );
  int   findAgain( RgnId ident );
  int   detach( );
  void  destroy( );

  int   flush();
  void  freeze();
  void  unfreeze();
  void  waitUpdate();
  ShortTime setUpdateDelay( const ShortTime &delay );
         // ShortTime est équivalent a long.
         // delay est expimé en milli-seconds.
  int   waitWriteAccess();
  int   releaseWriteAccess();

  void* Malloc( Size size );
  void  Free( void* ptr );

  Addr  startAddr();
  RgnId id();
  RgnId size();
  int   isReadWrite();
  int   isReadOnly();
    // etc ...
};
```

**figure 4.3** La classe RgnMirror

Un objet de la classe *RgnMirror* référence le descripteur local de la région, qui quant à lui détient une référence sur la zone de mémoire de la région, et sur le descripteur global (figure 4.4).

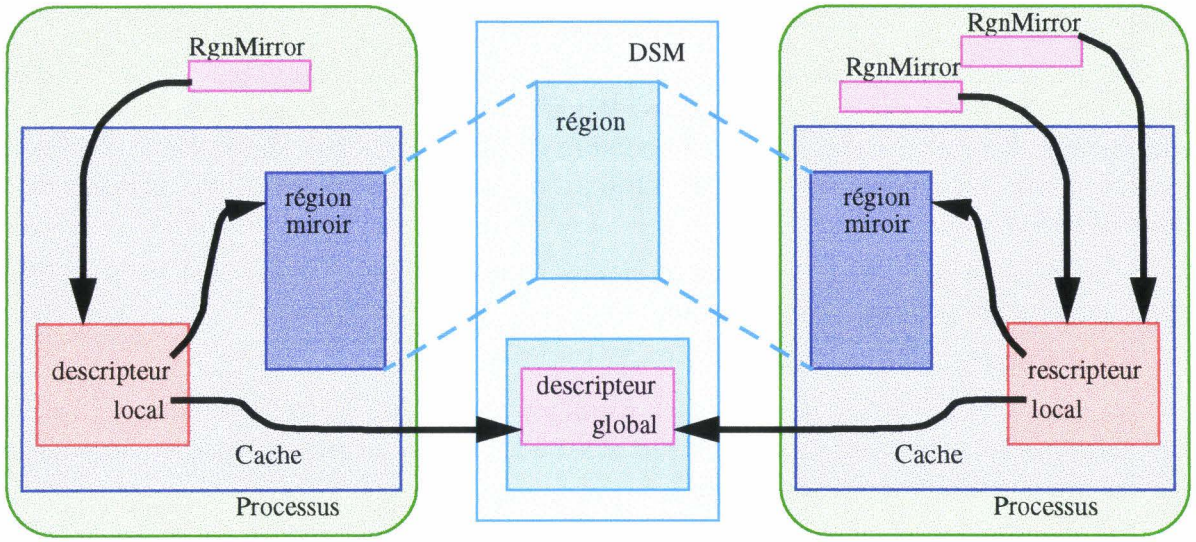



**figure 4.4** Implémentation d'une région

Ces différents objets permettent de séparer, d'une part ce qui est manipulable par le programmeur de ce qui est manipulé par Dream, et d'autre part les données locales des données globales. Le programmeur manipule des objets *RgnMirror* qui référencent des descripteurs locaux manipulés par Dream. Il est ainsi possible de créer dans un seul processus plusieurs objets *RgnMirror* référençant la même région.

Un objet de la classe *RgnMirror*, a une signification uniquement locale au processus, il ne peut pas être partagé ou échangé entre les processus.

### 4.2.2 Cache

Un cache contient les régions miroirs d'un processus. Il est alors naturel que les descripteurs locaux qui leur sont associés soient accessibles par ce même cache.

Ainsi, le cache prend en charge la gestion des descripteurs locaux. Il propose des opérations générales permettant de manipuler l'ensemble de ces descripteurs, comme le parcours des descripteurs, la recherche d'un descripteur, ou encore l'ajout et le retrait d'un descripteur.

Lors de l'attachement d'une région, une simple recherche dans le cache permet de vérifier si la région est déjà attachée, et le cas échéant retourner son descripteur local.

Il faut noter que le cache n'est pas accessible directement au programmeur. Seul Dream manipule le cache lors, par exemple, d'une création ou d'un attachement.

#### Descripteur local

Quand un processus crée ou attache une région, il crée non seulement une copie locale de la mémoire partagé (région miroir), mais aussi un descripteur local de la région. Ce dernier contient les données propres à la gestion locale, comme le droit d'accès, la date de la dernière mise à jour ou les fonctions à appeler lors d'une notification.

Le descripteur local contient également une référence sur le descripteur global de la région qui est contenu dans la liste des descripteurs globaux. Cette indirection sur les données

globales évite une recopie de ces dernières.

L'attachement ou la création d'une région donne au programmeur une référence sur le descripteur local. Chaque attachement supplémentaire de la région donne une nouvelle référence sur le même descripteur local. Une région miroir ne peut alors avoir qu'un seul descripteur local.

Le programmeur ne peut pas accéder directement à ce descripteur local. Il doit obligatoirement passer par la référence (*class RgnMirror*) qui lui est fournie à la création ou à l'attachement. Il peut consulter un certain nombre d'attributs liés à la région (Voir *Attributs*, page 81). Mais il ne peut agir sur ces attributs qu'indirectement par l'intermédiaire des méthodes de la région.

La porté d'un descripteur local n'a de signification que dans le processus l'ayant créé. Il ne peut pas être partagé entre les processus.

### 4.2.3 Espace virtuel partagé

L'espace virtuel partagé de Dream est formé de l'ensemble des régions potentiellement accessibles par les processus. Cet espace sert aussi à la gestion de Dream, notamment pour la mise en commun des descripteurs globaux.

Les descripteurs globaux sont communs à tous les processus. Il sont aussi bien utilisés pour l'attachement d'une région que pour garantir l'unicité d'un nom. Ces descripteurs sont regroupés dans une liste partagée des descripteurs globaux (figure 4.5). Ainsi, ils sont potentiellement accessibles par tous les processus.

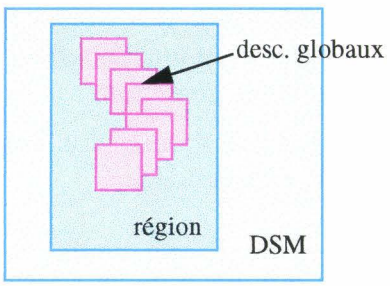



**figure 4.5** Espace virtuel partagé et liste des descripteurs globaux

Un descripteur global est retrouvé soit à l'aide de l'identificateur de la région, soit à l'aide d'une de ses adresses. Un identificateur est unique dans toute l'application. Dream génère ces identificateurs et contrôle l'unicité en utilisant la liste des descripteurs globaux et l'espace virtuel partagé. La génération des adresses est détaillée plus loin ("Gestion mémoire", page 103).

#### Descripteur global

Le descripteur global d'une région contient les données globales de la région, données qui ne varient pas, ou peu durant l'exécution. On trouve notamment le nom de la région (identificateur), l'adresse de la zone de mémoire partagée, sa taille, et le nom de son propriétaire. Parmi ces données, seul le nom du propriétaire peut changer.

Un descripteur global caractérise une région. Il est créé en même temps que la région, et contient toutes les données permettant par la suite de l'attacher. C'est grâce à lui que l'opération d'attachement ne nécessite que le nom de la région : ce dernier permet de retrouver le descripteur global, et toutes les données nécessaires à l'attachement.

**La classe Dsm**

L'espace virtuel partagé de Dream est accessible dans chaque processus par un objet instance de la classe *Dsm*, que nous appelons *interface de la Dsm* (figure 4.6). Cet objet permet à un processus de prendre connaissance de l'ensemble des régions existantes.

Cet ensemble est en fait constituée de la liste de descripteurs globaux, liste qui est accessible par *l'interface de la Dsm.*

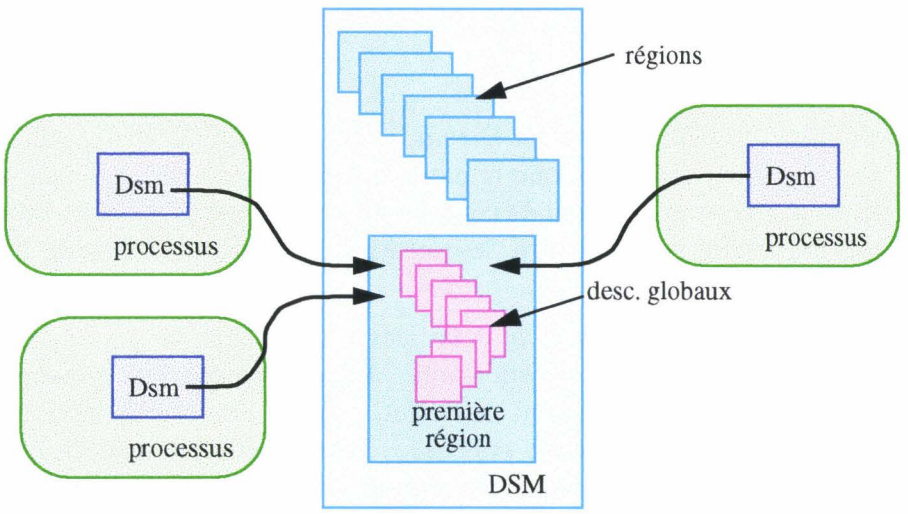



**figure 4.6** Liste partagées et interfaces sur la DSM

Le programmeur n'a pas à se soucier de la création ou de l'utilisation de *l'interface de la Dsm.* En effet, celle-ci est automatiquement créée à l'initialisation d'un processus. Elle est essentiellement sollicitée lors des opérations de création, d'attachement, de destruction et de recherche d'une région. Ces opérations prennent en charge les appels à l'interface de la Dsm.

**Liste partagée des descripteurs globaux**

La liste des descripteurs globaux est partagée à l'aide d'une ou plusieurs régions. La région contenant l'entrée de la liste est toujours la première région attachée par un processus. c'est aussi la première région créée dans l'application. Nous l'appelons communément la première région.

La liste des descripteur globaux est accessible par l'interface de la Dsm. Cette liste est modifiée à chaque création ou destruction d'une région. Le problème qui se pose alors est à qui confier la gestion de cette liste.

Nous proposons deux solutions : une version centralisée dans laquelle un processus à part entière effectue la gestion, et une version distribuée dans laquelle chaque processus effectue ses opérations.

1) Version centralisée

Dans les premières implémentations de Dream, le changement de propriétaire n'existait pas. Nous avons alors décidé de confier la liste à un processus externe à l'application du programmeur. Ce processus est chargé de gérer la liste, il est le seul a pouvoir y effectuer des opérations, comme l'ajout ou le retrait d'un membre. Ce processus est alors propriétaire de la première région contenant la liste.

Les autres processus ne peuvent que consulter la liste. Ceci permet déjà de vérifier l'existence d'une région. Pour modifier la liste, un processus envoi une requête au gestionnaire qui se charge de la modification.

La cohérence de la liste est faible, et les modifications ne sont pas immédiatement répercutées dans tous les processus. Quand l'un d'entre eux ne trouve pas une région dans la liste, il synchronise la première région (qui contient la liste), et recommence sa recherche. Si celle-ci n'aboutit toujours pas, c'est que la région demandée n'existe pas.

Le processus gestionnaire de la liste est lancé automatiquement par le premier processus essayant d'attacher la première région. Le programmeur n'a pas à s'en occuper. Nous avons choisi de placer le gestionnaire dans un processus à part entière, afin qu'il ne perturbe pas l'application du programmeur.

2) Version distribuée

Dans la dernière implémentation de Dream, il est possible de changer le propriétaire d'une région. On voit immédiatement que la première région est une candidate potentielle à des transferts du droit d'écriture: Au lieu de créer un processus spécifique et de lui envoyer des requêtes, chaque processus peut acquérir les droits d'écriture, et effectuer lui-même les modifications dans la première région.

Cette solution présente l'avantage de distribuer l'algorithme de gestion de la liste des régions, et d'éliminer le processus gestionnaire. L'inconvénient est que cette liste est potentiellement un goulot d'étranglement. En effet, plus le nombre de régions créées est important, plus la probabilité augmente pour que deux processus aient besoin d'accéder simultanément à la liste. Dans ce cas, chaque processus demande le droit d'écriture sur la liste, droit qui est accordé d'abord à l'un puis à l'autre.

Le problème vient du fait que la liste est partagée dans une seule région. On peut alors penser la décomposer en plusieurs sous-listes disséminées dans les processus de l'application. Dans ce cas, la première région sert essentiellement à retrouver les régions contenant les sous-listes. En créant une sous-liste par processus le problème rencontré précédemment est pratiquement annihilé: La première région change de propriétaire uniquement lors de la création ou de la destruction de la sous-liste, c'est à dire lors de la création ou de la destruction du processus. Ensuite, quand le processus crée des régions, il ajoute les nouvelles entrées dans sa propre sous-liste.

Le problème des destructions n'est que partiellement résolu. Si la région est détruite par son créateur, il n'y a pas de problème. Si elle est détruite par un autre processus, celui-ci doit quand même demander le droit d'écriture sur la sous-liste contenant les données globales. Quoiqu'il en soit, la probabilité que ce droit soit très prisé est moins grande pour des sous-listes de petite taille que pour une unique grande liste.

3) Solution retenue

L'implémentation actuelle de Dream utilise un processus gestionnaire centralisé. Les implémentations futures utiliseront la version distribuée.

**Génération et contrôle des identificateurs unique**

Dans Dream le nom d'une région est représenté par un identificateur numérique entier.

Pour assurer l'unicité de ce nom, Dream compare systématiquement un nouveau nom aux noms des régions déjà créées. Pour cela, Dream consulte la liste des descripteurs globaux. Cette liste étant partagée entre les processus, le contrôle s'effectue localement. Si le nom n'est pas trouvé, la nouvelle région peut être créée. Si le nom est trouvé, la région existe déjà, et le nom proposé n'est pas admis.

Pour générer des nouveaux noms, chaque processus dispose d'un "sac" contenant des noms à priori inutilisés. Quand le processus a besoin de donner un nom à une région, il se sert

dans le sac, et vérifie comme précédemment que le nom "pioché" n'est effectivement pas utilisé. Si le nom est utilisé, il en pioche un nouveau et recommence jusqu'a trouver un nom inutilisé.

Quand le sac est vide, le processus prend un nouveau sac contenant d'autres noms. En fait, le nom d'une région est un numéro, et un sac un intervalle. Obtenir un nouveau sac signifie obtenir un nouvel intervalle non utilisé. Pour cela, il existe une région contenant la liste des intervalles utilisés. Un processus voulant un intervalle acquiert le droit d'écriture sur la région, et prend un intervalle.

De la même façon, un processus n'ayant plus besoin d'un sac le remet dans la région. Ainsi, la génération des noms est distribuée entre les processus.

## 4.3 Propriétaire d'une région

Le processus propriétaire d'une région a des relations privilégiées avec elle. Par exemple, il est le seul à avoir le droit de la modifier. En contrepartie de ces privilèges, le processus propriétaire est chargé de la gestion de la région.

Comme le propriétaire est le seul à s'occuper de la région, il n'y a pas de conflit d'accès aux données servant à la gestion. Il n'y a pas non plus de problème de synchronisation avec les autres processus lors d'une prise de décision : le propriétaire est le seul à a décider des actions à entreprendre sur la région.

Le contenu de la région miroir du propriétaire contient toujours les plus récentes modifications. En fait, les modifications intervenant dans la région ont lieu directement dans la région miroir du propriétaire. Nous appelons cette région miroir la *région miroir primaire*. Par analogie, nous appelons les régions miroirs détenues par les autres processus *régions miroirs secondaires*.

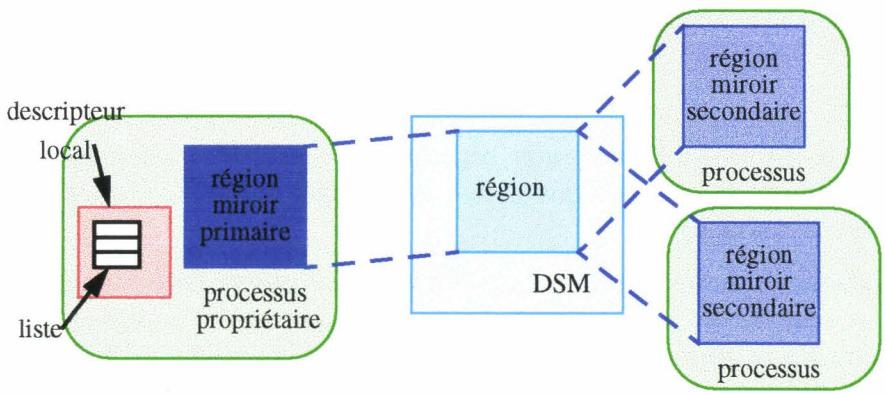





**figure 4.7** Régions miroirs primaire et secondaires

Le processus propriétaire d'une région détient une liste des processus ayant attaché la région. Cette liste fait partie du descripteur local. Elle permet de contacter les processus soit individuellement (message), soit collectivement (*broadcast*).

Le rôle du propriétaire est alors multiple : il est chargé de maintenir la liste des processus, et à partir de celle-ci, de maintenir la cohérence de la région. Il doit aussi répondre aux requêtes émanant des autres processus et concernant la région.

En plus de ceci, le propriétaire peut changer en cours d'exécution. Ce changement nécessite le transfert de données associées, comme la liste des processus, mais aussi un mécanisme permettant de retrouver le propriétaire actuel de la région.

### Propriétaire et cohérence

La synchronisation d'une région par le processus propriétaire entraîne la synchronisation de toutes ses régions miroirs. Le contenu de la région miroir primaire est alors envoyé à chaque processus détenant une région miroir secondaire.

Chaque région miroir mémorise la date de sa dernière synchronisation. Cette date sert au programmeur pour avoir une idée du degré de la cohérence de la région (Voir *Mise à jour automatique*, page 67). Pour le processus propriétaire c'est la dernière date connue à laquelle les régions miroirs étaient synchronisées (En fait, cette date est mise à jour à la première écriture suivant une synchronisation). Pour les autres processus, c'est la date à laquelle la région miroir secondaire a été synchronisée. Ces dates sont des dates locales aux processus. Elles différent de l'un à l'autre car il n'existe pas d'horloge globale.

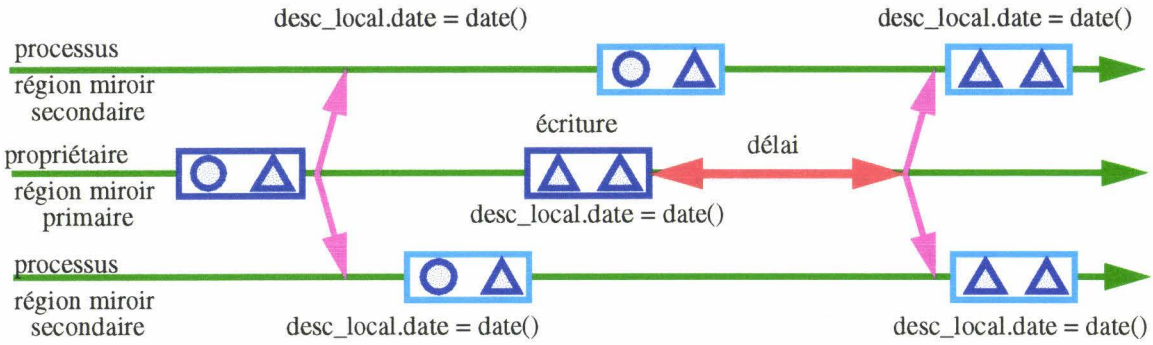



**figure 4.8** Date de synchronisation et cohérence

La date de la dernière synchronisation sert au propriétaire à déterminer si la région doit être mise à jour. Elle est régulièrement comparée à la date actuelle: si le temps écoulé est plus grand que l'intervalle spécifié entre deux mise à jour, le propriétaire effectue la synchronisation de la région.

### Requêtes

Le processus propriétaire doit non seulement envoyer des requêtes (messages) aux processus ayant attaché la région, mais il doit aussi répondre aux requêtes émanant de ces processus et concernant la région.

Il y a d'abord les demandes de synchronisation. Le propriétaire répond en envoyant le contenu de la région miroir primaire au processus demandeur. Si la région est gelée, ce qui interdit les demandes de mise à jour venant de l'extérieur, la requête est mémorisée dans une file d'attente. Le processus y répondra dès le dégèl de la région, ou lors d'une synchronisation explicite par le programmeur dans le processus propriétaire.

Viennent ensuite les demandes d'acquisition du droit d'écriture. Si le propriétaire à abandonné ce droit, il transfert son titre vers le demandeur. Sinon, la requête est mémorisée dans une file d'attente. Elle sera honorée dès que le droit d'écriture est abandonné. L'ordre des réponses est l'ordre de prise en compte des requêtes par le propriétaire. Ainsi, toutes les requêtes sont honorées.

L'attachement d'une région par un processus génère aussi une requête auprès du propriétaire. Celui-ci doit alors ajouter le processus à la liste de ceux ayant attaché la région. Il doit aussi envoyer le contenu de sa région miroir primaire vers le processus afin que ce dernier ait une région miroir secondaire synchronisée. Si la région miroir du propriétaire est gelée, la synchronisation n'a pas lieu, et la requête est mémorisée dans la file d'attente des demandes de

synchronisation. Elle sera honorée de la même façon que ces dernières.

Enfin, quand un processus détache une région, il en informe le propriétaire. Celui-ci enlève alors le processus de la liste de ceux ayant attaché la région. Cette liste reflète ainsi le plus fidèlement possible l'ensemble des processus détenant une vue sur la région.

**Changement de propriétaire**

Le changement de propriétaire se fait par le mécanisme d'acquisition-abandon décrit page 84.

Le titre de propriétaire est lié au droit d'écriture dans une région : un processus acquérant le droit d'écriture devient propriétaire, mais l'abandon du droit d'écriture n'entraîne pas systématiquement la perte du titre. Le titre n'est transféré que si un autre processus a demandé le droit d'écriture. Ceci signifie qu'un processus continu de gérer une région tant que le droit d'écriture n'a pas été acquis par un autre processus.

Quand un processus abandonne son droit d'écriture, aucun message n'est émis. Seul un drapeau est positionné dans le descripteur local de la région miroir primaire, signalant que le droit est libre. Si ce droit est redemandé par le même processus et qu'il n'a pas été acquis par un autre, le drapeau est repositionné dans sa position d'origine, et il n'y a toujours pas émission de messages. Ceci permet au processus propriétaire d'abandonner / acquérir le droit d'écriture aussi souvent qu'il le veut sans générer de messages inutiles (si ce droit n'est pas acquis entre temps).

Un processus peut demander le droit d'écriture alors que celui-ci est encore détenu par le propriétaire. La requête est alors mémorisée par le processus propriétaire avec les autres demandes. Dès que le droit est abandonné, la plus ancienne requête détenue par le propriétaire est honorée. Le processus concerné devient le nouveau propriétaire, et c'est lui qui exécute à nouveau le mécanisme pour les requêtes suivantes en attente. Ainsi, le droit d'écriture passe de processus en processus.

Seule la région miroir du nouveau propriétaire est synchronisé lors du changement de propriétaire. Les autres régions miroir ne le sont pas. Elle le seront lors de la prochaine synchronisation automatique, ou explicite de la part du programmeur.

Pour que la synchronisation automatique fonctionne correctement avec le nouveau propriétaire, il faut transférer la durée de l'intervalle de mise à jour, et le quantum de temps restant avant la prochaine mise à jour. La durée de l'intervalle est nécessaire afin que les changements apportés soit pris en compte par le nouveau propriétaire. Le quantum, quant à lui, assure que la région sera bien mise à jour dans un délai fini, notamment si les changements de propriétaire sont plus rapide que les synchronisations.

Imaginez en effet que, après chaque changement de propriétaire, on attende à nouveau que l'intervalle de mise à jour se soit écoulé pour synchroniser la région. Si un changement intervient avant la fin de l'intervalle, celui-ci est réinitialisé. Si les changement sont très fréquents, la région risque de ne plus être mise à jour.

**Suivi du changement de propriétaire**

Le nom du propriétaire d'une région peut être trouvé à partir du descripteur global contenu dans la liste partagée des descripteurs. Si le propriétaire de la région est constant durant l'application, le nom contenu dans la table est lui aussi constant. Si le propriétaire change, l'information peut ne pas être à jour. Le nom trouvé est alors celui d'un processus anciennement propriétaire de la région. Il existe alors un mécanisme permettant de remonter vers le propriétaire actuel.

Chaque région miroir mémorise le nom de son propriétaire probable. C'est à ce processus que sont envoyées les requêtes concernant la région (figure 4.9). Quand le titre de propriétaire change de processus, l'ancien propriétaire mémorise le nom du nouveau comme étant le propriétaire probable. Quand, par la suite, le processus reçoit des requêtes concernant le propriétaire, il les transferre vers le propriétaire probable.

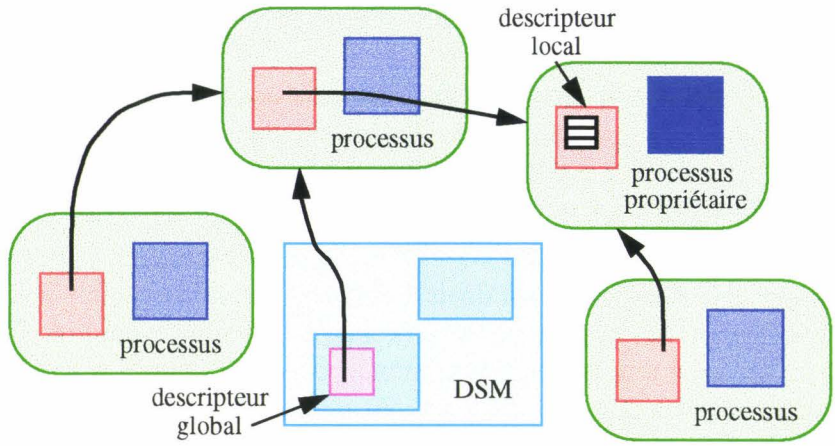



**figure 4.9** Chaîne des propriétaires probables

Si les changements de propriétaire sont fréquents, une chaîne de propriétaires probables se forme, permettant de remonter vers le véritable propriétaire.

Afin d'éviter que cette chaîne ne devienne trop longue, chaque requête du propriétaire vers les régions miroir contient le nom du propriétaire (figure 4.10). Ainsi, la région miroir secondaire peut mettre à jour le nom probable qu'elle détient. L'envoi du nom en même temps qu'une requête normale évite l'envoi d'un message inutile. Ce mécanisme permet de mettre à jour le nom probable du propriétaire lors d'une synchronisation de la région, mais aussi lors d'une réponse à une requête.

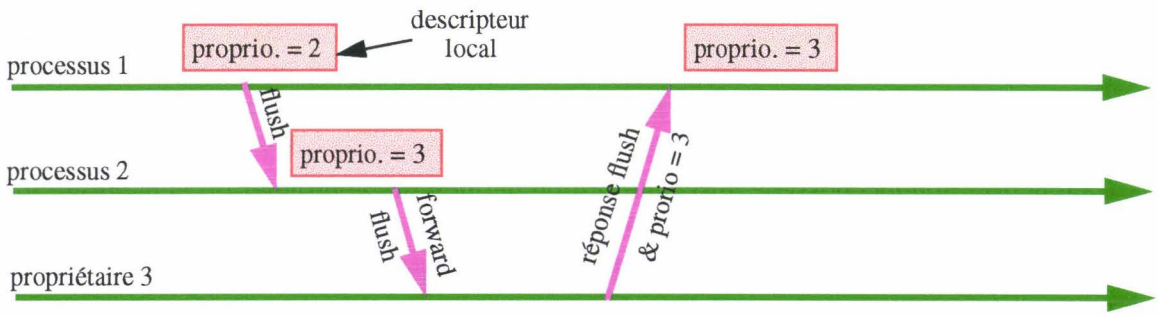



**figure 4.10** Mise à jour du nom du propriétaire probable

Avec ce mécanisme, la chaîne des propriétaires probables reste réduite. Malgré tout, il est possible que cette chaîne soit rompue par un processus détachant sa région miroir. Dans ce cas, le processus demandeur reçoit une erreur soit parce que le processus destinataire n'existe plus (il est détruit), soit parce qu'il ne connaît plus la région. Le processus demandeur est alors averti d'une erreur. Il essaie alors de contacter le processus propriétaire à l'aide du nom contenu dans la liste des descripteurs globaux. Si cette solution échoue, il ne lui reste plus qu'à attendre soit une mise à jour du nom probable, soit une mise à jour du nom contenu dans la liste. Cette attente est courte, car, quand un processus détache une région, il le signale au propriétaire. Celui-ci met alors à jour le nom contenu dans la liste des descripteurs globaux afin

justement de prévenir les ruptures de chaîne.

La chaîne peut être rompue à plusieurs endroits simultanément sans que cela pose de problème. Les processus les plus éloignés ne peuvent pas remonter immédiatement la chaîne, mais le processus le plus proche y arrive. Le propriétaire reçoit bien un message de détachement, et met à jour le nom dans la liste, permettant ainsi aux autres processus de le retrouver.

Quand un processus détache une région, il le signale au propriétaire. Ce message peut ne pas aboutir à cause d'une rupture de chaîne. Si tel est le cas, le processus n'essaie pas de joindre le propriétaire. Celui-ci s'apercevra du détachement lors d'une prochaine mise à jour. En effet, un processus recevant une requête à propos d'une région qu'il n'a pas (ou qui n'est plus) attachée renvoie une erreur. Le propriétaire enlève alors le processus de sa liste de ceux ayant attaché la région.

## 4.4 Gestion mémoire

Dream propose un espace de mémoire partagé entre les processus. Cet espace est composé de régions, chaque région ayant la même adresse dans chaque processus.

Au niveau implémentation, une première difficulté est d'allouer une zone de mémoire dans l'espace d'un processus à une adresse prédéfinie. Une seconde difficulté est de garantir une adresse identique dans tous les processus.

Une fois la région créée, Dream doit être capable de détecter les accès en écriture effectués dans la région, et ceci afin de maintenir la cohérence des régions modifiées. Ce maintien de la cohérence est effectué concurremment à l'exécution de l'application, ce qui ne va pas sans poser aussi quelques problèmes.

### 4.4.1 Allocation de l'espace mémoire d'une région

Dans les langages de programmation C ou C++, l'allocation mémoire se fait en général par les fonctions *malloc()* ou *new()*. Ces fonctions prennent comme paramètre la taille (en octets) de la zone mémoire à allouer, mais ne permettent pas de préciser l'adresse où doit se faire l'allocation.

Dans les systèmes Unix, il existe une autre fonction, moins connue, et permettant d'allouer de la mémoire. C'est la fonction *mmap()*. Le rôle premier de cette fonction est de projeter (*mapper*) un fichier Unix dans l'espace d'adressage d'un processus. Ainsi le contenu du fichier est accessible par des lectures et des écritures en mémoire centrale.

La fonction *mmap()* possède aussi une option permettant de projeter de la mémoire vive dans l'espace d'adressage. Cette mémoire ne dépend d'aucun fichier, et elle est adressable comme de la mémoire obtenue par une allocation conventionnelle.

En général, la fonction *mmap()* prend en charge le choix de l'adresse de projection. Mais il est possible au programmeur de spécifier une adresse. Dans ce cas, c'est au programmeur de gérer les adresses, et de prendre garde à ce qu'il n'y ait pas de collisions avec des adresses déjà utilisées. De plus, les adresses imposées doivent être alignées sur l'adresse d'une page mémoire.

Dans Dream, nous utilisons la fonction *mmap()* pour projeter de la mémoire vive à une adresse imposée. La gestion des adresses est alors assurée par Dream.

Cette fonction *mmap()*, bien que présente sur la plupart des systèmes Unix, est déconseillée lorsqu'il s'agit d'écrire des applications portables. Néanmoins, nous n'avons pas rencontré de grosses difficultés en portant Dream d'un système à l'autre. La partie "non

portable" réside essentiellement dans la façon de désigner l'option "mémoire vive", soit par un fichier spécial (fd <= /dev/zeromem pour Sun), soit par le positionnement d'un drapeau (flags = ANONYMOUS pour Alpha et Linux).

### 4.4.2 Adresse unique

Dans Dream, nous avons choisi de placer les objets aux mêmes adresses dans tous les processus d'une application. Nous avons déjà discuté de ce choix (Voir *Adresse d'une région*, page 77).

La difficulté de cette solution réside dans le choix de cette adresse: Il faut trouver une adresse libre dans l'espace d'adressage de chaque processus, afin que chacun d'entre eux puisse attacher la région. Il faut aussi s'assurer que les processus n'utilisent pas cette adresse pour autre chose que l'attachement de la région à laquelle elle est destinée.

Une solution consiste à laisser le processus créateur d'une région choisir une adresse. Ensuite, il faut demander à chaque autre processus si cette adresse est libre dans son espace d'adressage. Chacun d'entre eux répond, et si une des réponses est négative, il faut recommencer l'opération. Cette solution demande beaucoup d'échange de messages, et risque d'être assez lente.

Dans Dream, nous avons choisi de réserver une partie de l'espace d'adressage des processus pour l'usage exclusif de la mémoire partagée (figure 4.11). Cet espace doit être assez grand pour contenir toutes les régions qui seront créées par les applications. Dans un système proposant un adressage sur 32 bits (environ 4 milliards d'adresses), il est tout à fait possible de réserver quelques millions d'adresse (quelques méga octets d'adresses) pour la mémoire partagée. Dans un système où l'adressage se fait sur 64 bits (plus de 18 milliards de milliard d'adresses), la réservation de plus d'un milliard d'adresses (plus d'un giga octets d'adresses) est envisageable.

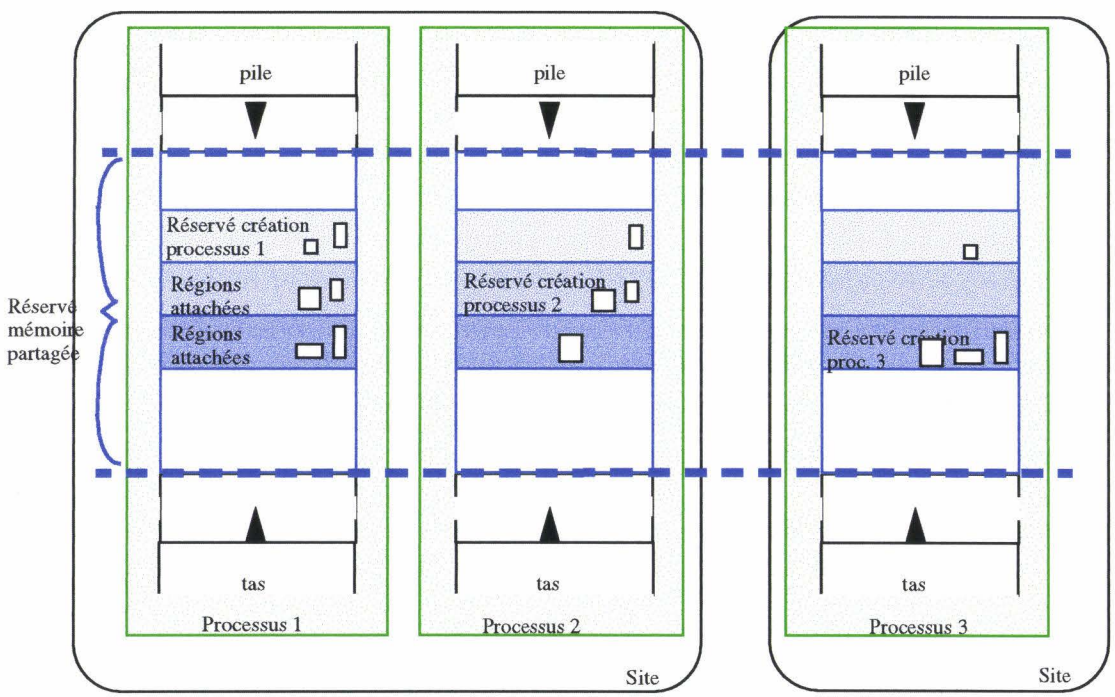



**figure 4.11** Espaces mémoires des processus

Par réservation nous voulons dire que cet espace d'adressage n'est utilisé que pour la

mémoire partagée, et non pas que cet espace est physiquement réservé. Si au cours de l'exécution il s'avère que l'espace initialement réservé est trop petit, il est possible de déployer un algorithme permettant de trouver un autre grand espace commun à réserver.

La réservation d'une partie de l'espace d'adressage assure que les processus n'utilisent pas cette zone pour autre chose que le partage de la mémoire. Il faut maintenant assurer que deux processus n'utilisent pas la même adresse pour deux régions différentes. Nous avons choisi un procédé similaire au précédent: chaque processus se voit attribuer une sous-partie de l'espace réservé, que nous pouvons assimilé à un "paquet" contenant des adresses disponibles. Un processus se sert des adresses de ce paquet pour créer des nouvelles régions. Il ne crée jamais de régions en dehors des adresses qui lui sont allouées, c'est à dire en dehors du sous-espace d'adressage qui lui est réservé.

Ainsi, quand un processus crée une région, il est sûr que l'adresse de cette région est et restera libre dans tous les autres processus de l'application.

Quand un processus a épuisé son paquet d'adresse, il essaie de s'en procurer un autre.

**Gestion de la liste des paquets**

L'attribution des paquets est simple : tout d'abord l'espace d'adressage réservé à la mémoire partagée est découpé en sous-zone qui sont autant de paquets non utilisés. Il existe une liste des paquets non utilisés, et une liste des paquets utilisés. Quand un processus est créé, ou quand il a besoin d'un nouveau paquet, il se sert dans la liste des paquets libres, et l'ajoute dans celle des paquets utilisés.

La question qui se pose est: qui se charge de la gestion des paquets ? Ici aussi il existe une solution centralisée et une solution distribuée. La solution centralisée consiste à confier la gestion à un processus qui traite les requêtes de demande ou de libération des paquets. Comme un paquet permet de créer plusieurs régions, ces requêtes ne sont pas fréquentes. Elles interviennent presque exclusivement à la création ou à la destruction d'un processus. Il est peu probable que le processus gestionnaire devienne un goulot d'étranglement.

Dans la solution distribuée, la liste des paquets est réalisée dans une région partagée. Quand un processus veut un nouveau paquet, il devient propriétaire de la région, et se charge lui-même de la gestion de la liste. Il abandonne le droit d'écriture sur cette liste dès qu'il a fini. Ainsi, la gestion est distribuée entre les différent processus de l'application.

Le premier prototype de Dream utilise la solution avec un gestionnaire de liste centralisé. La raison est qu'il existait déjà un processus dévolu à la gestion de la mémoire. Il était donc assez simple d'étendre les fonctionnalités de ce processus à la gestion des adresses. Les futures versions évolueront vers la solution distribuée.

### 4.4.3 Détection des accès

Dans Dream, seules les régions modifiées sont mise à jour. Les régions qui n'ont pas été modifiées depuis leur dernière mise à jour ne sont pas synchronisées. Elles ne génèrent donc pas de messages inutiles.

Dream doit être capable de détecter les régions qui ont été modifiées. Hors, le programmeur n'indique pas les opérations qu'il effectue. Notamment, il ne prévient pas lorsqu'il modifie la mémoire. Il faut donc trouver un mécanisme permettant à Dream de connaître les modifications intervenues dans une région. Les modifications ne peuvent avoir lieu que dans la région miroir primaire (la copie détenue par le propriétaire), le mécanisme peut alors ne porter que sur cette région miroir.

**Utilisation des signaux Unix**

Le noyau Unix offre peu de solution permettant de détecter les accès mémoire. Un mécanisme accessible est l'utilisation de la protection mémoire et la détection des accès illicites à l'aide de signaux (Segmentation Violation, Bus Error) auxquels sont associés des *handlers*.

Pour l'implémentation de Dream, nous avons choisi une solution simple : nous utilisons le mécanisme de protection pour interdire les accès à la région miroir primaire. Dès que le processus essaie d'écrire dans la région, un signal est déclenché. Ceci provoque le déroutement de l'exécution normale du processus, et l'appel à la fonction associée au handler. Cette fonction est chargée de trouver la région accédée, et d'y positionner un drapeau indiquant qu'elle est modifiée. Ensuite, l'accès en écriture est autorisé pour la région, et l'exécution normale est reprise juste avant l'opération d'écriture ayant déclenché le déroutement. Cette fois-ci l'opération d'écriture aboutit, et la région est modifiée.

Avec Unix, il est possible de protéger individuellement des zones mémoire de la taille d'une page ce qui représente en général 4 ou 8 kilo-octets. Il n'est pas possible de protéger individuellement des zones plus petites. Pour que la solution précédente fonctionne correctement, il faut associer une région à une ou plusieurs pages (suivant sa taille). Si la région n'est pas un multiple de la taille d'une page, il se produit un gaspillage de l'espace d'adressage disponible.

Il existe d'autres solutions, permettant de placer plusieurs régions dans une même page. Par exemple, il est possible, dans le cas ou plusieurs régions sont dans une même page, de détecter précisément quelles sont les régions modifiées. Pour cela, il faut comparer la page à une copie effectuée auparavant (lors du déclenchement du signal).

Cette solution permet une meilleure utilisation de l'espace d'adressage en plaçant plusieurs régions dans une page. L'inconvénient est qu'une partie des accès en écriture à une région est ralentit à cause de la création de la copie.

Le premier prototype de Dream utilise les signaux Unix : il associe une région à une page et n'effectue aucune copie. Cette solution à été retenue pour sa simplicité et ces bonnes performances lors des accès en mémoire, ce qui compense le "gâchis" de mémoire qui lui est associé.

L'utilisation des signaux Unix commence à faire l'objet d'une norme, mais celle-ci n'est pas encore bien respectée. Ceci nous pose des problème pour porter Dream sous Linux, car dans ce système, le manque de documentation nous empêche de connaître l'adresse déclenchant un signal d'erreur d'accès. Par conséquent, il n'est pas possible de marquer les régions qui sont modifiées. L'implémentation sur ce système se fera dés que le comportement des fonctions fautives sera mieux connu.

### 4.4.4 Exécution du maintien de la cohérence

Dans Dream, il n'y a pas de relation entre les accès en mémoire partagée et la gestion de celle-ci. Cette gestion doit s'effectuer en même temps, ou presque, que le programme utilisateur. Elle doit permettre de répondre le plus vite possible aux requêtes venant des autres processus, afin de ne pas les bloquer trop longtemps.

La gestion de la mémoire partagée et l'exécution du programme utilisateur sont deux flots d'exécution distincts se déroulant dans un même processus. Le flot de gestion de Dream doit répondre aux requêtes des autres processus concernant la mémoire partagée. Il doit aussi maintenir périodiquement la cohérence des régions dont le processus est propriétaire.

Dans un système proposant plusieurs thread par processus, la réalisation de ces deux

flots ne pose pas de problème: l'utilisateur utilise un ou plusieurs threads pour son application, tandis que Dream utilise un thread pour répondre aux requêtes, et un thread pour maintenir la cohérence. Ce dernier "dort" le plus souvent, et se réveille à intervalle périodique pour synchroniser les région qui en ont besoin.

Dans un système ne proposant qu'un seul flot d'exécution, le problème est moins évident à résoudre. C'est pourtant sur un tel système que nous avons développé Dream (Voir *Choix du modèle de processus communicants*, page 92). Alors, il nous a fallu mettre en oeuvre des solutions assurant le comportement correct de Dream.

La première est classique : à chaque invocation par le programmeur d'une primitive de Dream, une fonction de gestion est implicitement appelée. Cette fonction répond à toutes les requêtes en attente, et synchronise les régions primaires qui en ont besoin.

Dream fonctionne correctement si le programmeur utilise régulièrement et assez fréquemment ces primitives. Hors, un des objectifs de Dream est de rendre l'utilisation de la mémoire partagée le plus transparent possible à l'utilisateur. Notamment en lui évitant des appels explicites à des primitives spécifiques.

De plus, si le programmeur utilise souvent des fonctions bloquantes provenant des processus communicant, Dream fonctionnera de façon dégradé. Pourtant, un autre objectif de Dream est de pouvoir fonctionner en même temps que les processus communicants. Que faire alors si l'attente de message perturbe le bon fonctionnement de Dream?

Nous avons choisi comme solution de redéfinir certaines fonctions des processus communicants, et notamment les fonctions bloquantes (figure 4.12). Vis à vis du programmeur, ni l'interface de ces fonctions, ni leur comportement ne change. Il les utilise exactement de la même manière que les fonctions qu'elles remplacent. Tout ce que nous faisons, c'est ajouter à ces fonctions un appel à la fonction de gestion de Dream. En plus, pour les fonctions bloquantes, nous testons soit la fin de la condition de blocage, qui correspond en général à l'arrivé d'un message précis, soit l'arrivé d'un message destiné à Dream. Ces derniers sont traités aussitôt, alors que la fin de la condition de blocage marque la fin de la fonction.

```
int dream_recv( int from, int tag )
int dream_trecv( int from, int tag, , TimeVal *tmout)
        // TimeVal == struct timeval
```

**figure 4.12** Fonctions bloquantes de réception de messages

Malheureusement, il existe des cas où le programmeur n'utilise ni les fonctions de Dream, ni les fonctions des processus communicants. Ce sont en général des boucles dans lesquelles le programmeur, soit attend un événement (lecture d'un fichier), soit effectue un calcul.

L'attente bloquante d'un événement venant d'un descripteur de fichier, perturbe le bon déroulement de Dream, Cela peut aussi bien être la lecture dans un fichier que l'attente d'une saisie au clavier. Nous proposons là aussi une fonction simple permettant au programmeur d'attendre l'arrivée d'un événement sur un ou plusieurs descripteurs de fichiers (figure 4.13). Cette attente se fait tout en assurant une gestion régulière de Dream. Dès que l'événement est arrivé, l'attente cesse et le programmeur peut lire l'événement sur le descripteur de fichier.

```
int dream_waitFdInput( int *fds, int nb, TimeVal *tmout=NULL );
int dream_waitFdInput( DreamWaitInputOpt opt, TimeVal *tmout=NULL );
  // avec opt = [DREAM_WAIT_STDIN, DREAM_WAIT_PVM]
```

**figure 4.13** Fonctions d'attentes sur un descripteur de fichier

Il est aussi possible que le programmeur exécute une boucle effectuant uniquement du calcul, et aucune invocation à une primitive permettant de gérer Dream. Dans ce cas, nous conseillons au programmeur d'appeler lui-même explicitement la fonction de gestion de Dream (figure 4.14) à un endroit judicieux de la boucle. La possibilité d'appeler directement la fonction de gestion permet au programmeur de traiter des cas spéciaux que nous n'avons pas prévus.

```
int dream_pendingRequest()
```

**figure 4.14** Fonctions de gestion de Dream

## 4.5 Conclusion

L'implémentation de Dream présente comme particularité d'utiliser, à chaque fois que cela est possible, le concept de mémoire partagée. Ainsi nous avons pu distribuer une partie de la gestion de la mémoire partagée.

Dream n'est pas dédié à une machine précise. Il a été porté avec succès sur trois systèmes différents (SunOs, Solaris et OSF/1). Les deux points d'achoppement sont l'allocation mémoire à une adresse précise (*mmap*), et la récupération des erreurs d'accès à l'aide des signaux Unix. Ce sont les seuls points à considérer pour porter Dream sur d'autre système.

Le prototype actuel peut être amélioré par divers points :

- En évitant la bufférisation des messages de mise à jour. En effet, lors d'une synchronisation, la région est copiée dans un *buffer* avant d'être envoyée sur le réseau. De même, la réception passe par un *buffer*. La suppression de cette bufférisation doit entraîner un gain de temps notoire.
- En réalisant les algorithmes distribués décrits pour la gestion des descripteurs globaux et la gestion des adresses. Ceci afin d'éviter les inconvénients d'une gestion centralisée.
- En ajoutant un serveur de noms afin de pouvoir nommer les régions par des noms symboliques. Ce serveur de noms est réalisé en mémoire partagée, à l'aide d'une ou plusieurs régions. Chaque processus peut ainsi consulter directement le serveur de noms, ou y effectuer des modifications en utilisant le changement de propriétaire.

L'implémentation d'un prototype prouve que le modèle Dream est réalisable. Il nous reste maintenant à prouver que Dream permet de développer des applications fonctionnelles. Ceci fait l'objet du chapitre suivant consacré aux applications.

# 5 Applications

Après avoir présenté le modèle de mémoire partagée DREAM et son implémentation, nous allons maintenant nous intéresser et présenter les applications les plus significatives qui ont été développées.

Dream a été implanté à l'aide du langage C++, et les différentes fonctionnalités du modèle sont donc accessibles au travers de classes et d'objets. Nous montrerons donc que le choix de ce langage permet très facilement et de manière très élégante le développement d'applications.

L'objectif initial de Dream était de fournir un modèle de mémoire partagée facilitant la conception et l'implémentation d'applications que nous appellerons "applications système", c'est à dire des applications fournissant des services d'assez bas niveau (et non présents dans les systèmes d'exploitation) et de trop bas niveau pour figurer dans les applications des utilisateurs. Pour mieux comprendre cette notion d'application "système", citons deux applications qui entrent dans cette catégorie:

- "Régulation de charge"
  Les mécanismes de régulation de charge ne sont pas encore intégrés aux systèmes bien que ces mécanismes soient indispensables à certaines classes d'application pour pouvoir obtenir de bonnes performances. Nous nous sommes ici particulièrement intéressés aux mécanismes d'implantation d'une politique d'information basée sur des indicateurs de charge.
- "Communication collective"
  Dans le chapitre 1 consacré aux systèmes distribués, nous avons constaté que seules des communications point à point étaient disponibles et que des besoins d'outils de communications plus globaux (multicast, groupe) pouvaient se justifier au niveau de nouvelles classes d'applications. Nous présenterons ici une description d'une implémentation distribuée de communication de groupe.

D'autres chercheurs de l'équipe s'intéressent depuis quelques temps à des applications d'optimisation combinatoire, en particulier aux algorithmes de type Branch&Bound [Denneulin96]. Ces applications, très gourmandes en ressources calcul et mémoire, peuvent tirer profit de modèle de mémoire partagée comme celui proposé par Dream. Ces applications, lorsqu'elles sont parallélisées, ont généralement besoin de structures de données globales et partagées. Nous présenterons dans ce chapitre une implémentation du problème du TSP (Voyageur de Commerce) avec le modèle Dream. Les performances et les résultats sont largement évalués et comparés avec d'autres implémentations de mémoire partagée.

L'équipe a également travaillé et acquis une certaine expérience dans les supports d'exécution pour des compilateurs, en particulier pour les langages à objets (Projet PVC). Une question que nous nous sommes alors posé était de savoir si la plate-forme Dream pouvait également constituer un support pour ce type de langage. Nous essaierons de le démontrer

expérimentalement par le portage du langage Orca [Bal94]. Ce qui nous a particulièrement attiré au niveau du langage Orca, c'est que celui-ci proposait la notion d'objet partagé au niveau du langage et il était intéressant de voir si le compilateur de ce langage pouvait facilement exploiter les possibilités fournies par Dream.

La première partie de ce chapitre est consacrée à la présentation de classes d'objets partagés développées avec Dream. Ces classes permettent le développement rapide et simple de régions ou d'objets partagés, qui utilisent une politique de cohérence prédéfinie.

La seconde partie est consacrée aux applications : d'abord les *applications «système»*, puis les *applications «calcul scientifique»* et enfin l'application *«support d'exécution»*.

## 5.1 Des classes d'objets partagés

L'interface de programmation de Dream est essentiellement composée de la classe *RgnMirror*. Cette classe permet un contrôle total de la région, mais les manipulations inhérentes à ce contrôle peuvent paraître difficiles à un nouvel utilisateur, ou répétitives à un utilisateur confirmé. Nous proposons alors des classes prenant en charge ces manipulations.

Nous avons commencé par développer, à partir de la classe *RgnMirror*, des classes permettant de créer des régions dont la cohérence est prédéfinie. Chaque classe offre une interface adaptée à sa cohérence. L'instanciation d'un objet à partir d'une de ces classes crée une région ayant la cohérence choisie. Cette région est ensuite contrôlée et manipulée par l'interface proposée par la classe.

La région créée, il faut encore y projeter un objet quelconque en faisant coïncider son adresse avec celle de la zone de mémoire partagée. Nous proposons alors une classe, *ObjectForRgn*, prenant en charge cette projection/création d'un objet (ou de plusieurs) dans une région déjà créée.

Un objet héritant de la classe *ObjectForRgn* dispose alors de méthodes permettant de l'instancier dans une région existante.

Le plus souvent, le programmeur crée un seul objet par région. Nous proposons une dernière catégorie de classes permettant de créer un objet partagé sans se soucier de la création de la région le contenant. Ces classes facilitent le développement d'objets partagés : l'objet doit simplement être dérivé (hériter) d'une de ces classes.

Toutes les classes sont écrites au dessus de Dream et n'en utilisent que les fonctionnalités standards. Elles redéfinissent simplement les méthodes d'allocation et de libération d'un objet. Le programmeur crée ses objets partagés à l'aide des méthodes de la classe, méthodes qui prennent en charge la projection de l'objet dans la région.

### 5.1.1 Classes de cohérences

La classe *RgnMirror*, propose par défaut une cohérence faible contrôlable à tout moment. Nous avons développé, à partir de cette classe, des classes de cohérences prédéfinies (figure 5.1). Les cohérences proposées correspondent aux trois types d'objets rencontrés lors de la présentation de Dream (page 58 : objet constant, faiblement variable, fortement variable).

Le choix de ces trois cohérences n'est pas exhaustif, car Dream permet de développer bien d'autres modèles de cohérence. Comme exemple nous proposons une classe de régions implémentant la cohérence relâchée rencontrée page 42 (*RcRgn* sur la figure). Le programmeur peut bien sûr développer ses propres classes de cohérence, et les ajouter à celles déjà existantes.

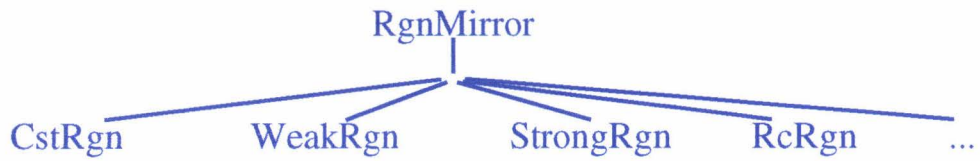



**figure 5.1** Classes de régions partagées

Chaque classe propose une interface correspondante au type de cohérence. Si le programmeur suit les contraintes imposées par la classe, il est assuré d'avoir la cohérence correspondante.

Pour aider le programmeur à choisir le type de cohérence adapté à son objet, nous proposons également une méthodologie basée sur un questionnaire.

**Classe pour objet constant**

Un objet constant est un objet qui est initialisé par un processus, puis consulté par les autres processus.

Une région contenant de tels objets doit donc pouvoir être modifiée par un des processus, qui initialisera le ou les objets. Les processus voulant consulter l'objet doivent attacher la région, et cet attachement doit, de préférence, retourner une région contenant l'objet initialisé.

L'interface de la classe *CstRgn* est donnée figure 5.2. Le principe est de créer une région qui est gelée le temps de son initialisation. Ainsi, un processus attachant la région ne peut pas voir son état incohérent. La région est aussi créée avec un délai de mise à jour infini, interdisant ainsi les mises à jour automatique de la part de Dream.

```
class CstRgn : public RgnMirror {
 public:
  int create( Size size, RgnId ident=0 );
   // delay = ShortTimeInfiny
  int attach( RgnId ident );
  int attach( Addr addr );
  int endInit();
};
```

**figure 5.2** La classe CstRgn

Un processus crée la région, initialise le ou les objets, puis signale la fin de l'initialisation. Les autres processus attachent la région, puis la consultent. L'attachement est ici une opération bloquante : elle n'aboutit qu'après que le processus créateur ait signalé la fin de l'initialisation.

**Classe pour objet faiblement variable**

Un objet faiblement variable est un objet modifié par un processus, et consulté par les autres.

Une région contenant de tels objets est créée par le processus modifiant les objets, puis attachée par les processus lecteurs.

L'interface de la classe *WeakRgn* est donnée figure 5.3. Elle est réduite puisque que constituée uniquement d'une méthode de création et d'une méthode d'attachement; méthodes héritées de la classe *RgnMirror*.

```
class WeakRgn : public RgnMirror {
 public:
  int create( Size size, RgnId ident=0 );
   // delay = WeakConsistencyDelay (1s)
  int attach( RgnId ident );
  int attach( Addr addr );
};
```

**figure 5.3** La classe WeakRgn

### Classe pour objet fortement variable

Un objet fortement variable est un objet modifié et consulté par différent processus.

Un processus ne peut modifier un objet que si il a le droit d'écriture sur la région. Il doit donc acquérir ce droit avant une modification, et le relâcher après afin que d'autres puissent l'acquérir.

La consultation de la région peut se faire n'importe quand, mais sans garantie sur son contenu. Cependant, une consultation effectuée alors que le processus est propriétaire de la région retournera toujours la valeur la plus récente.

L'interface de la classe *StrongRgn* est donnée figure 5.4. La région est créée avec un intervalle de mise à jour de 0,5 seconde, afin d'obtenir une «bonne» cohérence.

```
class StrongRgn : public RgnMirror {
 public:
  int create( Size size, RgnId ident=0 );
   // delay=StrongConsistencyDelay (.5s)
  int attach( RgnId ident );
  int attach( Addr addr );
  int waitWriteAccess();
  int releaseWriteAccess();
};
```

**figure 5.4** Classe StrongRgn

### Questionnaire pour le choix de la cohérence

Choisir un type de cohérence pour un objet (ou région) n'est pas toujours évident. Nous proposons une méthode constituée de questions simples concernant l'objet et son utilisation.

L'objectif est de guider le programmeur dans son choix, afin qu'il utilise la cohérence la plus adapté à son objet.

Les questions et leurs réponses forment un arbre (figure 5.5). Un noeud est constitué d'une question et de réponses possibles, tandis que les feuilles indiquent la cohérence conseillée pour l'objet (plus précisément pour la région contenant l'objet).

Pour trouver cette cohérence, il faut partir de la racine de l'arbre (en haut de la figure), et répondre aux questions. Deux à trois réponses sont proposées, chaque réponse amenant à la question suivante, ou à une feuille contenant la cohérence conseillée.

Voici le questionnaire :

1) Combien de processus modifient l'objet ?

- **un processus** : l'objet est toujours modifié par le même processus, ou les changements de propriétaire sont très rares.

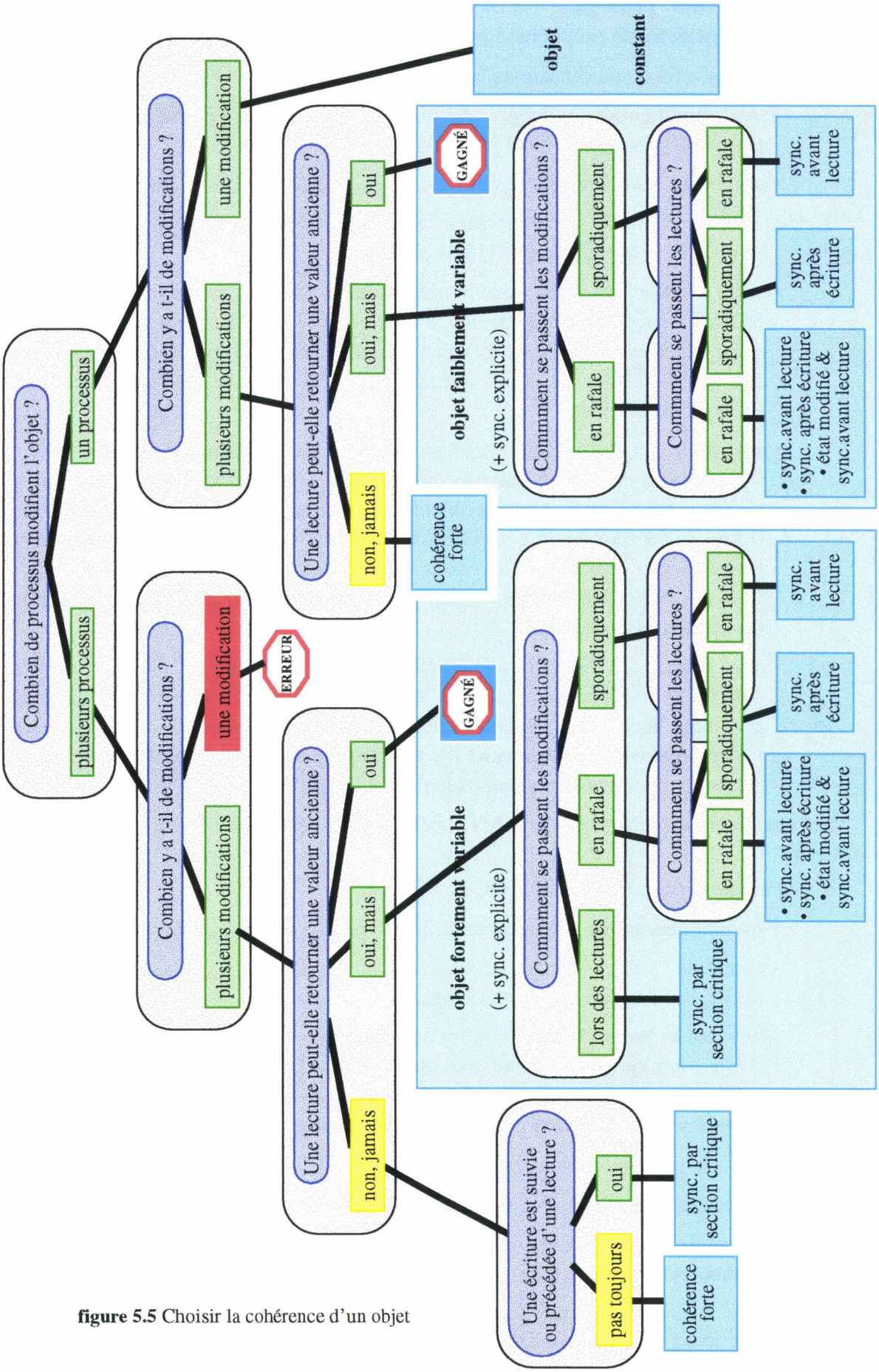



**figure 5.5** Choisir la cohérence d'un objet

- **plusieurs processus** : l'objet est modifié par différents processus. Les changements de propriétaire sont fréquents

2) Combien y a t-il de modifications ?

- **une modification** : l'objet est initialisé puis il est uniquement lu par les processus. C'est donc un objet constant.
- **plusieurs modifications** : l'objet est modifié plusieurs fois durant l'application

3) Une lecture peut-elle retourner une valeur ancienne ? Autrement dit, peut-on appliquer à l'objet la cohérence faible définie par Dream ?

- **oui** : c'est un objet fait pour la cohérence de Dream, vous avez donc gagné ! Suivant les réponses précédentes, c'est soit un objet faiblement variable, soit un objet fortement variable (se référer à l'intitulé du grand bloc). Il faut alors utiliser soit la classe pour les objets fortement variables, soit la classe pour les objets faiblement variables.
- **oui, mais ...** : l'application peut se contenter d'une valeur un peu ancienne de l'objet, mais en certain point la plus récente valeur est nécessaire. Suivant les réponses précédentes, c'est un objet faiblement ou fortement variable pour lequel il faut de temps à autre contrôler explicitement la cohérence. Le programmeur peut effectuer ce contrôle soit explicitement en des points bien précis de son code (il synchronise l'objet quand il a besoin de la plus récente valeur) soit en s'aidant des questions suivantes.
- **non, jamais** : l'objet a besoin d'une cohérence forte. Il faut programmer cette cohérence, ou utiliser l'une des cohérences définies plus loin. Il est aussi possible, dans le cas où une écriture est suivie ou précédée de la lecture de l'objet, d'utiliser le mécanisme d'acquisition-abandon du droit d'écriture afin de former une section critique dans laquelle la cohérence de l'objet est forte.

Les questions suivantes sont destinées à aider le programmeur à contrôler explicitement la cohérence.

4) Comment se passent les modifications ?

- **sporadiquement** : l'objet est modifié une fois de temps à autre.
- **en rafale** : l'objet subit plusieurs modifications dans un court laps de temps, puis plus rien durant un temps plus long.
- **lors des lectures** : la modification de l'objet est suivie ou précédée de sa lecture. Il n'y a pas de consultation sans modification. La cohérence est assurée par le mécanisme d'acquisition-abandon du droit d'écriture, les consultations se faisant dans la section critique formée par ce mécanisme.

5) Comment se passent les lectures ?

- **sporadiquement** : l'objet est consulté une fois de temps à autre.
- **en rafale** : l'objet est lu plusieurs fois dans un court laps de temps, puis plus rien durant un temps plus long.

La réponse à la dernière question amène à une feuille contenant une ou plusieurs propositions pour le contrôle de la cohérence de l'objet. Les propositions sont les suivantes :

- **synchronisation avant lecture** : il faut synchroniser l'objet avant la lecture (ou la rafale de lecture) afin d'obtenir la plus récente modification.
- **synchronisation après écriture** : il faut synchroniser l'objet après l'écriture (ou la rafale d'écriture) pour propager les modifications.
- **état modifié & synchronisation avant lecture** : il faut utiliser le mécanisme de propagation de l'état modifié, et synchroniser l'objet avant la lecture si l'objet est modifié.
- **synchronisation par section critique** : il faut utiliser le mécanisme d'acquisition-abandon du droit d'écriture, et effectuer les lectures et les écritures dans la section critique définie par cet acquisition-abandon.
- **cohérence forte** : L'objet a besoin d'une cohérence forte. Dream a été développé pour des applications pouvant fonctionner avec des objets dont la valeur n'est pas toujours la plus récente. Cet objet ne répond pas à ce critère. Néanmoins, il est possible de définir, et d'utiliser des classes proposant des cohérences "plus fortes" ou adaptées au problème.

Le choix d'une cohérence prédéfinie permet au programmeur de s'affranchir de l'initialisation correspondante de la région. Elle n'interdit pas une reprise en main ultérieure du contrôle de la région. C'est pourquoi toutes les méthodes de contrôle de la classe *RgnMirror* restent accessibles des classes proposant une cohérence prédéfinies.

**Et les autres ...**

Dream permet aussi de développer des cohérences telle que celles rencontrées dans le chapitre 2. Ainsi, nous avons implémenté la classe *RcRgn* correspondant à la cohérence relâchée (Release Consistency).

La classe *RcRgn* (figure 5.6) fournit une interface permettant de manipuler des régions ayant cette cohérence. Les accès à la région se font dans des sections critiques correspondant au type d'accès voulu, lecture ou écriture.

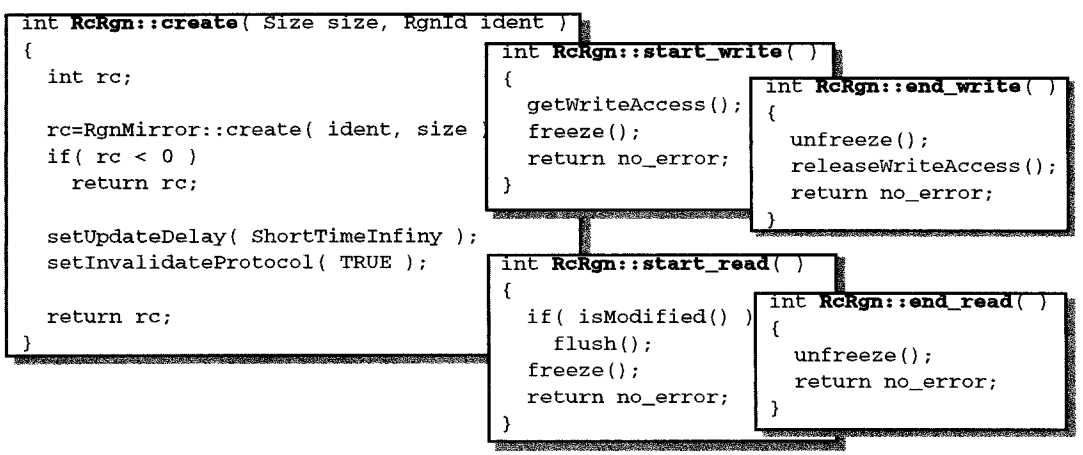



**figure 5.6** Implémentation de la cohérence paresseuse avec Dream

Le contrôle de la cohérence se fait à l'entrée et à la sortie de la section critique. Les mises à jour automatiques sont désactivées, et chaque région miroir est synchronisée si nécessaire (si modifiée) au début d'un accès en lecture. Dans la section critique, la région miroir du processus est gelée afin de prévenir toute modification intempestive. Le gel d'une région

miroir interdit les modifications de cette copie de la région, mais n'interdit pas les accès à la région par les autres processus.

### 5.1.2 Objets pour région partagée

La classe *ObjectForRgn (objets pour région partagée)*, dont l'interface est donnée (figure 5.7), sert à créer ou détruire des objets dans une région qui existe déjà. Si la région est assez grande, il est possible de la considérer comme un *tas*, et d'y créer plusieurs objets. Tous ces objets auront alors les propriétés de la région, notamment en ce qui concerne la cohérence.

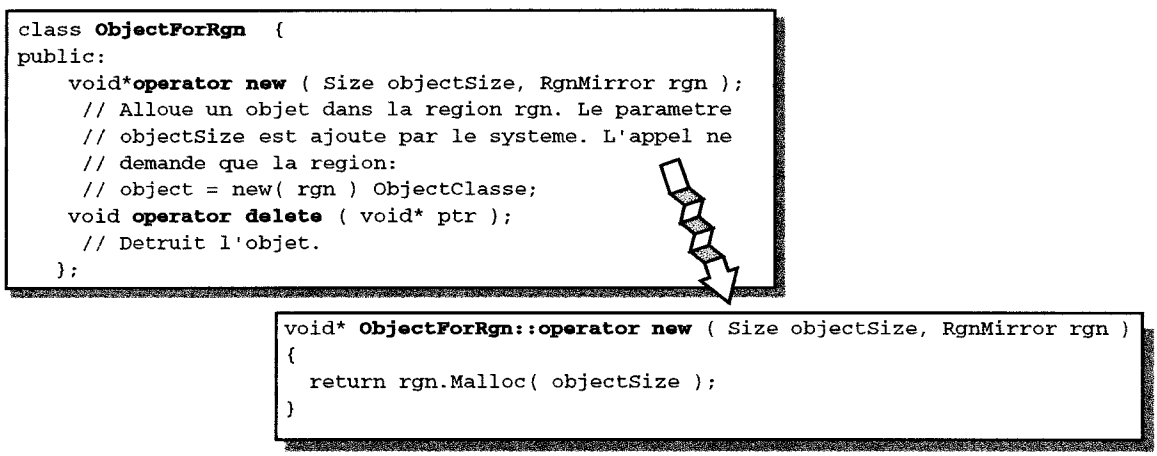



**figure 5.7** Classe des objets pour régions partagées

La classe *ObjectForRgn* fournit uniquement une méthode d'allocation d'objet dans une région, et une méthode de destruction d'objet. Les méthodes de gestion du partage (attachement, synchronisation, cohérence, ...) sont liées à la région contenant les objets. Bien sûr, cette région peut être retrouvée à partir de n'importe laquelle de ses adresses, et donc à partir de n'importe quel objet qu'elle contient.

En C++, les fonctions d'allocation renvoient l'adresse de l'objet alloué. Ensuite, le compilateur C++ se charge d'appeler le constructeur de l'objet, qui lui s'occupe de l'initialisation de l'objet. Nous ne redéfinissons que la fonction d'allocation, libre au programmeur de définir le constructeur.

La classe *ObjectForRgn* permet de créer des structures complexes à l'intérieur d'une même région. Par exemple, il est possible de créer une liste chaînée partagée dans une région, simplement en héritant des classes de gestion de liste développées pour des objets non-partagés, et de la classe *ObjectForRgn*.

#### Un exemple

La figure 5.8 donne le graphe d'héritage ainsi qu'un exemple d'utilisation de la classe *ObjectForRgn*. La région est d'abord créée, ici nous utilisons l'interface générale *RgnMirror*, mais nous aurions pu utiliser une cohérence prédéfinie. Ensuite, la tête de liste est projetée dans la région, ainsi qu'un noeud qui est immédiatement inséré dans la liste.

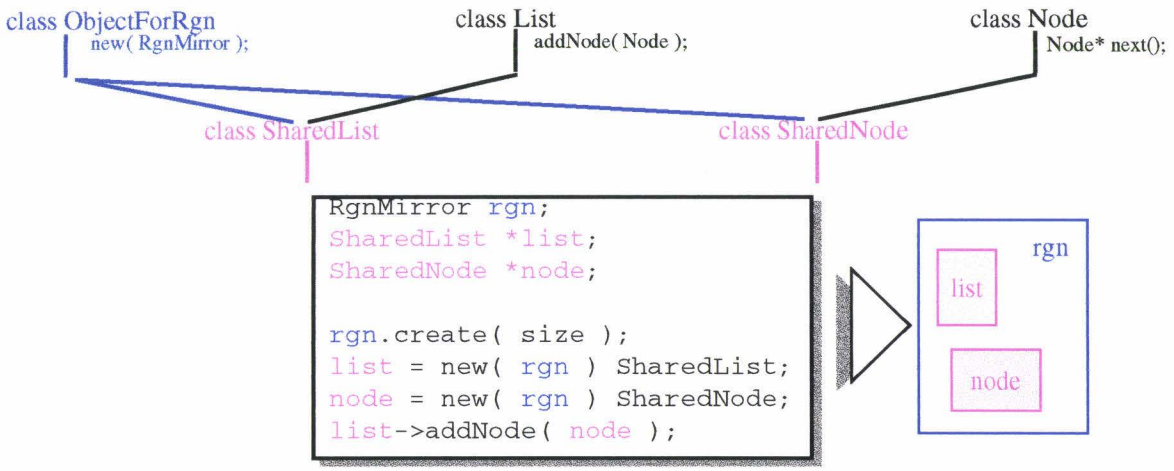



**figure 5.8** Liste partagée dans une région

### 5.1.3 Classes d'objets partagés

Dream propose des classes permettant au programmeur de construire par héritage ses propres classes d'objets partagés. Les classes fournissent toutes les méthodes nécessaires à la création et à la gestion des objets partagés. Le programmeur implémente ses objets partagés en ne se souciant que de l'interface et de l'implémentation qui leurs sont propres. L'interface et l'implémentation concernant le partage sont héritées d'une des classes d'objets partagés.

Ces classes proposent les mêmes cohérences que celles définies pour les régions, avec une interface adaptée à la manipulation de l'objet (figure 5.9).

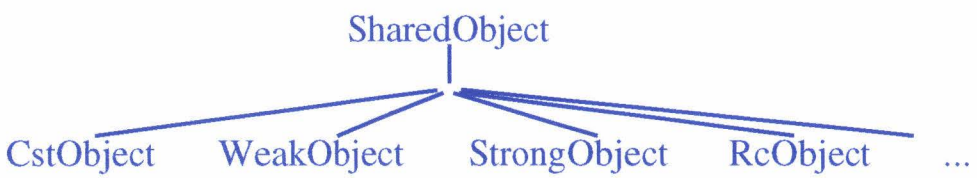
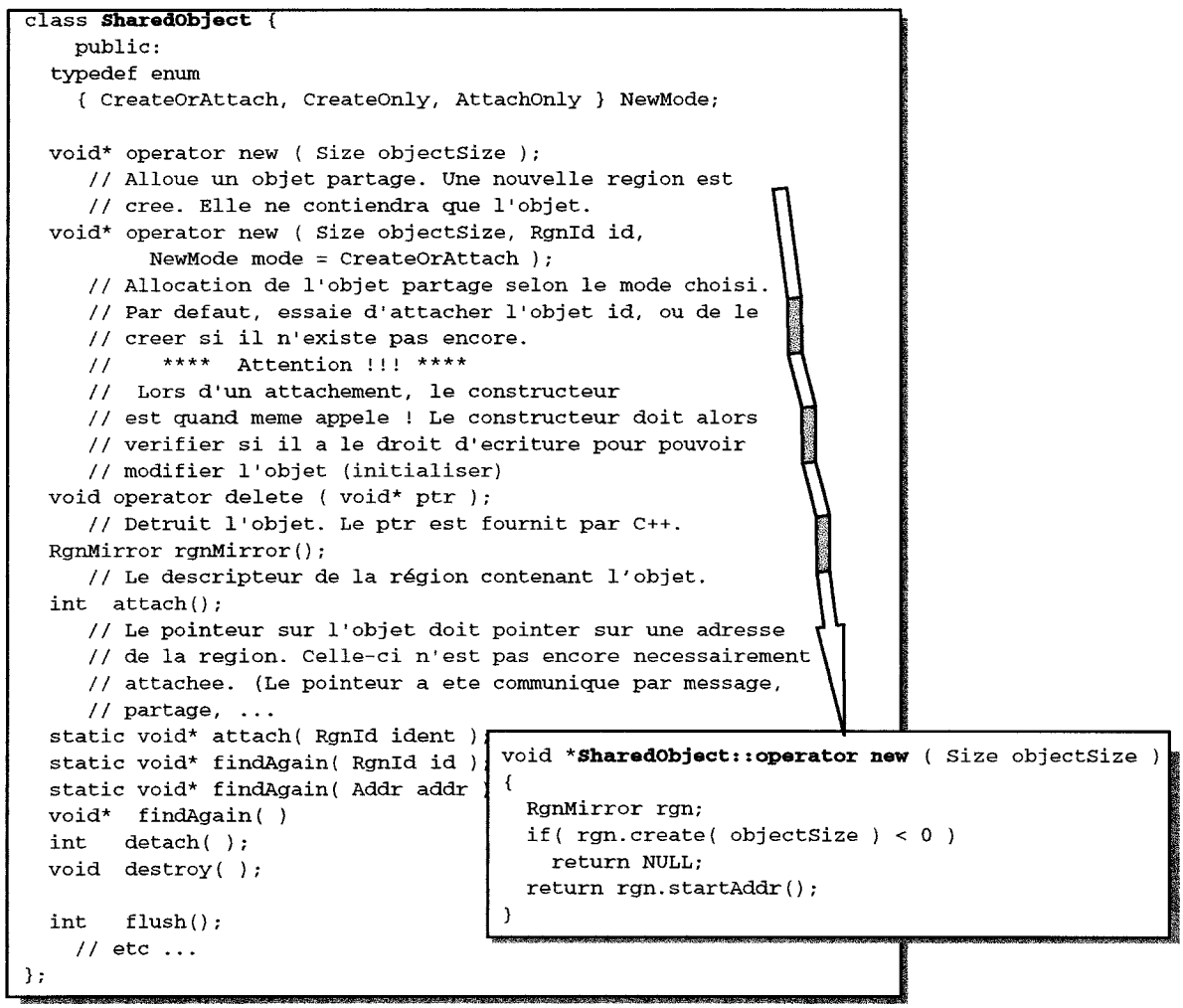



**figure 5.9** Classes d'objets partagés

Les classes d'objets partagés sont dérivées de la classe *sharedObject*, dont l'interface est donnée figure 5.10. Cette classe, permet de créer un objet partagé, sans avoir à se soucier de la création de la région le contenant. Cette création est prise en charge par la classe, ainsi que la projection de l'objet dans la région. L'objet partagé est alors confondu avec sa région, notamment le nom de la région devient le nom de l'objet partagé. La plupart des méthodes associées au partage et s'appliquant à la région s'appliquent aussi à l'objet.

La classe *sharedObject* se spécialise en classes proposant des cohérences prédéfinies : *CstObject*, *WeakObject*, *StrongObject*, *RcObject* (figure 5.11). Ces classes se manipulent comme la classe *SharedObject*, à la manipulation de la cohérence près. Comme pour les classes sur les régions, ces classes évitent au programmeur un travail répétitif : il peut créer directement des objets ayant la cohérence choisie.



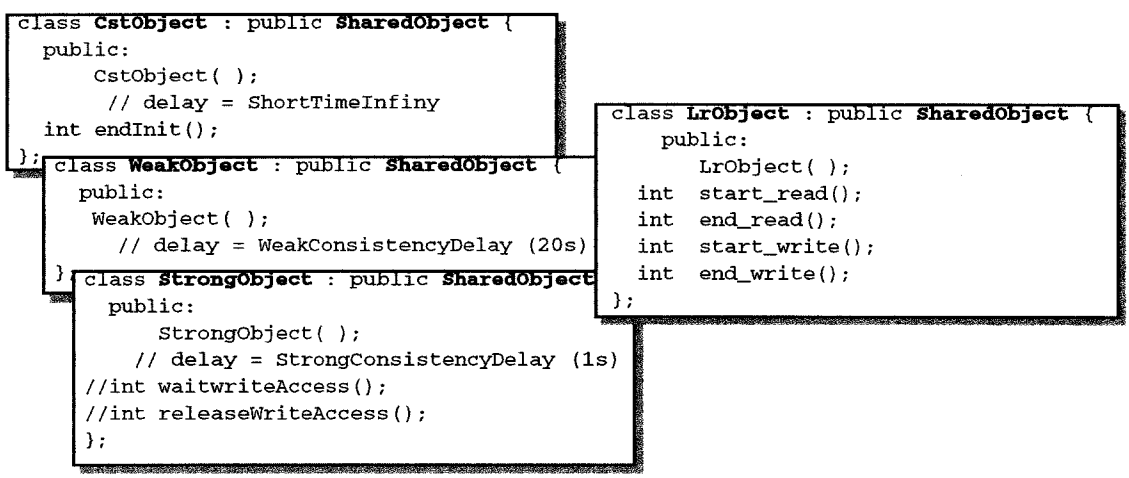
```
class SharedObject {
    public:
  typedef enum
    { CreateOrAttach, CreateOnly, AttachOnly } NewMode;

  void* operator new ( Size objectSize );
     // Alloue un objet partage. Une nouvelle region est
     // cree. Elle ne contiendra que l'objet.
  void* operator new ( Size objectSize, RgnId id,
          NewMode mode = CreateOrAttach );
     // Allocation de l'objet partage selon le mode choisi.
     // Par defaut, essaie d'attacher l'objet id, ou de le
     // creer si il n'existe pas encore.
     //    ****  Attention !!! ****
     //  Lors d'un attachement, le constructeur
     // est quand meme appele ! Le constructeur doit alors
     // verifier si il a le droit d'ecriture pour pouvoir
     // modifier l'objet (initialiser)
  void operator delete ( void* ptr );
     // Detruit l'objet. Le ptr est fournit par C++.
  RgnMirror rgnMirror();
     // Le descripteur de la région contenant l'objet.
  int  attach();
     // Le pointeur sur l'objet doit pointer sur une adresse
     // de la region. Celle-ci n'est pas encore necessairement
     // attachee. (Le pointeur a ete communique par message,
     // partage, ...
  static void* attach( RgnId ident );
  static void* findAgain( RgnId id );
  static void* findAgain( Addr addr );
  void*  findAgain( )
  int    detach( );
  void   destroy( );

  int    flush();
    // etc ...
};
```

```
void *SharedObject::operator new ( Size objectSize )
{
  RgnMirror rgn;
  if( rgn.create( objectSize ) < 0 )
    return NULL;
  return rgn.startAddr();
}
```

**figure 5.10** Classe des objets partagés



```
class CstObject : public SharedObject {
  public:
      CstObject( );
       // delay = ShortTimeInfiny
  int endInit();
};
```

```
class WeakObject : public SharedObject {
  public:
   WeakObject( );
     // delay = WeakConsistencyDelay (20s)
};
```

```
class StrongObject : public SharedObject
  public:
      StrongObject( );
     // delay = StrongConsistencyDelay (1s)
  //int waitwriteAccess();
  //int releaseWriteAccess();
  };
```

```
class LrObject : public SharedObject {
    public:
       LrObject( );
  int  start_read();
  int  end_read();
  int  start_write();
  int  end_write();
};
```

**figure 5.11** Classes d'objets partagés, cohérences prédéfinies

### Création et attachement

La création et l'attachement d'un objet partagé se font à l'aide des méthode d'allocation *new()* ou de la méthode d'attachement *attach()*.

La classe *SharedObject* propose deux méthodes d'allocation d'un objet : une ne demandant aucun paramètre de la part du programmeur (le paramètre *objectSize* est automatiquement ajouté par le compilateur C++), et une nécessitant un nom. La première crée systématiquement l'objet et sa région en lui donnant un nouveau nom, tandis que la seconde crée ou attache l'objet de nom passé en paramètre. Son comportement exact dépend d'un paramètre *mode* qui peut prendre trois valeurs différentes : *CreateOrAttach*, *CreateOnly*, *AttachOnly*. Par défaut, la valeur *CreateOrAttach* est utilisée : la méthode crée l'objet seulement si il n'existe pas déjà, et l'attache dans le cas contraire.

Dans la classe *SharedObject*, il n'existe pas de méthode *create()* comme pour les régions, seule la méthode d'allocation *new()* permet de créer un objet. Ceci est dû au fait que la création nécessite la taille réelle de l'objet, taille qui n'est a priori pas connue de la classe *SharedObject,* et donc de la méthode *create()*. Par contre, cette taille est systématiquement passée en paramètre à la méthode *new()* par le compilateur C++.

L'utilisation de la méthode *new()* pose cependant un problème: tout appel réussi à la méthode entraîne un appel au constructeur de la classe, appel qui est malvenu lorsque l'objet est attaché. En effet, le constructeur sert en général à initialiser l'objet; or, lors d'un attachement, l'objet est déjà initialisé !

Une solution consiste à faire un test dans le constructeur, et à initialiser la région uniquement si elle vient d'être créée (test sur le droit d'écriture). Cette solution est possible si le programmeur a accès au constructeur, ce qui n'est pas le cas quand il hérite d'une classe écrite par ailleurs. La solution dans ce dernier cas consiste à créer l'objet avec l'opérateur *new()* et le mode *CreateOnly*, puis à l'attacher avec la méthode *attach()*.

**Un exemple**

Avec cette classe, la réalisation d'objets partagés est des plus simples comme le montre la figure 5.12. Dans cet exemple, il s'agit de créer un objet partagé contenant les données communes aux différents processus d'une application.

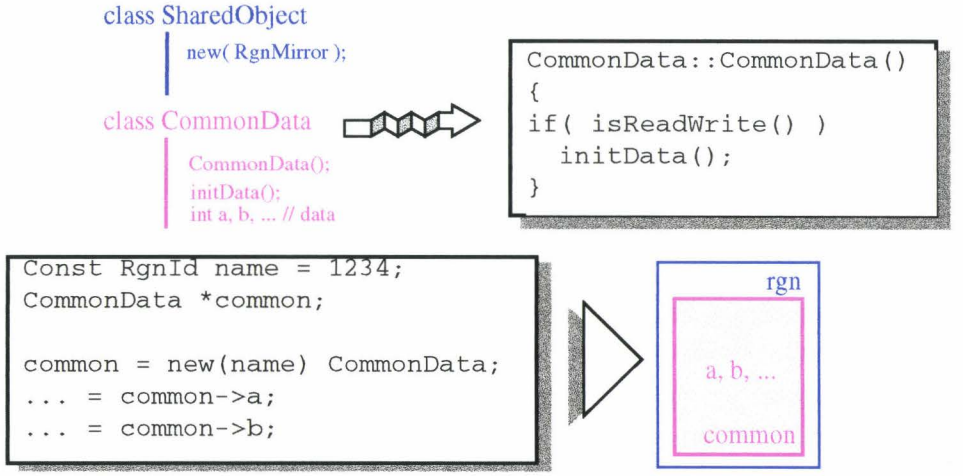



**figure 5.12** Exemple d'objet partagé

Le programmeur écrit de manière standard sa classe d'objet et lui fait hériter de la classe *SharedObject,* ou de l'une des classes avec une cohérence prédéfinie. Il peut notamment hériter de la classe CstObject si par la suite les données communes sont constantes. Dans ce cas, il appelle la méthode *endInit()* à la fin de l'initialisation des données.

Les objets partagés sont créés ou attachés simplement en les allouant dans le programme.

Dans l'exemple, nous avons accès au constructeur de la classe, ce qui nous permet de faire un test afin d'initialiser l'objet uniquement s'il vient d'être créé.

### 5.1.4 Conclusion

Les classes d'objets partagés proposées par Dream facilitent la conception et la réalisation des objets partagés. Le programmeur se concentre uniquement sur la conception propre à l'objet, et non sur son partage. cette dernière phase est prise en charge par les différentes classes proposées.

Les cohérences disponibles actuellement ne sont pas exhaustives : d'autres classes viendront s'ajouter par la suite au fur et à mesure de leur développment.

Les applications développées dans la suite de ce chapitre utilisent les classes d'objets partagés. Ceci nous permet de donner d'autres exemples d'utilisation, mais aussi de démontrer la facilité d'emploi de ces classes pour réaliser des applications distribuées nécessitant un partage de données.

## 5.2 Applications "systèmes"

Nous appelons "applications systèmes" des applications situées dans le système, ou juste au dessus, et dont le programmeur ne doit pas avoir à se soucier de la mise en oeuvre. Parmi ces applications, nous trouvons : la répartition de charge, les groupes, la récolte de trace pour le déverminage ou bien encore le ramasse-miettes.

Afin de conserver un certain niveau de portabilité, nous avons préféré ne pas intervenir dans le système. Nous proposons des solutions pour implémenter ces applications immédiatement au dessus du système. Le programmeur peut alors utiliser ces solutions ou leurs implémentations pour développer d'autre applications.

### 5.2.1 Répartition de charge

L'objectif de cette application est de mettre en oeuvre une politique d'information permettant de connaître la charge de chacun des sites d'un réseau. Ces informations, permettront de déterminer et de localiser un site particulier, comme par exemple le site le moins chargé.

Nous nous limitons à une application dans laquelle chaque site possède un processus "démon" chargé de calculer régulièrement la charge de ce site (*loadd* figure 5.13). Une (ou plusieurs) console lancée sur un (ou plusieurs) site quelconque permet de visualiser la charge des sites, ainsi que le nom du site le moins chargé.

#### Méthodologie

Chaque site possède un indicateur de charge qui est maintenu par le processus démon. Cet indicateur est modifié uniquement par le démon, et est consulté par les consoles.

Une console (ou un processus quelconque) consulte l'ensemble des indicateurs afin de rechercher un site particulier (par exemple le moins chargé). Cette console doit être capable de localiser les indicateurs (les retrouver), et de consulter leurs valeurs.

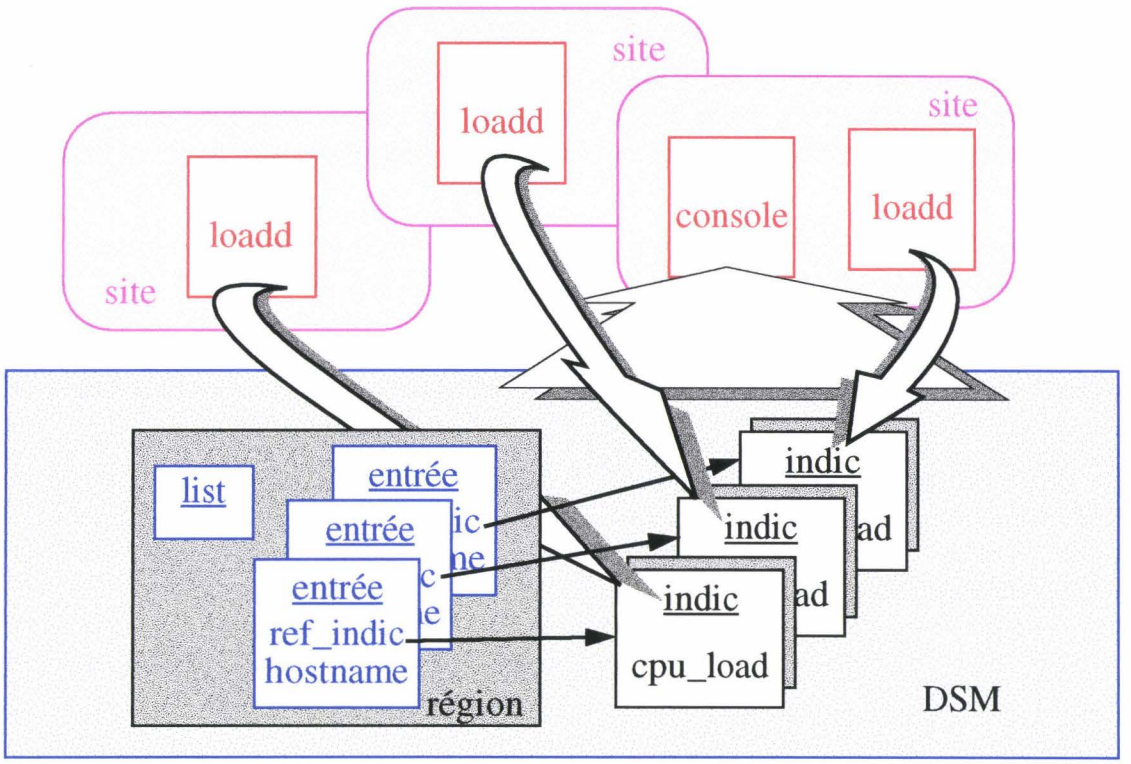



**figure 5.13** Implémentation de la répartition de charge

### Implémentation

Nous avons choisi de partager chaque indicateur à l'aide d'une région différente. Ainsi l'ensemble des indicateurs est potentiellement accessible de n'importe quel processus, et notamment des consoles.

Pour pouvoir retrouver les différents indicateurs, nous créons une liste partagée des indicateurs. Une entrée de la liste référence un indicateur, et contient d'autres informations telles que le nom de son site. Ces informations permettent de savoir si un site est déjà dans la liste, sans avoir à attacher l'indicateur partagé correspondant.

Les objets partagés de l'application sont donc : l'indicateur, *LoadrateIndic* (figure 5.14), et la liste et ses noeuds, *ListIndic* et *NodeIndic*.

```
class LoadRateIndic : public WeakObject  {
  public:
    LoadRateIndic( int tid, char *hostname );
    ~LoadRateIndic();
    void set( RStat &stat );
      // Met a jour l'indicateur local.
    int  operator < ( LoadRateIndic &left );
      // Comparaison entre deux indicateurs de charge
    char* hostName();
      // Le nom du site a qui appartient l'indicateur.
  protected:
    int  cpu_time[RStatCpuCount];
    char hostname[HOSTNAMELEN];
};
```

```
void LoadRateIndic::set( RStat &stat )
{
  int i;

  for( i=0; i<RStatCpuCount; i++ )
    cpu_time[i] = stat.cpuTime(i);
}
```

**figure 5.14** La classe des indicateurs de charge partagés

Pour connaître la cohérence à appliquer à ces objets, répondons aux questions du paragraphe "Questionnaire pour le choix de la cohérence", page 112.

Un indicateur est modifié uniquement par son créateur. Il est consulté par différents processus, et ces processus acceptent une valeur un peu ancienne. Le graphe indique alors un objet faiblement variable. Nous implémentons donc un indicateur partagé en héritant de la classe *WeakObject*.

La liste et ses noeuds sont souvent consultés par différents processus (à chaque parcours des indicateurs), mais sont rarement modifiés (uniquement lors de l'ajout ou du retrait d'un indicateur). Les processus consultant la liste peuvent se contenter de données anciennes, mais nous préférons propager au plus vite les modifications de la liste. Nous avons alors affaire à des objets fortement variables dont nous contrôlons la cohérence après une modification.

Afin de faciliter l'insertion-suppression des noeuds, nous partageons la liste et ses noeuds dans une même région. Les classes *ListIndic* et *NodeIndic* héritent alors de la classe *ObjectForRgn* pour créer différent objets partagés dans une même région. Ces classes héritent aussi de classes implémentant une *liste doublement chaînée (GenericDoubleList et GenericDoubleNode* figure 5.16 et figure 5.15).

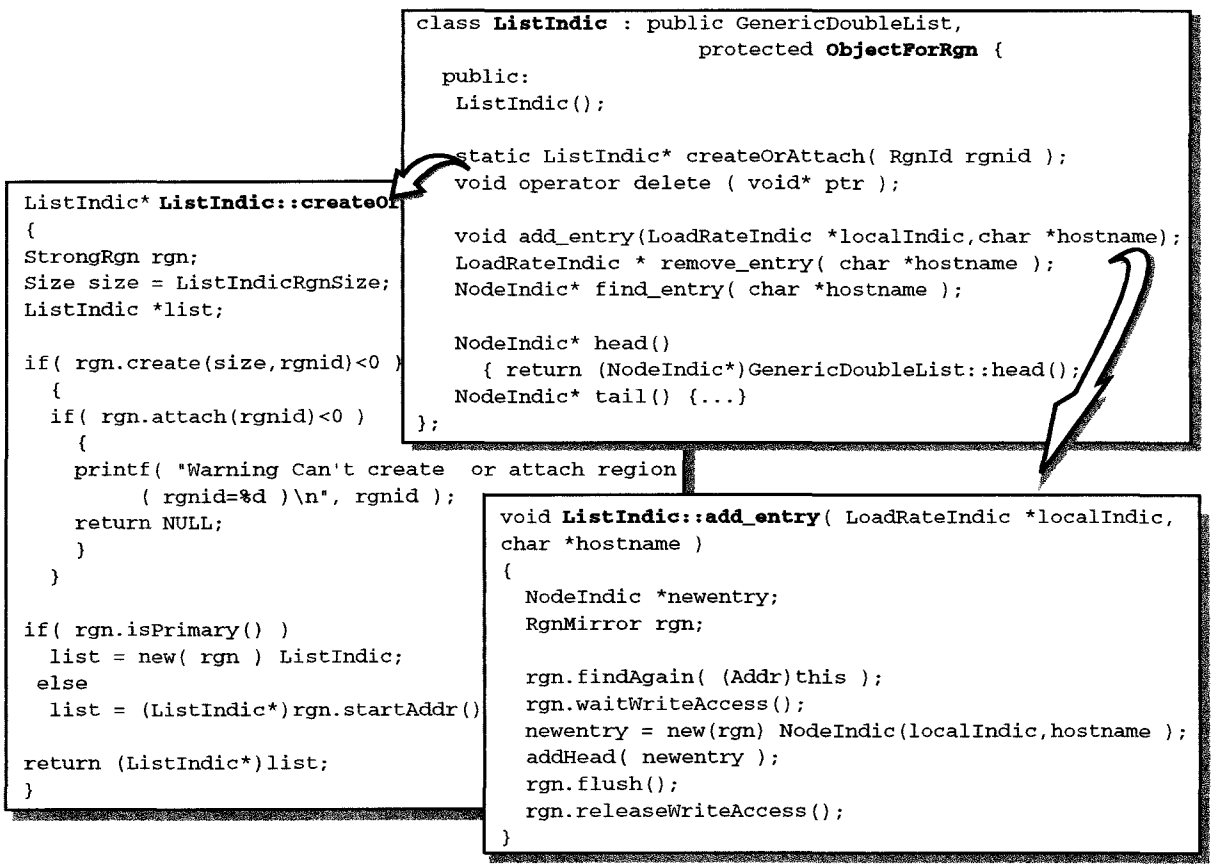



**figure 5.15** classe liste partagée

La région contenant la liste et ses noeuds est créée dans la méthode de liste *CreateOrAttach()*. Cette méthode essaie de créer la région (de type *StrongRgn*) si elle n'existe pas, ou de l'attacher dans le cas contraire. Nous n'avons pas utilisé la méthode *new()* standard, car elle implique l'appel des constructeurs des classes héritées, et notamment dans ce cas ci l'appel du constructeur de la liste doublement chaînée, constructeur qui est difficilement modifiable.

```
class NodeIndic : public GenericDoubleNode, public ObjectForRgn {
  public:
    NodeIndic( LoadRateIndic* indic, char *hostname_ );

    NodeIndic* nextEntry()
      { return (NodeIndic*)GenericDoubleNode::nextNode(); }
    NodeIndic* prevEntry() {...}
    void insertAfter( NodeIndic* node ){...}
    void insertBefore( NodeIndic* node ){...}
    void remove();
    int isLast();
    int isFirst();
      // Inherited from  GenericDoubleNode

    LoadRateIndic* indic();
    char* hostName()
      { return hostname; }
  protected:
    LoadRateIndic* indicator;
    char hostname[HOSTNAMELEN];
};
```

```
LoadRateIndic* NodeIndic::indic( )
{
  indicator->attach();
  return indicator;
}
```

**figure 5.16** La classe noeuds partagé

L'utilisation des indicateurs et de la liste est illustrée figure 5.17. La liste est explicitement créée ou attachée par les processus l'utilisant. Les indicateurs sont créés par les processus démons. Ils sont attachés par les consoles lors de l'appel de la méthode retournant leur référence (*indic()*), car celle-ci est obligatoirement appelée avant leur consultation.

```
  // Creation et attachement
ListIndic *list;
LoadRateIndic *localIndic;

localIndic = new LoadRateIndic;
list = createorAttach( list_name );
list->add_entry( localIndic );
```

```
  // Recherche d'un site
NodeIndic *cur = list->head();
LoadRateIndic *min = cur->indic;

while( cur )
  {
  if( cur->indic() < *min )
    min = cur->indic();
  cur = cur->nextEntry();
  }
```

**figure 5.17** Aperçu de l'implémentation de la console

## Conclusion

Le choix d'une cohérence faible pour les indicateurs de charge donne des valeurs approximatives pour la charge des sites. Il est possible d'avoir une valeur s'approchant de la valeur instantanée en renforçant la cohérence, mais aussi en augmentant les échanges de messages.

La cohérence faible de Dream présente l'avantage, d'une part de diminuer le nombre de messages échangés, et d'autre part de permettre un accès rapide aux données partagées.

Un autre avantage de Dream est d'autoriser l'adaptation de la cohérence en cours d'exécution de l'application. Ainsi une application peut utiliser la cohérence faible, et la renforcer quand le besoin s'en fait sentir.

L'application de répartition de charge que nous proposons donne un aperçu de ce qui est réalisable avec la mémoire partagée de Dream. L'algorithme retenu est des plus simples, d'autres algorithmes plus complexes de répartition de charge peuvent être envisagés.

Ce petit exemple n'avait qu'un objectif pédagogique pour illustrer l'utilisation de Dream et de son guide (questionnaire). Les mesures de performances restent difficiles à mettre en oeuvre et sont dépendantes de l'application mesurée.

### 5.2.2 Groupe

Un groupe est un ensemble de processus pouvant être considérés comme une seule entité. Ainsi, il est possible d'envoyer un message au groupe, message qui sera alors reçu par chacun de ses membres. Il est aussi possible de synchroniser un groupe : tous les processus sont alors synchronisés sur une phase bien définie.

Notre objectif est de réaliser les fonctions de groupe de la figure 5.18. Ces fonctions permettent l'ajout et le retrait dynamique d'un membre, la constitution, l'envoi et la réception de messages, ainsi que la synchronisation par barrières. Ces fonctions ont le même comportement que leurs équivalents PVM (mais elles n'utilisent pas les groupes PVM !).

```
int joingrp( GrpId id );
int leavegrp( GrpId id );

int initsend( );
int msg_pkint( int *n, int stride=1, int num=1);
int msg_pkstr( char *str );

int msg_upkint( int *n, int stride=1, int num=1 );
int msg_upkstr( char *str );

int msg_bcast( GrpId id, int tag );
int msg_recv( int tag  );

int barrier( GrpId id, int count);
```

**figure 5.18** Interface des groupes de communication

#### Méthodologie

Pour réaliser un groupe, nous avons choisi une solution hybride, utilisant à la fois les échanges de messages pour les communications, et la mémoire partagée pour la mise en commun des informations.

A chaque groupe est associé une liste de ses membres. L'envoi d'un message au groupe se réalise par l'envoi d'un message à chaque membre, envoi individuel, ou envoi "multi-cast" si cette fonctionnalité est disponible.

L'ajout ou le retrait d'un membre consiste alors à localiser la liste des membres, puis à y ajouter ou enlever un nom.

La barrière est constituée d'un compteur dénombrant le nombre de processus l'ayant atteinte. Quand un processus atteint la barrière, il incrémente le compteur et se met en attente. Le dernier processus atteignant la barrière débloque les autres et remet le compteur à zéro pour une prochaine synchronisation.

#### Implémentation

Pour chaque groupe, la liste des membres et le compteur de barrière sont partagés en mémoire. Nous les avons regroupé au sein d'un seul objet partagé dont la classe *SharedGroupMngr* est donnée figure 5.19. Cette classe permet l'ajout ou le retrait d'un nom de membre, la consultation des noms des membres et l'attente sur la barrière.

Un objet de cette classe est modifié par différents processus, c'est donc un objet

fortement variable, il hérite alors de la classe *StrongObject*. Afin de propager plus rapidement les modifications de la liste, nous effectuons en plus une synchronisation de l'objet après l'ajout ou le retrait d'un membre.

Pour le compteur de barrière, chaque consultation du compteur est immédiatement suivie ou précédée de sa modification. La cohérence est ici assurée par l'acquisition-abandon du droit d'écriture sur l'objet.

La figure 5.19 donne l'implémentation de la fonction d'entrée d'un membre dans le groupe (fonction appelée par le programmeur), et la fonction d'ajout de ce membre dans la liste (méthode du gestionnaire). On remarque la méthode utilisée pour retrouver le gestionnaire (appel à *new(id)*) et l'acquisition-abandon du droit d'écriture pour modifier la liste.

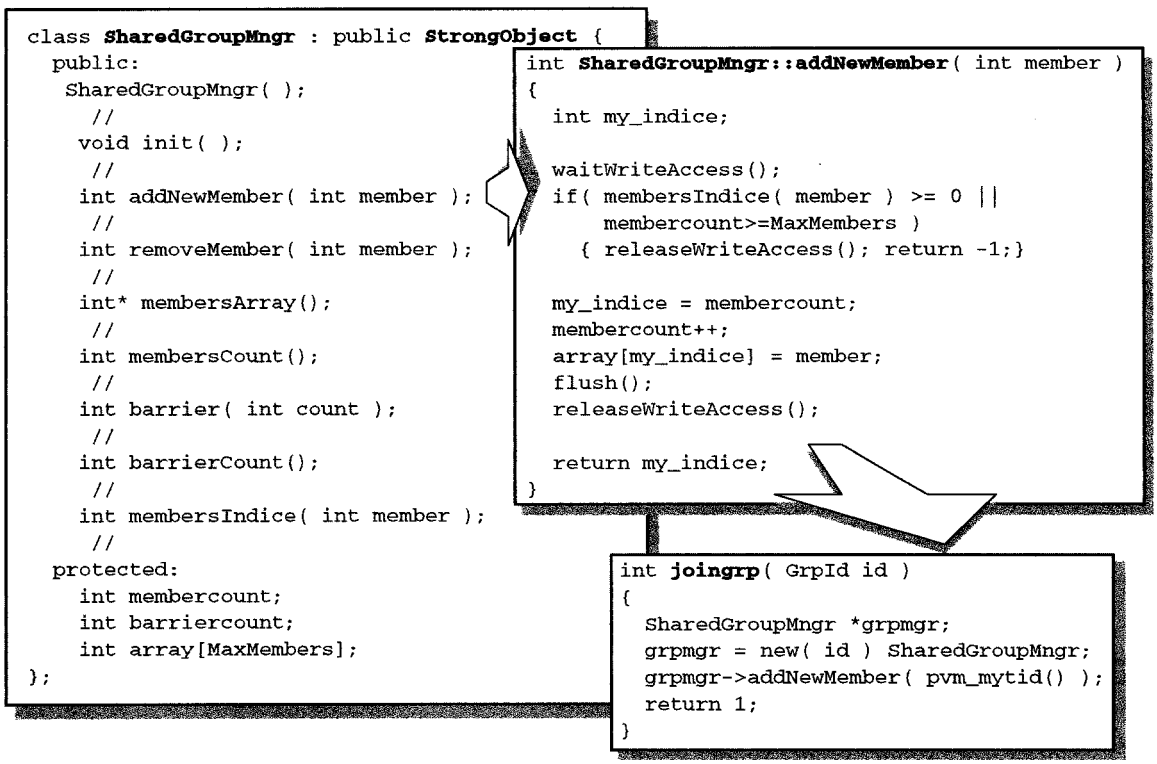



**figure 5.19** Classe d'un objet partagé gestionnaire de groupe

Les autres méthodes et fonctions sont implémentées de façon similaire sauf pour les fonctions de construction et de réception de message qui ne font pas appel à un gestionnaire de groupe. Ces fonctions sont directement implémentées avec la fonction équivalente du modèle de communication.

### Evaluation

Pour évaluer notre implémentation des groupes, nous avons repris un exemple fourni dans la distribution PVM (*gexamp.c*). Nous avons simplement remplacé les appels PVM par leurs équivalents implémentés avec Dream. Cet exemple, dont l'organigramme est donné figure 5.20, crée *n* processus qui se joignent à un groupe, et se synchronisent sur une barrière. Ensuite, chaque processus envoie un message au groupe, et attend les *n-1* messages venant des autres processus. Enfin, les processus se synchronisent une nouvelle fois sur la barrière avant de quitter le groupe.

Nous avons mesuré le temps d'exécution de chacune des primitives d'un groupe, et ce dans la version Dream et la version PVM. Les mesures ont été faites sur une ferme de 15

Alphas.

La figure 5.20 donne les moyennes obtenues pour chacune des primitives, et pour l'exécution totale du processus maître (celui qui lance les autres). Chaque moyenne est calculée à partir des temps obtenus sur les *n* processus de l'application. Les temps minimum et maximum sont ceux trouvés dans cette même application. La figure donne les temps pour dix processus et les temps pour trente processus.

Les temps les plus significatifs sont celui du *broadcast*, ainsi que le temps mis pour envoyer un message au groupe et recevoir les *n-1* "réponses" (*receive messages*). Du fait de la barrière précédente, tous les processus arrivent à peu près en même temps au *broadcast*. Avec Dream, les données sont présentes dans le processus, et le *broadcast* est immédiat. Avec Pvm, le *broadcast* interroge le serveur de groupe qui devient alors un goulot d'étranglement. Ceci se traduit par des temps de dix à vingt fois plus rapide avec Dream.

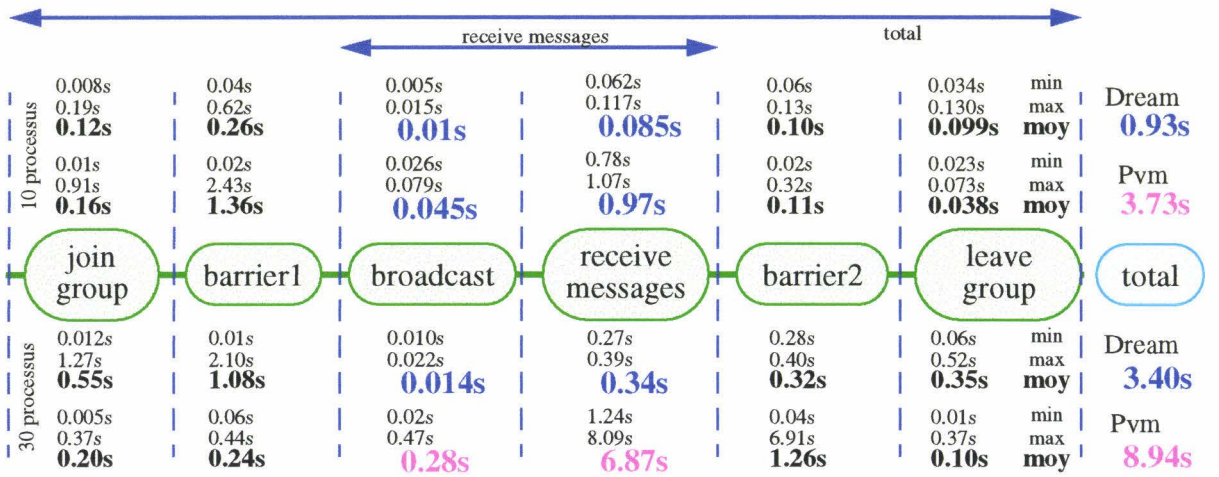



**figure 5.20** Performances

Les temps d'entrée dans le groupe (*join group*) ou des barrières sont moins significatifs. En effet, ces temps sont influencés soit par le délai mis pour lancer le processus (*join group* et *barrier1*) soit par le délai mis pour recevoir les messages (*barrier2*).

Le temps pour quitter un groupe est plus long avec Dream, car chaque processus commence par acquérir le droit d'écriture sur la région puis détache la région, et enfin se termine. Le propriétaire de la région change alors constamment, et le mécanisme permettant de le retrouver est mis à rude épreuve du fait de la destruction des processus. Dans la solution Pvm, un processus se contente d'envoyer un message au gestionnaire de groupe et d'attendre une confirmation.

Globalement, les résultats sont meilleurs avec la solution hybride proposée par Dream. Le temps global d'exécution de l'application est plus rapide.

**Conclusion**

Nous proposons une implémentation simple d'un groupe pouvant servir aussi bien pour la communication que pour la synchronisation.

Notre implémentation est une alternative à celle proposée par PVM. Les avantages en sont :

- Une implémentation simple. Pour s'en assurer, il suffit de comparer la version de Dream (donnée en annexe) avec celle de PVM.

- Une gestion distribuée des groupes, contre une gestion centralisée dans un processus pour PVM.
- Des performances globalement meilleures, essentiellement dues à la répartition des groupes (il n'y a pas de processus formant un goulot d'étranglement), et à la présence des données globales du groupe (liste des membres, compteur) dans chaque processus.

Notre implémentation peut encore être améliorée en lui adjoignant un serveur de nom afin de proposer des noms symboliques au lieu d'identificateurs numériques. Le serveur de nom permettrait la conversion nom symbolique <-> identificateur, et pourrait être réalisé en mémoire partagée.

Cette implémentation des groupes de communication met aussi en évidence l'avantage du modèle de programmation hybride : nous avons tiré parti des avantages de chacun des modèles :

- de la DSM pour partager aisément les données communes,
- des processus communicants pour bufférisér et transmettre les messages.

La cohérence programmable proposée par Dream permet une programmation naturelle de cette application : les données comunes, accesibles dans la DSM, sont explicitement synchronisées chaque fois que la plus récente valeur est requise.

### 5.2.3 Et les autres ...

Nous avons cité à plusieurs reprises d'autres applications "systèmes" telles que la récolte de trace pour le déverminage ou le ramasse-miettes.

La récolte de trace permet de récolter un ensemble de données produites par les différents processus d'une application. Ces données sont ensuite utilisées en vue du déverminage, par exemple pour une réexécution post-mortem [Roos94]. Nous avons étudié une solution dont l'implémentation est similaire à l'application de répartition de charge. Les indicateurs se transforment en buffers recevant les traces d'exécution produites par un processus, et la console devient le processus chargé d'interpréter ces traces. Du point de vue utilisation de la mémoire partagée, cette application ressemble à la répartition de charge. Nous ne la décrirons pas d'avantage.

Le ramasse-miette n'a pas été implémenté, mais nous présentons ici de manière synthétique le principe d'une solution adaptée de l'algorithme proposé par Kafura [Kafura91]. Rappelons d'abord que l'objectif est de ramasser les miettes produites par les processus d'une application. Une miette est une zone de mémoire allouée dynamiquement (ou parfois une entité active telle qu'un thread), et dont plus aucun processus ne possède la référence. L'algorithme de Kafura consiste à construire un graphe des dépendances des entités (zones mémoire ou thread), et à récupérer celles qui ne sont pas accessibles à partir de racines bien définies.

La construction de ce graphe nécessite la connaissance d'informations réparties sur les différents sites. Notre idée est de construire ce graphe en mémoire partagée, chaque site construisant le sous-graphe correspondant aux données qu'il détient. L'algorithme de Kafura est ensuite appliqué par chaque site sur son sous-graphe, en tenant compte des informations contenues dans les autres sous-graphes.

Les informations contenues dans un sous-graphe sont uniquement modifiées par le processus gérant ce sous-graphe. Ces informations sont lues par tous les processus, mais un processus peut très bien s'accommoder d'information "un peu anciennes". Dans ce cas, toutes les miettes ne sont pas toujours récupérées immédiatement, mais elles le seront tôt ou tard.

Ainsi, le ramasse-miettes est un bon candidat à la cohérence faible proposée par Dream.

## 5.3 Applications "calcul scientifique"

Les applications que nous appelons "applications de calcul scientifique" sont des applications dont la principale exigence concerne les ressources processeurs et mémoire. L'exécution de telles applications sur des architectures parallèles et distribuées (plusieurs processeurs, taille de mémoire plus importante) permet de résoudre des problèmes de plus en plus grands.

Dans ce cadre, nous avons développé deux applications représentatives de cette classe d'applications : le produit matriciel et le problème du voyageur de commerce.

Ce qui nous intéresse particulièrement dans ces applications, ce n'est pas l'aspect algorithmique, mais plutôt l'aspect concernant la structuration des données, ainsi que l'évaluation des performances.

### 5.3.1 Produit matriciel

L'objectif de cette application est de répartir un produit matriciel sur un ensemble de sites (processeurs), chacun d'entre eux effectuant une partie du calcul.

Ce calcul met en jeu trois matrices de nombres flottants : deux matrices opérandes et une matrice résultat. Pour faciliter la présentation, la multiplication se fait entre matrices de même taille. Le résultat est stocké dans la matrice résultat, chaque "case" étant calculée avec la formule consacrée : $C_{ij} = \Sigma_K A_{ik} * B_{kj}$.

**Méthodologie**

L'algorithme choisi consiste à calculer en parallèle chaque ligne de la matrice résultat. Comme il y a plus de ligne que de processeur disponible, chaque processeur calcule un bloc constitué de plusieurs lignes consécutives.

Un processus maître crée et initialise les matrices opérandes, puis lance les travailleurs en indiquant pour chacun les lignes qu'il doit calculer. Le maître attend la fin des travailleurs pour éventuellement afficher le résultat, ainsi que le temps d'exécution.

Pour calculer une ligne, un processeur a besoin de la ligne correspondante de la première matrice opérande (A), et de la totalité de la seconde matrice opérande (B). Les résultats sont rangés dans la ligne correspondante de la matrice résultat (C).

**Implémentation**

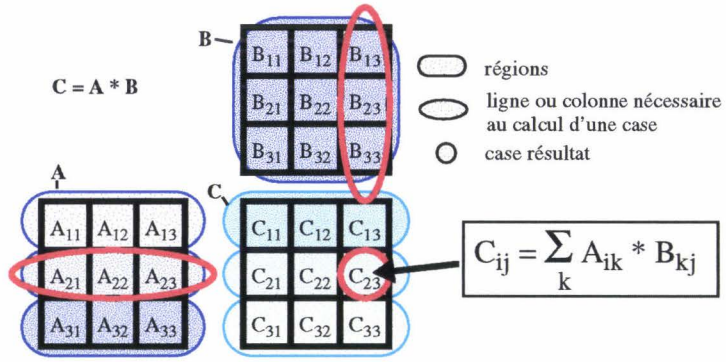



**figure 5.21** Partage des matrices

Nous avons réalisé une implémentation du produit matriciel similaire aux exemples

fournis avec les distributions PVM et Phosphorus. Ceci nous permettra de comparer les performances de Dream avec ces deux autres modèles.

Les objets à partager sont : les lignes de la matrices A, la matrice B et les lignes de la matrice résultat C (figure 5.21).

Les lignes de la matrice A ainsi que la matrice B sont initialisées par le processus maître, puis lues par les travailleurs. Ce sont donc des objets constants. Chaque ligne de la matrices C est modifiée par le processus la calculant, puis est lue par le processus maître. Nous avons aussi choisi de les créer en temps qu'objets constants.

Avec Dream, pour obtenir les lignes des matrices A et C, nous créons dans un premier temps la matrice partagée, puis nous la divisons en lignes qui sont alors autant d'objets partagés à part entière.

La (figure 5.22) donne un aperçu de l'implémentation. Les matrices sont représentées par des tableaux de nombres flottant. Trois régions sont créées avec la taille requise, et chaque zone de mémoire partagée ainsi obtenue est affectée à une matrice. Les matrices opérandes A et B sont initialisées, puis les matrices opérandes A et la matrice résultat C sont divisées.

Ce travail est effectué par le processus maître, qui se charge aussi du lancement des travailleurs.

```
  // Création par le maître
float *A, *B, *C;
CstRgn rgnMatA, rgnMatB, rgnMatResult;

  rgnMatA.create( N*N*sizeof(float), MATRIX_A );
  A = (float*)rgnMatA.startAddr();

  rgnMatB.create( N*N*sizeof(float), MATRIX_B );
  B = (float*)rgnMatB.startAddr();

  rgnMatResult.create( N*N*sizeof(float), MATRIX_C );
  C = (float*)rgnMatResult.startAddr();
  rgnMatResult.releaseWriteAccess();
```

```
  // Attachement par les travailleurs
float *A, *B, *Cpart;
CstRgn rgnMatA, rgnMatB, rgnMatResult;

  rgnMatA.attach( MATRIX_A );
  A = (float*)rgnMatA.startAddr();

  rgnMatB.attach( MATRIX_B );
  B = (float*)rgnMatB.startAddr();

  rgnMatResult.attach( matrix_desc );
  rgnMatResult.waitWriteAccess();
  Cpart = (float*)rgnMatResult.startAddr(

for(i=first_row,j=0; i<last_row; i++,j++
    dotprod( &A[i*N], B, &Cpart[j*N], N)
```

```
void dotprod(float *x, float *y, float *z, int n)
{
  int i, k;
  register int  ky;
  float total;

  for(i = 0; i < n; i++)
    {
    total = 0.0;
    ky = i;
    for(k = 0; k < n; k++)
      {
      total += (x[k] * y[ky]);
      ky += n;
      }
    z[i] = total;
    }
```

**figure 5.22** Implémentation à l'aide de tableaux partagés

Chaque travailleur commence par attacher les régions contenant ses parties de matrices. Il acquiert aussi le droit d'écriture pour les lignes qu'il calcule, et effectue le calcul. Quand un travailleur se termine, ses objets partagés sont automatiquement détachés, ce qui entraîne le transfert des lignes calculées vers le maître.

Nous avons réalisé une seconde implémentation du calcul matriciel. Celle-ci utilise le même principe que la première (trois matrices partagées, calcul par des travailleurs), mais cette fois-ci nous avons utilisé la notion de classe pour réaliser les matrices, et le concept d'héritage pour pouvoir les partager en mémoire.

Nous avons développé une classe *Row (ligne partagée)* et une classe *Matrix (matrice partagée)*. La figure 5.23, donne un aperçu du code correspondant. Chaque classe implémente les fonctionnalités propres aux matrices, les fonctionnalités liées au partage sont obtenues par héritage.

Cette seconde implémentation est plus simple tant au niveau de la réalisation des matrices partagées, que au niveau de leur utilisation (figure 5.24) : création des matrices, attachement, ou application de méthodes.

```
class Matrix : public CstObject {
  public:
        Matrix( );

    Row& operator [](int i)
      {   return mat[i]; }
    void generate( int seed=1024 );
    void generate_identity( int val = 1);
    void generate_indice();

    void check_indice();

    void print( char *str );

    void row_divide( int nb_part );
  protected:
    Row  mat[N];
};
```

```
class Row : public SharedObject {
  public:
        Row( ) {};

    float& operator [](int i)
      {  return row[i]; }
    void compute( Row &row, Matrix &mat );

    void print( char *str );
  protected:
    float row[N];
};
```

```
void Row::compute(  Row &a_row, Matrix &b )
{
  int i,k;
  float total;

  for( i=0; i<N;i++ )
    {
    total = 0.0;
    for( k=0; k<N; k++ )
      total += a_row[k] * b[i][k];
    row[i] = total;
    }
}
```

**figure 5.23** Classes lignes partagées et matrices partagées

```
  // Création par le maître
Matrix *A, *B, *C;

  A = new Matrix;
  B = new Matrix;
  C = new Matrix;

  A->generate_indice();
  B->generate_identity();

  A->row_divide( NSlaves );

  C->releaseWriteAccess();
  C->row_divide( NSlaves );
   ...
```

```
void worker( Matrix &A, Matrix &B, Matrix &C,
             int start_row, int end_row )
{
int row_num;

  A[start_row].attach();
  B.attach();
  C[start_row].attach();

  C[start_row].waitWriteAccess();

  for( row_num = start_row; row_num<end_row ; row_num++ )
    C[row_num].compute( A[row_num], B );
}
```

**figure 5.24** Création et attachement des matrices

**Performances**

La figure 5.25 présente les performances obtenues pour la multiplication de matrices 400 * 400. Quatre courbes sont représentées : une pour chacune des deux implémentations de Dream, et, pour comparaison, une courbe pour l'implémentation avec Phosphorus et une courbe pour l'implémentation avec PVM. Nous nous sommes limités à des matrices de taille 400 * 400.

Pour la première version avec Dream (*dream*), les temps d'exécution sont proches de la solution avec échange de messages (*pvm*). On constate un minimum dans la courbe, puis une dégradation du gain. Cette dégradation vient du fait que le processus maître chargé de répartir les morceaux de matrice entre les esclaves passe plus de temps à échanger des messages qu'un processus travailleur ne met à effectuer son calcul.

La comparaison avec Phosphorus est interessante car Dream et Phosphorus utilisent PVM comme bibliothèque de communication, mais diffèrent par leur modèle de cohérence. On constate que les performances obtenues par Phosphorus (version de novembre 1995) sont nettement en retrait, ce qui est principalement lié au fait que le degré de cohérence de Phosphorus est beaucoup plus fort que celui de Dream.

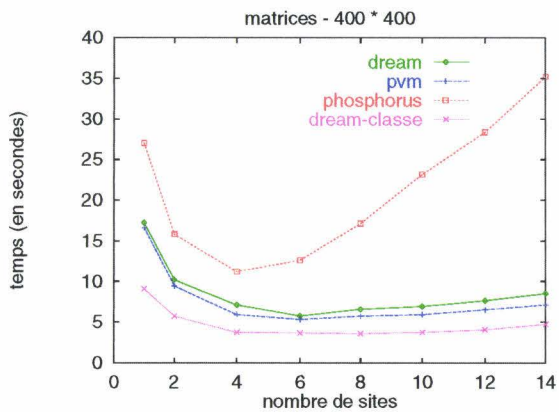



**figure 5.25** Comparaisons des temps d'exécutions pour une matrice 400 * 400

Sur la figure, on constate que les meilleurs temps sont obtenus par l'implémentation Dream avec classe ! Ces temps sont même meilleurs que la solution uniquement à base de messages !

En analysant les temps d'exécution des divers fonctions ou méthodes, nous nous sommes aperçus que le temps de calcul d'une ligne est plus rapide pour les classes (27ms contre 37ms). Cela vient d'une optimisation dans le calcul de l'adresse d'une case, optimisation effectuée automatiquement par le compilateur (celui-ci détecte et optimise la boucle contenant les accès séquentiels aux cases du tableau, cette optimisation n'est pas effectuée dans les autres versions du fait du calcul "à la main" des indices. Comparer la fonction *dotprod()* figure 5.22 et la méthode *Row::compute()* figure 5.23).

### 5.3.2 Problème du voyageur de commerce - TSP

Le problème de voyageur de commerce (Traveling Salesman Problem TSP) consiste à trouver le plus court chemin permettant de parcourir un ensemble de villes à partir d'une ville de départ donnée. Les distances entre les villes sont connues, et chaque ville doit être visitée une et une seule fois.

### Méthodologie

Ce problème peut être résolu à l'aide d'un algorithme de *Branch And Bound* dans lequel un arbre des solutions possibles est construit (figure 5.26). Un noeud de l'arbre décrit une ville, et un chemin de la racine vers une feuille décrit une solution contenant les ville décrites par les noeuds, chaque ville étant parcourue une et une seule fois.

L'algorithme consiste à parcourir l'arbre en profondeur d'abord, et à comparer la distance formée par le chemin partiel dans l'arbre à la plus petite distance déjà trouvée. Si le chemin partiel est plus grand que la plus petite distance déjà trouvée, nul doute que l'ajout d'une ville supplémentaire augmentera encore cette distance partielle. La branche de l'arbre ne contiendra pas de meilleure solution, et son exploration peut être abandonnée.

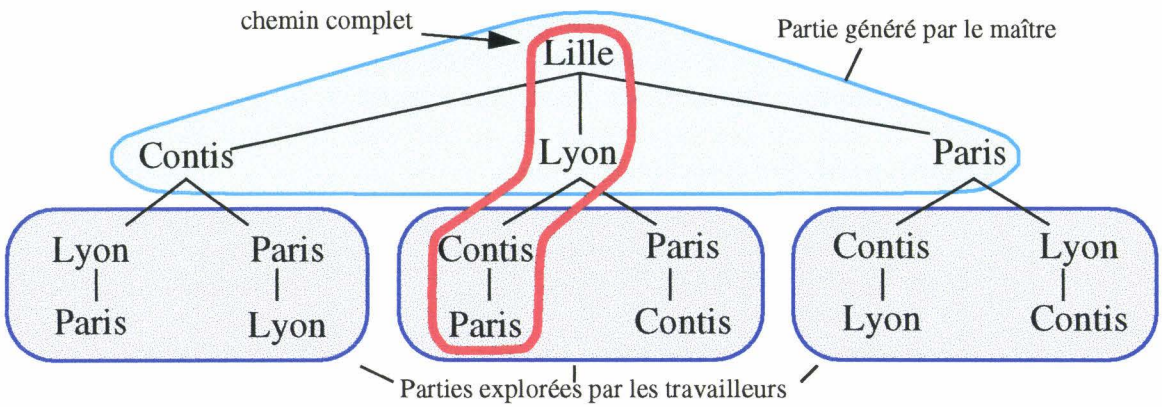



**figure 5.26** Arbre des solutions

Cet algorithme peut facilement être parallélisé : les sous-arbres peuvent être explorés par des processus indépendants. La seule variable dynamique commune à chaque sous-arbre est la distance du plus court chemin. Les exécution montrent que cette distance est rarement modifiée, peut être une dizaine de fois durant la recherche, mais est très souvent consultée, en fait à chaque évaluation d'une nouvelle solution possible.

### Implémentation

Un processus maître est chargé d'initialiser la partie supérieure de l'arbre à partir des distances inter-villes. Puis ce processus lance les travailleurs et attend leurs terminaisons. Les travailleurs explorent les sous-arbres plus en profondeur à la recherche du plus court chemin.

Le travail est réparti entre les travailleurs à l'aide d'une variable de distribution. Cette variable indique le prochain sous-arbre à explorer. Un travailleur n'ayant plus rien à faire prend ce sous-arbre, et modifie la variable de distribution.

Les objets partagés de l'application sont : la distance du plus court chemin, la variable de distribution, les débuts de chemin formant la partie supérieure de l'arbre et les distances inter-villes.

La distance du plus court chemin est consultée et modifiée par différents processus. Un processus consultant la distance peut s'accommoder d'une valeur un peu ancienne. La distance du plus court chemin est donc un objet fortement variable. Mais, une propagation rapide de la nouvelle valeur réduit le nombre de recherches (en coupant les branches plus longues). Nous propageons donc immédiatement la nouvelle valeur (appel à *flush()*).

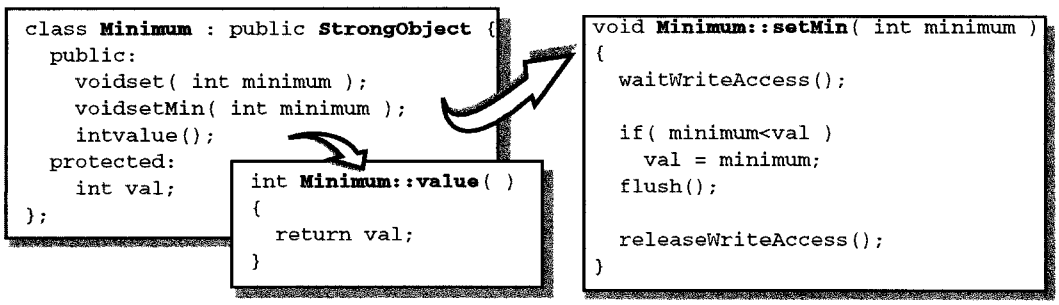
La variable de distribution est lue et écrite par les processus travailleurs. La consultation de la variable est toujours suivie de sa modification. Sa cohérence est assurée par une section

critique formée à partir du mécanisme d'acquisition-abandon du droit d'écriture.

La partie supérieure de l'arbre, constituée des débuts de chemins, ainsi que les distances entre les villes sont générées par le processus maître, puis lues par les processus travailleurs. Ce sont donc des objets constants.

Nous avons écrit les classes correspondantes aux objets partagés.

Nous trouvons d'abord la classe *Minimum* (figure 5.27) implémentant la distance du plus court chemin. La consultation de la valeur se fait sans aucune synchronisation. Le positionnement d'une nouvelle valeur se fait en ayant le droit d'écriture, ceci assure d'avoir la plus récente version de l'objet. Avant le positionnement, il faut vérifier que la nouvelle valeur est bien inférieure à celle contenue dans l'objet.



```
class Minimum : public StrongObject {
  public:
    voidset( int minimum );
    voidsetMin( int minimum );
    intvalue();
  protected:
    int val;
};
```

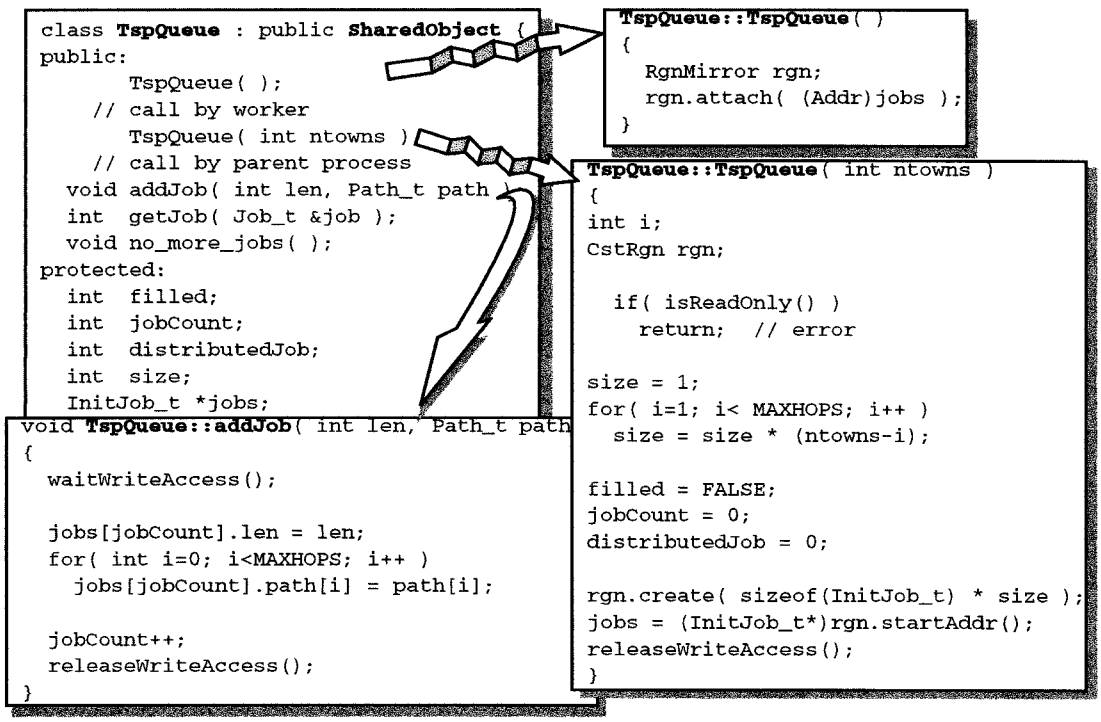
```
int Minimum::value( )
{
  return val;
}
```

```
void Minimum::setMin( int minimum )
{
  waitWriteAccess();

  if( minimum<val )
    val = minimum;
  flush();

  releaseWriteAccess();
}
```

**figure 5.27** Classe Minimum



```
class TspQueue : public SharedObject {
public:
      TspQueue( );
    // call by worker
      TspQueue( int ntowns )
    // call by parent process
  void addJob( int len, Path_t path );
  int  getJob( Job_t &job );
  void no_more_jobs( );
protected:
  int  filled;
  int  jobCount;
  int  distributedJob;
  int  size;
  InitJob_t *jobs;
```

```
void TspQueue::addJob( int len, Path_t path
{
  waitWriteAccess();

  jobs[jobCount].len = len;
  for( int i=0; i<MAXHOPS; i++ )
    jobs[jobCount].path[i] = path[i];

  jobCount++;
  releaseWriteAccess();
}
```

```
TspQueue::TspQueue( )
{
  RgnMirror rgn;
  rgn.attach( (Addr)jobs );
}
```

```
TspQueue::TspQueue( int ntowns )
{
int i;
CstRgn rgn;

  if( isReadOnly() )
    return;  // error

size = 1;
for( i=1; i< MAXHOPS; i++ )
  size = size * (ntowns-i);

filled = FALSE;
jobCount = 0;
distributedJob = 0;

rgn.create( sizeof(InitJob_t) * size );
jobs = (InitJob_t*)rgn.startAddr();
releaseWriteAccess();
}
```

**figure 5.28** Classe TspQueue

Une autre classe partagée est la classe *TspQueue* regroupant les débuts de chemins et les variables nécessaires à la distribution de ces chemins (figure 5.28). On trouve notamment un compteur indiquant le nombre de débuts de chemin présents dans la queue, et le nombre de chemins déjà explorés. C'est ce dernier compteur que les travailleurs modifient par

l'intermédiaire de la fonction *GetJob()*.

Les débuts de chemin sont rangés sous la forme d'un tableau partagé dans une région, tableau accessible par *jobs[]*. Les variables concernant la distribution sont rangées dans une autre région, créée lors de la création d'un objet *TspQueue*.

Une dernière classe, *DistTab,* permet de créer une matrice partagée contenant les distances entres les villes. Ces distances sont rangées par ordre croissant. Un objet de cette classe est partagé, l'accès aux données contenues dans la matrice se fait en utilisant la notation consacrée aux matrices (distance[i][j]).

La figure 5.29 donne le code d'un travailleur recherchant le plus court chemin. Après attachement des diverses régions, le travailleur demande et évalue les chemins tant qu'il en reste.

```
void tsp (int hops, int len, Path_t path, Minimum *minimum, DistTab &distance)
{
int i, city, me, dist ;

if (len >= minimum->value()
  return ;

if (hops == distance.NrTowns
  {
  if (len >= minimum->value(
    return;
  minimum->setMin( len );
  }
 else
  {
  me = path [hops-1] ;
  for (i=0; i < distance.NrTowns; i++)
   {
    city = distance.dst[me][i].ToCity
    if (!present (city, hops, path) )
     {
     path [hops] = city ;
     dist = distance.dst[me][i].dist ;
     tsp (hops+1, len+dist, path, minimum, distance) ;
     } // end if
   } // end for
  } // end if
return;
}
```

```
void worker( Minimum *minimum, TspQueue *q, DistTab *distance )
{
Job_t job ;

while ( q->getJob( job ) )
  {
  tsp (MAXHOPS, job.len, job.path, minimum, *distance) ;
  }
}
```

```
Minimum *minimum;
DistTab *distance;
TspQueue *queue;

minimum = (Minimum*)minimum->attach( MinimumId
distance = (DistTab*)distance->attach( Distance
queue = new( QueueId ) TspQueue;

worker( minimum, queue, distance );
```

**figure 5.29** Tsp: création du partage et recherche du minimum par un travailleur

Si la longueur actuelle est plus grande que le plus court chemin déjà trouvé, l'exploration de la branche se termine. Si toutes les villes ont été parcourues, la distance trouvée est comparée au minimum, et le minimum est modifié si nécessaire. Sinon, une ville est ajoutée, et la nouvelle distance est calculée.

**Performances**

Nous avons évalué les performances de notre implémentation du TSP d'une part en mesurant le temps d'exécution, et d'autre part en comparant les résultats obtenus avec des implémentations réalisées avec d'autres modèles de mémoire partagées. Les modèles à notre disposition sont Crl [Johnson95] et Phosphorus [Demeure95].

1) Dream

La figure 5.30 donne en partie gauche le temps d'exécution du TSP avec Dream pour 15

villes. On constate que ce temps diminue régulièrement quand on augmente le nombre de sites (un processus travailleur par site).

La partie droite donne quant à elle le gain obtenu (speedup). Celui-ci est calculé par rapport au temps mis sur 1 site (gain = temps sur 1 site / temps sur n sites). Ce gain quasi-linéaire est conforme à ce qui est obtenu par les modèles Munin [Carter95] et Orca [Levelt92]. Il est quasi-linéaire car la parallélisation permet de trouver rapidement une bonne solution, et de réduire ainsi le nombre d'évaluations nécessaires en éliminant un plus grand nombre de branches de l'arbre.

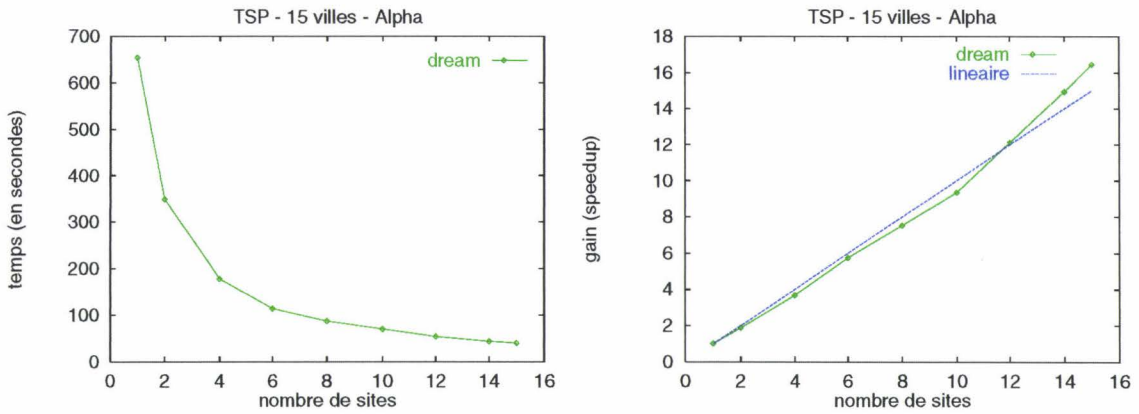



**figure 5.30** Temps d'exécution et speedup pour 15 villes avec Dream

2) Comparaison avec Phosphorus

La distribution Phosphorus ne propose pas d'exemple de TSP. Nous avons alors développé une implémentation similaire à celle de Dream, mais en utilisant la mémoire partagée de Phosphorus. Nous avons du modifier le distributeur afin d'y ajouter des données locales (non partagées) référençant les données partagées. En effet, la référence d'un objet partagé est différente sur chaque site, il faut donc la retrouver et la stocker sur chacun d'entre eux.

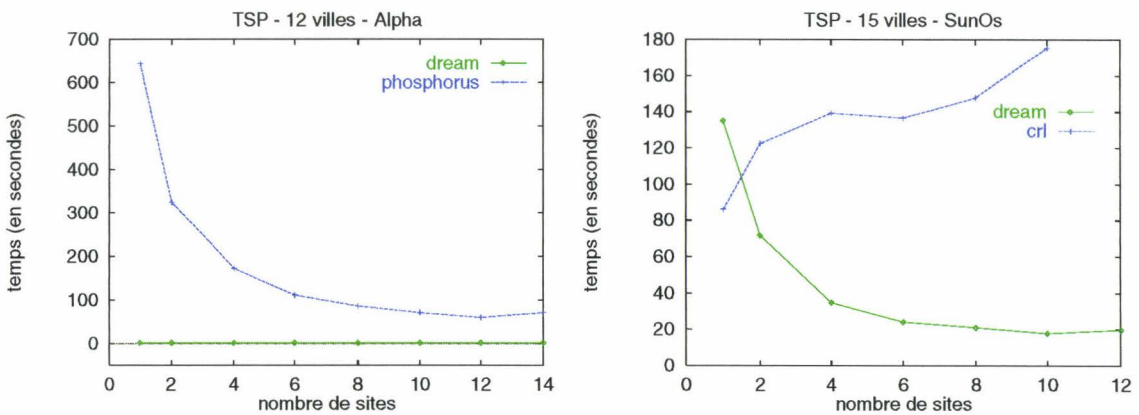



**figure 5.31** Comparaison entre Dream et Phosphorus et entre Dream et Crl

Nous avons mesuré les performances pour 12 villes sur un réseau de stations Alpha. La figure 5.31 donne les résultats. La courbe de Phosphorus est a priori correcte, et la courbe de Dream semble linéaire. En fait, la courbe de Dream est elle aussi correcte, mais les temps d'exécution varient entre 1 et 2 secondes, les temps d'exécution de Phosphorus étant de 50 à

450 fois plus élevés !

La disparité entre les temps d'exécution est ici aussi lié à la cohérence proposée d'une part par Phosphorus et d'autre part par Dream. Cette différence ce fait essentiellement lors de la consultation du minimum. En effet, cette variable est consultée à chaque comparaison. Dans Dream, la consultation est directe, alors que dans Phosphorus, la consultation est toujours précédée d'une synchronisation.

Ceci se vérifie facilement en diminuant le nombre de synchronisation dans Phosphorus, par exemple en effectuant une copie et en la synchronisant une fois toutes les cents consultations. Les temps d'exécution diminuent alors pour se situer entre six et huit secondes !

Cela revient en fait à réaliser de la cohérence faible au sens de Dream ! Cette cohérence faible est naturelle avec Dream, alors qu'avec Phosphorus nous devons effectuer une entorse au modèle de cohérence.

3) Comparaison avec Crl

Crl n'est disponible que sur SunOs. Nous avons utilisé l'implémentation du TSP qui est proposé dans la distribution Crl. Nous avons simplement modifié le jeu de distances inter-villes afin qu'il corresponde à celui utilisé par Dream.

La courbe de droite de la figure 5.31 correspond à la comparaison avec Crl. La courbe de Dream est décroissante comme il se doit, mais celle de Crl augmente avec le nombre de processeurs.

Crl utilise une cohérence par entrée, chaque accès à une variable partagée étant encadré par un début et une fin d'accès. La version du TSP proposée par Crl utilise déjà une copie du minimum, copie qui est ici synchronisée à chaque fois que le travailleur prend un nouveau sous-arbre.

Le mauvais comportement de la courbe vient à la fois de la cohérence de Crl : chaque accès à une variable partagée (arbres, distances, minimum) entraîne sa synchronisation, et d'un mauvais élagage de l'arbre (mauvaise propagation du minimum).

Il est à noter que les temps obtenus avec 15 villes sur SunOs ne sont pas comparables avec ceux obtenus sur Alpha car le jeu d'essai (distance inter-villes) est différent, et génère des arbres différents (Le jeu d'essai est obtenu par génération pseudo-aléatoire, et les générateurs différent entre Alpha et SunOs).

### 5.3.3 Conclusion

Nous venons de montrer que Dream permet aussi de réaliser des applications de type "calcul scientifique" applications qui ne faisaient pas partie des objectifs initiaux.

Ces applications ont pu être réalisées grâce aux possibilités de programmation de la cohérence, qui permet d'adapter la cohérence d'un objet aux besoins de l'application. Les performances obtenues sont très satisfaisantes, et nous considèrons que le surcoût introduit par Dream est négligeable comparé à la facilité de programmation offerte.

Avec ces applications, nous avons pu constater que l'interface de programmation proposée par Dream est simple d'utilisation. Elle permet une programmation naturelle des objets partagés, rendant leur conception et réalisation similaire à celle d'un objet local. Cette similitude permet au compilateur, dans certain cas, d'effectuer des optimisations sur des algorithmes utilisant des structures partagées. Ceci conduit à de meilleures performances pour des applications réalisées avec une mémoire partagée à la place d'échange direct de messages.

## 5.4 Dream : un "support d'exécution"

Un support d'exécution (ou *run-time*) est un ensemble de fonctionnalités nécessaires à l'exécution d'une application. Par exemple, le langage Orca que nous avons décrit page 51 fait appel à un support d'exécution dont les fonctionnalités sont spécifiées dans un manuel [Bal94].

En fait le compilateur du langage Orca génère un code intermédiaire en langage C. C'est ce code qui fait appel aux fonctionnalités du support d'exécution. Le support d'exécution est alors un ensemble de structures de données et de fonctions écrites en langage C. Le code intermédiaire et quant à lui un programme C composé d'appels aux fonctions du support.

Dans la version actuelle d'Orca, le support d'exécution est réalisé sur le système Panda [Bhoedjang93] ou sur Amoeba [Tanenbaum90]. Il existe aussi une version "Unix" dans laquelle tous les processus Orca s'exécutent sous forme de Threads dans un seul processus Unix.

Nous proposons un support d'exécution réalisé avec Dream, dans lequel les objets sont partagés à l'aide de la DSM. Le contrôle des processus est quant à lui assuré par PVM.

Pour réaliser notre run-time, nous avons repris la version "Unix", et avons effectué quelques modifications. Tout d'abord, il a fallu modifier la *création des processus* afin de créer des processus Unix au lieu des Threads. Ensuite, nous avons modifié l'allocation des structures dynamiques lors de la *gestion des objets et des structures de données* afin de prendre en compte les objets partagés. Ceci fait, il nous a fallu réécrire les fonctions contrôlant les *accès à un objet* puis modifier la fonction garantissant le mécanisme d'*attente sur une garde*.

Enfin nous avons pu tester notre run-time en compilant et exécutant des applications écrites avec le langage Orca.

**Gestion des objets et des structures de données**

Orca propose des objets et des structures de données complexes, comme des tableaux à une ou plusieurs dimensions, des graphes, des enregistrements (*records*), ou encore des sacs (*bag*) ou des ensembles (*set*).

Pour chacune des ces structures, le run-time fournit des fonctions de gestion permettant l'allocation (*allocate*), l'initialisation (*initialize*), la copie (*copy*) et l'affectation (*assign*). Il fournit en plus pour certaines structures des fonctions spécifiques, comme la taille pour les tableaux, ou la création de noeuds pour les graphes.

Pour toutes ces structures, nous avons repris l'implémentation d'origine. Celle-ci assure la définition et la gestion des structures équivalentes en C. Ces structures sont complexes, et comportent souvent des parties allouées dynamiquement en cours d'exécution. Toutes les allocations mémoire se font par un appel à une fonction spécifique du run-time (*myalloc()*) prenant comme paramètre la taille demandée, et un drapeaux indiquant si la structure est partagée ou locale.

Avec Orca, seuls les objets peuvent être partagés, mais un objet peut être composé de structures complexes pouvant elles mêmes être composées d'autres structures complexes. Une structure partagée commence donc toujours par une structure décrivant un objet.

La figure 5.32 donne l'exemple d'un objet partagé contenant un tableau de chaînes de caractères. Une chaîne de caractères est elle-même déclarée en tant que tableau, dont la taille sera spécifiée à l'instanciation de la chaîne.

```
OBJECT IMPLEMENTATION ChaineArray;
# TYPE Chaines = ARRAY[integer] OF char;
 TYPE TabChaines = ARRAY[integer] OF Chaines;

  count : integer;
  tab : TabChaines;
  ...
END;
```

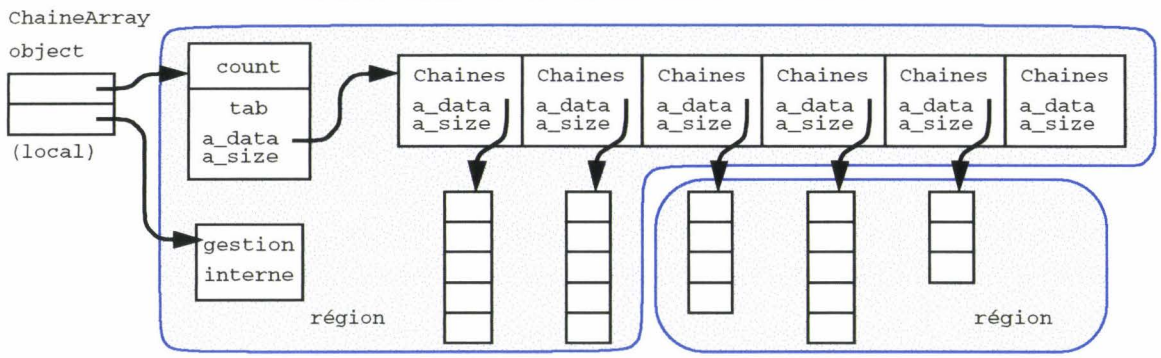



**figure 5.32** Structure d'un objet partagé Orca

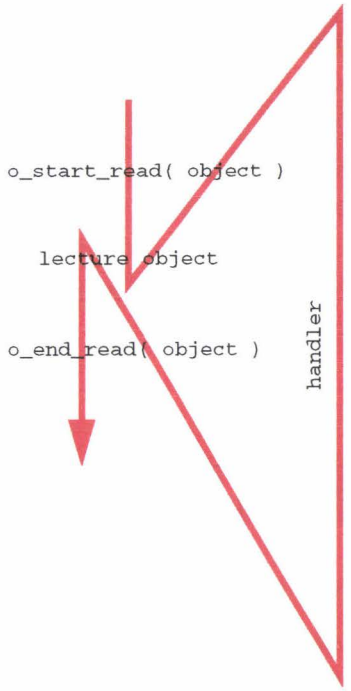
Dans notre run-time, un objet est partagé dans une ou plusieurs régions. Dans l'exemple précédent, l'objet est partagé dans deux régions. Le choix est fait en fonction de la connaissance ou non de la taille de l'objet avant sa création.

Quand la taille est connue, le partage se fait dans une région ayant exactement la taille nécessaire à l'objet. Si la taille n'est pas connue, l'objet est créé dans une région ayant au minimum la taille d'une page machine, et chaque allocation supplémentaire (pour cet objet) est faite dans cette page. Quand la page est pleine, une nouvelle région est créée pour les allocations suivantes.

**Accès aux objets partagés**

Orca n'autorise l'accès à un objet qu'à travers ses méthodes. Ceci permet de délimiter les accès à la mémoire partagée à l'intérieur de ces méthodes. Dans le code C généré, chaque accès à un objet est alors encadré par l'appel des fonctions indiquant le début ou la fin de l'accès, ainsi que le type d'accès, lecture ou écriture. Ces fonctions vont nous permettre de gérer le partage d'un objet.

Avec Dream, les accès en lecture ne demandent rien de particulier. Les régions ne sont attachées que si elles sont effectivement consultées. Ceci est réalisé en utilisant la récupération des erreurs d'accès : la lecture d'une région non attachée provoque un déroutement du programme et l'appel d'une fonction attachant la région (figure 5.33). Ensuite, l'exécution est reprise juste avant l'instruction ayant provoquée l'erreur.




```
int rgnErrorHandler( RgnMirrorDesc *localdesc, Addr addr,
RgnId id )
{
  RgnMirror rgn( localdesc );
  int rc;

  if( !localdesc )
    { // erreur en lecture
    rc = rgn.attach( addr );
    if( rc < 0 )
      printf( "Error : Can't attach region %d\n", id );
    return rc;
    }
   else
    { // erreur en écriture
    rc = rgn.waitWriteAccess();
    if( rc >= 0 )
      { // gèle et mémorise la région
      rgn.freeze();
      usedRgn.push( rgn );
      }
     else
      printf( "Error Can't get write access for %d\n", id );
    return rc;
    }
```

**figure 5.33** Attachement ou acquisition du droit d'écriture suite à une erreur d'accès

Un accès en écriture doit être précédé de l'acquisition du droit d'écriture, et suivie de son abandon. Ces deux opérations se font respectivement avant l'accès, et à la fin de cet accès. Pour les objets partagés dans plusieurs régions, le droit d'écriture est acquis avant l'accès uniquement pour la région de base (la première allouée). Il est acquis pour les autres régions lors de l'accès effectif, en utilisant ici aussi la récupération de l'erreur d'accès. A la fin, seuls les droits acquis sont relâchés.

Dans la version actuelle, nous avons choisi de contrôler explicitement la cohérence des régions. A la fin de chaque lecture, les régions ayant été modifiées sont synchronisées. Ceci permet de propager rapidement les modifications et donne une cohérence assez forte.

**Attente sur une garde**

Une garde est une condition à laquelle est associée du code. Le code est exécuté si la condition est remplie. Sinon, le programme attend que la condition soit remplie. Une garde peut être constituée de plusieurs conditions, dès que l'une d'entre-elles est remplie, le code correspondant est exécuté, et la garde se termine.

Une garde ne peut avoir lieu que dans une méthode d'un objet. Orca transforme les gardes en "opérations" retournant le résultat de l'évaluation de la condition (vrai ou faux) et effectuant le code associé si la condition est remplie.

Le code C généré est un appel à la fonction *DoOperation()* prenant comme paramètres l'objet, et la liste des opérations possibles (ensemble des gardes). *DoOperation()* exécute les opérations jusqu'a ce que l'une d'entre elle aboutisse. Si tel n'est pas le cas, *DoOperation()* attend que l'objet soit modifié avant de recommencer l'évaluation.

Avec Dream, l'évaluation des opérations ne demande rien de particulier. L'attente d'une modification d'un objet se fait quant à elle en attendant que l'une des régions contenant l'objet soit modifiée (appel à *waitUpdate()*).

### Evaluation

Nous avons compilé et exécuté avec succès les exemples fournis dans la distribution Orca.

Pour la compilation, nous avons développé un script, *oc_dream*, similaire aux scripts d'origine (*oc_unixproc*, *oc_panda*, ...). Ce script prend en charge toutes les étapes : génération du code intermédiaire en C, compilation de ce code, puis édition de lien pour obtenir un exécutable.

Les temps d'exécution des applications *tsp* et *water* ont été comparé aux temps obtenus avec le runtime Unix d'Orca. Les résultats sont résumés dans la figure 5.34.

| | TSP - 15 villes | | | | | TSP - 12 villes | | | | | | water | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| nb sites | 1 | 4 | 6 | 8 | 10 | 1 | 2 | 4 | 6 | 8 | 10 | 1 | 2 | 4 |
| orca - dream | 27'17s | 378s | 256s | 227s | 205s | 27s | 26s | 33s | 42s | 43s | 45s | 55s | 66s | 65s |
| orca - unix | 48' | | | | 47' | 14s | 7s | | 7s | | 8s | 55s | 56s | 56s |

**figure 5.34** Comparaison des exécutions entre Orca-Dream et Orca-Unix

Globalement, les temps sont meilleurs avec Dream pour le *tsp* quand le nombre de ville est important, c'est à dire quand la quantité de calcul est supérieure au nombre d'écritures en mémoire partagée. Par contre, les temps de *water* se dégradent avec l'augmentation du degré de parallélisme (du nombre de sites).

Ceci s'explique par le choix de la méthode de maintien de la cohérence : synchronisation après chaque écriture. Cette solution entraîne un grand nombre d'échange de messages, et un surcoût important pour *water.*

Ce surcoût est peu perceptible dans le *tsp*, car le nombre d'écritures est restreint (une écriture pour chaque sous-arbre pris, plus une écriture pour chaque nouveau minimum = ~250 écritures pour 15 villes). Par contre, le nombre d'écritures est bien supérieur pour *water* .

Une autre cause de ce surcoût est une fonction appelée après chaque écritures.Cette fonction vérifie que toutes les références contenues dans un objet partagé désignent des adresses en mémoire partagée. Cette fonction est nécessaire car dans certain cas (*bug* ?) le compilateur Orca effectue des copies directe de zones mémoires (*memcpy*), sans passer par les fonctions de copies, et donc sans possibilité d'allouer en mémoire partagée. Malheureusement cette fonction est très coûteuse (~10% du temps du tsp !).

### Conclusion

Nous avons réaliser une version du run-time Orca afin de montrer que cela était possible. Ce run-time peut être amélioré :

- En utilisant une méthode de maintien de la cohérence plus adaptée. Il est ainsi possible d'exploiter des informations fournies par le compilateur Orca afin d'adapter la cohérence à chaque objet.
- En supprimant le «*bug*» du compilateur. Cela demande une intervention dans le compilateur Orca lui-même.
- En améliorant la projection des objets partagés dans les régions.

## 5.5 Conclusion

Dream facilite la réalisation et la programmation d'applications distribuées demandant une mise en commun de données. Dream a été développé pour des applications «systèmes». Nous avons également montré qu'il est possible de réaliser des applications «numériques» ou encore des «supports d'exécution».

Le modèle hybride proposé par Dream permet l'utilisation des processus communicants pour certaines opérations (synchronisations, coopérations ponctuelles ou furtives), et de la mémoire partagée pour la mise en commun de données globales. Dream autorise donc la réalisation d'applications tirant parties des avantages de chacun des deux modèles

La cohérence faible et la possibilité de la contrôler offrent une «programmation naturelle» de la mémoire partagée, sans perturber le déroulement de l'application. Dream propose un certain nombre de cohérences prédéfinies adaptées à des cas généraux. Si ces cohérences ne conviennent pas, il est toujours possible d'en définir ou d'en programmer une adaptée à l'application.

Les performances de Dream sont globalement bonnes, même comparées à des solutions directement à base d'échanges de messages. Nous trouvons que le surcoût introduit est faible, surtout si on l'oppose à la souplesse de programmation offerte.

En effet, la programmation avec Dream est simple. Cette simplicité provient à la fois d'un contrôle souple de la cohérence, c'est à dire un contrôle sans contraintes, et de la possibilité de partager les références d'un objet partagé. Cette dernière spécificité apporte aussi d'autre avantages :

- La réalisation de structure partagée complexe est facilitée. Elle est similaire à son développement en mémoire locale.
- Le typage du langage est conservé. Les structures partagées peuvent donc être typées, et le compilateur effectue alors les contrôle de types et les vérifications d'usage, prévenant ainsi des erreurs courantes.
- Le compilateur peut optimiser la manipulation des structures partagées, sans intervention du programmeur (sauf la partie propre au partage : attachement, création et contrôle de cohérence).

Le développement d'applications opérationnelles comme les applications systèmes ou de calcul scientifiques montre que Dream est un modèle à part entière.

# 6 Conclusion et perspectives

## 6.1 Conclusion

Dream est un modèle de mémoire partagée définie et implémentée pendant cette thèse. Le modèle de programmation est hybride, c'est à dire qu'il permet dans un même programme d'avoir une approche basée à la fois sur le modèle des processus communicants, mais aussi sur le modèle des mémoires partagées.

Cette approche hybride permet de tirer parti des avantages des deux modèles : une mise en commun des informations par la mémoire partagée, et une meilleure expression du parallélisme par le modèle des processus communicants.

La mémoire partagée proposée par Dream se singularise par les points suivants :

- Une cohérence faible contrôlable à tout moment. Alors que les autre modèles de DSM proposent un choix limité de cohérences, voire une seule cohérence, Dream propose un contrôle précis permettant de définir toute une palette de cohérences.
- Une structuration en régions de tailles variables. Une région sert de support au partage d'objets. C'est une entité indépendante présentant sa propre cohérence. Ainsi, il est possible de réaliser des objets partagés offrant des cohérences différentes.
- Une référence (adresse) unique pour chaque objet partagé. Ainsi l'espace d'adressage de la mémoire partagée est identique dans tous les processus d'une application. Il n'y a pas de différence entre l'adressage d'un objet partagé et l'adressage d'un objet non-partagé. Les conséquences sont multiples :
    - Les références peuvent elles aussi être partagées, permettant de construire des structures partagées complexes de la même façon que si elles n'étaient pas partagées.
    - Les structures complexes peuvent être typées. Le compilateur peut alors effectuer des contrôles, et détecter d'éventuelles erreurs avant l'exécution, mais aussi optimiser le code sans intervention du programmeur. Le programmeur se focalise alors plus sur la méthodologie associée aux objets que sur leur implémentation en mémoire partagée.

Dream est un environnement portable et qui n'exige pas de dispositif spécifique, à la fois au niveau matériel et logiciel. Dream a été porté avec succès sur les trois systèmes suivant : SunOs, Solaris, OSF/1.

Dream préconise l'emploi de la mémoire partagée pour la mise en commun de données globales. Nous avons donc utilisé la mémoire partagée pour l'implémentation même de Dream ! Ainsi nous avons pu distribuer aisément une partie des algorithmes de gestion de Dream, et envisager des solutions pour un certain nombre d'autres.

Nous avons validé le modèle Dream par la réalisation d'applications fonctionnelles. Ces

applications nous ont permis de montrer que :

- Le concept de cohérence faible est adapté aux applications visées, telle que la répartition de charge. A l'origine, cette cohérence a été envisagée pour l'application de ramasse-miettes. Un prochain objectif est alors l'étude et la réalisation d'un ramasse-miettes à l'aide de la mémoire partagée.
- Le contrôle de la cohérence permet l'utilisation de Dream pour toutes sortes d'applications. Il est ainsi possible de réaliser et d'utiliser des modèles de cohérences prédéfinies, ou d'adapter dynamiquement la cohérence aux besoins de l'application.
- Le partage d'information est aisé, même en comparaison des solutions proposées par d'autres modèles de DSM. Ainsi le partage de structures de données complexes n'est pas plus difficile que leur réalisation en mémoire locale. Les différences résident dans la méthode d'allocation de la mémoire (création de régions contre allocation dans la mémoire locale) et dans le contrôle éventuel de la cohérence. La manipulation de la structure est identique en mémoire partagée ou en mémoire locale.
- L'emploi d'un langage orienté objet, et de ses concepts de classes et d'héritage permet de réaliser simplement des objets partagés : ceux-ci héritent des fonctionnalités de partage en héritant d'une classe de cohérence prédéfinie.
- Les performances des applications réalisées avec le modèle Dream sont bonnes :
  - Comparées à la cohérence d'autres modèles de DSM disponibles, les performances obtenues avec la cohérence faible de Dream sont équivalentes ou meilleures, confirmant par là l'efficacité de notre cohérence faible contrôlable.
  - Comparées aux solutions à base d'échanges de messages, ces performances sont généralement légèrement inférieures, ce qui est conforme au résultat attendu. En effet, Dream utilise les échange de messages, les performances ne peuvent alors qu'être inférieures ou égale à une solution exclusivement à base d'échange de messages. (Nous avons quand même eu la surprise d'obtenir de meilleur résultats avec Dream qu'avec une solution à base d'échanges de messages ! page 131)

## 6.2 Perspectives

Le modèle Dream tel qu'il a été défini dans cette thèse est à ce jour opérationnel. Néanmoins un certain nombre d'évolutions ou d'extensions sont envisageables.

**Evolution du modèle**

Le modèle Dream peut évoluer afin d'offrir des fonctionnalités facilitant encore plus son utilisation.

1) Multithreading

Un projet important de notre équipe porte sur la conception et la réalisation d'un environnement de programmation distribué multithreadé appelé $PM^2$ [Namyst95]. $PM^2$ propose un parallélisme à grain fin : une application est constituée de threads répartis sur différents sites et communicants par des appels RPC.

Le modèle Dream n'est pas lié au modèle de processus communicant associé. En effet, dans Dream, le modèle de processus communicant est repris tel quel, sans modification. Dream s'en sert pour son implémentation, mais ne modifie pas l'interface de programmation proposée.

Ainsi un sujet de réflexion possible concerne l'intégration de Dream et $PM^2$. Ceci permettrai de proposer un modèle de programmation distribué à grain fin (les threads $PM^2$) intégrant une mémoire partagée répartie (Dream).

Pour Dream, $PM^2$ apporte le parallélisme à grain fin par l'intermédiaire des *threads*, ainsi que les outils de communication (RPC). Les *threads* offrent une solution au problème de la gestion de Dream : celle-ci est alors effectuée dans un ou plusieurs *threads* s'exécutant concurremment au code de l'application.

Pour $PM^2$, Dream apporte le concept de mémoire partagée. Celui-ci facilite la mise en commun des données, permettant ainsi de créer des objets globaux, sans avoir à les communiquer explicitement par les RPC. La migration des *threads* peut aussi tirer parti de la mémoire partagée en créant le contexte d'un *threads* (pile, ...) dans la mémoire partagée. La migration consiste alors à préciser sur quel processeur doit s'exécuter le *thread* (et éventuellement à transférer les registres du processeur).

**Applications**

Nous avons déjà proposé plusieurs catégories d'applications pour Dream, d'autres sont encore possibles.

1) Classes de cohérences

Dream propose déjà quelques classes de cohérence prédéfinies. D'autres classes sont réalisables, par exemple en proposant les classes de cohérence étudiées dans la première partie de cette thèse. Ces cohérences permettront d'étoffer le questionnaire sur le choix d'une cohérence, notamment lors des réponses conduisant à une cohérences "forte".

Une particularité de Dream est de permettre au programmeur d'adapter dynamiquement la cohérence d'un objet. Une réflexion possible est d'étudier une classe dont le comportement s'auto-adapterait à l'objet.

2) Applications coopératives

Un dernier sujet de réflexion est l'extension de Dream aux applications coopératives.

Les applications coopératives nécessitent bien souvent la mise en commun d'informations. Une mémoire partagée telle que Dream peut alors faciliter le développement de telles applications. Nous avons déjà évoqué (chapitre 2) le cas d'une application de dialogue. Cet exemple donne un aperçu de ce qui peut être réalisé. Nous sommes allé un peu plus loin en réalisant une application de dessin coopératif. Une fenêtre de dessin est partagée entre différents utilisateurs, chacun peut y dessiner, et voir les modifications des autres.

Bien sûr cet exemple est simpliste, mais il ouvre des perspectives dans le domaine des applications coopératives.

# Annexes

## Annexe 1
## Comparaisons de modèles de DSM

Les tableaux suivant comparent quelques modèles de DSM existant actuellement.

Une case restée blanche signifie soit que les articles disponible ne permettent pas de la renseigner, soit que ce n'est pas significatif pour ce modèle.

Nous avons réparti les modèles présentés en trois tableaux, afin de faciliter la présentation. Chaque tableau comporte les mêmes rubriques dans le même ordre.

Le premier tableau contient : Munin, Midway, Treadmark et Crl

Le second :Adsmith, Phosphorus, Quarks et Dosmos

Et le troisième : Orca et Linda

Table 1:

| | Munin | Midway | TreadMark | CRL |
|---|---|---|---|---|
| **Granularité** | • Objet | • Structure (C, C++) | • Zone de mémoire de taille variable | • Régions de tailles variables. |
| **Présentation** | • Préprocesseur<br>• Linker modifié<br>• Système modifié | • Extension du langage<br>• Compilateur<br>• Run time système | • Librairie<br>• Pas de compilateur spécial | • Librairie<br>• Pas de compilateur spécial |
| **Structure des applications** | • | • | • Nombre fixe de processus, égal au nombre de sites.<br>• Espace mémoire identique sur tous les processus.<br>• SPMD | • A priori, les processus peuvent être différents.<br>• Impossible d'ajouter de nouveaux processus (dans la version actuelle). |
| **Démarrage de l'application** | • Un maître lance les esclaves. | • | • Lancement du processus maître qui se charge de lancer les autres processus. | • Lancement du processus maître qui se charge de lancer les autres processus. (dans la version actuelle) |
| **Déclaration d'un objet partagé**<br>• statique<br>• dynamique | • Statique: déclaration du type de l'objet avec ajout du mot réservé *shared*, et du type de protocole devant être utilisé. | • Statique: déclaration de la variable partagée avec ajout du mot réservé *shared*. | • Dynamique: déclaration de la référence sur l' objet partagé.<br>• Allocation avec une fonction dédiée créant la zone partagée. | • Dynamique: déclaration de la référence sur l' objet partagé.<br>• Allocation avec une fonction dédiée créant la région. |
| **Dénomination, identification** | • Nom de l'objet donné lors de sa déclaration. | • Nom de la variable donné lors de sa déclaration. | • Les zones de mémoire doivent être allouées dans le même ordre dans chaque processus. L'ordre d'allocation permet d'identifier les zones. | • Une région est nommée par un identificateur (nombre entier). |

**Table 1:**

| | Munin | Midway | TreadMark | CRL |
|---|---|---|---|---|
| **Création du partage**<br>• statique<br>• dynamique | • Statique, par le runtime. | • Statique par le runtime. | • Création dynamique des zones de partage, mais les créations doivent se faire dans le même ordre dans tous les processus de l'application. (Problème pour la création conditionnelle d'objets partagés). | • Création d'une région à partir de son identificateur.<br>• Un processus voulant utiliser une région doit la projeter.<br>• La projection permet d'initialiser la référence sur l'objet partagé. |
| **Adresse d'un objet partagé**<br>• adresse différente<br>• adresse identique | • L'adresse d'un objet partagé est différente d'un processus à l'autre. | • | • Une zone mémoire est certainement à la même adresse dans chaque processus, mais cela n'est pas exploitable par le programmeur | • L'adresse d'une région est différente d'un processus à l'autre. |
| **Accès à un objet**<br>• dans une section critique (SC)<br>• après une synchronisation | • Accès direct par l'objet.<br>• Accès obligatoire dans une section critique.<br>• Le compilateur prend en charge les conflits d'accès. | • Accès direct par l'objet.<br>• Accès obligatoire dans une SC par *lock() - unlock()*.<br>• Deux sortes d'accès: exclusif (écriture) ou non exclusif (lecture).<br>• Un objet doit être associé à une variable contrôlant la SC.<br>• La variable de contrôle peut être changée durant l'exécution. | • Accès direct par la référence de l'objet.<br>• Accès dans une SC par *lock_acquire() - lock_release()*<br>• L'accès hors des SC est possible, le programmeur doit s'assurer que les accès ne sont pas concurrents.<br>• La SC est contrôlée par une variable distincte (un entier) de l'objet. La SC assure un accès exclusif à l'objet. C'est en fait une synchronisation distincte. | • Accès direct par la référence de l'objet.<br>• Accès obligatoire dans une section critique.<br>• La SC est contrôlée par la région (il faut passer la référence de l'objet).<br>• Différencie les lectures des écritures, et pour la SC, les entrées des sorties. |
| **Synchronisation / mise à jour**<br>• fonction spécifique<br>• implicite | • La synchronisation est faite lors de la sortie de la SC. | • La synchronisation est faite avec la SC. | • La synchronisation est faite lors de l'entrée dans la SC (*lock_acquire()* ). | • La synchronisation est faite lors de l'entrée dans la SC. |

**Table 1:**

| | Munin | Midway | TreadMark | CRL |
|---|---|---|---|---|
| **Pluralité des accès / protocole** | • | • | • Multiple-writers<br>• Obtenue par création d'une page contenant les différences. Lors de la synchronisation, les différences sont propagées. Rien n'est dit sur la manière de résoudre les conflits d'écriture. | • Un écrivain, plusieurs rédacteurs. |
| **type de cohérence** | • Release consistency avec plusieurs protocoles selon le type de variable. | • Entry consistency | • Lazy release consistency. | • Entry consistency pour chaque objet. (?) |
| **Architectures cible** | • | • | • Réseau de stations | • AleWife<br>• CM5<br>• réseau de stations + PVM |
| **prototype/exemples** | • | • | • Prototype et exemples | • Prototype et exemples |
| **Remarques** | • Sert souvent de référence. | • Système de date permettant de ne transférer que ce qui a changé. | • Utilisation des signaux.<br>• Diffusion des différences. | • |
| **Auteurs** | • [Carter93] | • [Bershad91] | • [Amza95] | • [Johnson95] |

Table 2:

| | Adsmith | Phosphorus | Quarks | Dosmos |
|---|---|---|---|---|
| **Granularité** | • Structures de tailles variables | • Tableaux de tailles variables.<br>• Un type élémentaire (entier, caractère, bytes, ...) par tableaux. | • Régions de tailles variables | • Basée sur les objets (Pas de définition). |
| **Présentation** | • Librairie. | • Librairie. | • Librairie. | • |
| **Structure des applications** | • Processus différents.<br>• Ancêtre commun obligatoire. | • Ensemble de processus indépendants. | • Processus différents. | • |
| **Démarrage de l'application** | • Par le processus de départ (ancêtre commun) qui lance les autres processus. | • Chaque processus peut être lancé séparément. | • Un maître lance les esclaves.<br>• Le maître crée les régions | • |
| **Déclaration d'un objet partagé**<br>• statique<br>• dynamique | • Déclaration locale d'un objet qui sera «partagé» (buffer local)<br>• Déclaration d'un descripteur *adsmobj* associé à l'objet. | • Déclaration locale d'un tableau qui sera «partagé» (buffer local). | • Dynamique: déclaration de la référence sur l'objet à partager. | • Déclaration de l'objet partagé. |
| **Dénomination, identification** | • Un objet partagé est nommé par un identificateur (chaîne de caractères), puis manipulé par un descripteur *adsmobj*. | • Chaque objet est associé à un descripteur qui est un numéro entier. | • Identificateur de région (entier) | • |
| **Création du partage**<br>• statique<br>• dynamique | • Un processus crée l'objet partagé en lui donnant un nom. Il récupère le descripteur *adsmobj*.<br>• Les autres processus recherchent l'objet par son nom, et récupèrent le descripteur *adsmobj*. | • Dynamique: déclaration des variables partagés en précisant le nom, le type, le nombre d'éléments du tableau et le mode de partage. Les types sont les types courant de PVM (int, bytes, char, ....).<br>• Il existe plusieurs modes de partage. | • Par le maître.<br>• Lors de la création d'une région, il faut fournir l'adresse de la référence sur l'objet partagé. Puis le maître fork, et les autres processus utilisent la référence, dont le contenu a pu être modifié. | • |

Table 2:

| | Adsmith | Phosphorus | Quarks | Dosmos |
|---|---|---|---|---|
| **Adresse d'un objet partagé**<br>• adresse différente<br>• adresse identique | • L'adresse est différente d'un processus à l'autre. | • L'adresse est différente d'un processus à l'autre. | • L'adresse est différente d'un processus à l'autre.<br>• | • |
| **Accès à un objet**<br>• dans une section critique (SC)<br>• après une synchronisation | • Accès direct par l'objet.<br>• La SC est inutile. | • A l'aide de fonctions fournissant le nom et l'adresse de la variable. | • Accès direct par la référence de l'objet.<br>• SC inutile. | • |
| **Synchronisation / mise à jour**<br>• fonction spécifique<br>• implicite | • Par des fonctions, en fournissant le descripteur *adsmobj*, et un buffer pour accueillir la version la plus récente de l'objet partagé. | • La synchronisation est faite explicitement par le programmeur (read() et write()). | • Fournit des SC, mais elles n'ont pas de rapport avec l'objet partagé.<br>• La mise à jour se fait aux points de synchronisation (lock, barrier) | • Il faut déclarer les opérations de lecture-écriture effectuées sur un objet. |
| **Pluralité des accès / protocole** | • Pris en charge à l'intérieur des fonctions (a priori, un rédacteur, plusieurs lecteurs) | • Lecture seule.<br>• Migratoire.<br>• Conventionel (*invalidate*) : un écrivain, plusieurs lecteurs.<br>• Ecriture partagée (prévue). | • *Write invalidate protocol*<br>• *Write shared protocol*<br>• Le choix de la librairie détermine le choix du protocole. | • Un rédacteur, plusieurs lecteurs |
| **type de cohérence** | • *Release consistency* ou *strict consistency.*<br>• Choix par l'appel de la fonction. | • | • | • *Weak consistency* |
| **Architectures cible** | • Réseau de stations avec PVM | • Réseau de stations avec PVM | • | • |

Table 2:

| | Adsmith | Phosphorus | Quarks | Dosmos |
|---|---|---|---|---|
| **prototype/exemples** | • Prototype et exemples | • Prototype et exemples.<br>• Comparaisons avec des solutions PVM. | • Le prototype existe, mais n'est pas diffusé.<br>• L'interface du prototype est disponible. | • |
| **Remarques** | • Nous décrivons ici la première version. Une nouvelle version existe (1.0), assez différente.<br>• Seule la première version fonctionne dans un milieu hétérogène. | • Fonctionne dans un milieu hétérogène.<br>• Construit sur PVM, qui reste accessible.<br>• Partage difficile de structures complexes. | • | • |
| **Auteurs** | • [King94], [King96] | • [Demeure95] | • [Quarks95] | • [Brunie94] |

Table 3:

| | Orca | Linda |
|---|---|---|
| **Granularité** | • Objets partagés | • *Tuple* : structure formée de types élémentaires |
| **Présentation** | • Compilateur et langage de programmation. | • Librairie de fonctions<br>• préprocesseur |
| **Structure des applications** | • | • Une application est composée de plusieurs processus se partageant un *Tuple Space*. |
| **Démarrage de l'application** | • Un processus de départ crée d'autres processus. | • |
| **Déclaration d'un objet partagé**<br>• statique<br>• dynamique | • La déclaration des objets partagés est identique à celle des objets non partagés | • Les objets partagés sont dans le *Tuple Space*. Ils y sont placés dynamiquement par l'application. |
| **Dénomination, identification** | • Identification d'un objet par son nom donné lors de sa déclaration. | • Identification d'un tuple par une de ses composantes (recherche associative). |
| **Création du partage**<br>• statique<br>• dynamique | • Création de "*process*" s'exécutant en parallèle. Les objets à partager sont passés en paramètre du processus, avec le mode *shared.* | • En ajoutant et en enlevant des tuples du *Tuple Space* (opérations *in* et *out*) |
| **Adresse d'un objet partagé**<br>• adresse différente<br>• adresse identique | • | • |

**Table 3:**

| | Orca | Linda |
|---|---|---|
| **Accès à un objet**<br>• dans une section critique (SC)<br>• après une synchronisation | • Accès par les méthodes de l'objet.<br>• Le compilateur se charge des conflits d'accès. | • Pour accéder à un objet, il faut le retirer du *Tuple Space*.<br>• Il n'y a pas d'accès concurrents. |
| **Synchronisation / mise à jour**<br>• fonction spécifique<br>• implicite | • Synchronisation à la sortie de la méthode sur l'objet. | • Lors de l'ajout dans le *Tuple Space*. |
| **Pluralité des accès** | • | • |
| **type de cohérence** | • | • |
| **Architectures cible** | • Réseaux de stations<br>• Architectures avec Amoeba. | • |
| **prototype/exemples** | • Le langage existe, ainsi que son runtime implémentant le partage. | • Des implémentations existent. |
| **Remarques** | • | • |
| **Auteurs** | • [Bal92] | • [Carriero89] |

# Annexe 2
# Manuel de l'utilisateur

## *Index des fonctions*

# Classe RgnMirror

Création et manipulation des régions.

## Synopsis

**#include "dream.hxx"**

**RgnMirror object;**

## Description

La classe RgnMirror permet de créer et de manipuler des objets représentant des régions.

# RgnMirror::attach()

Attache une région miroir à l'espace d'adressage d'un processus.

## Synopsis

**#include "dream.hxx"**

**int RgnMirror::attach( RgnId ident )**

**int RgnMirror::attach( Addr addr )**

## Paramètres

| | |
|---|---|
| ident | Le nom de la région à attacher. |
| addr | Une adresse contenue dans la région à attacher. |

## Description

La fonction **attach()** projette une région dans l'espace du processus. Le programmeur fournit soit l'identificateur *ident*, soit une adresse *addr* contenue dans la région. Si la région est déjà présente dans le processus, la fonction retrouve la projection et fournit une nouvelle référence dessus. Cette fonction peut donc être appelée plusieurs fois pour la même région dans un même processus. Le processus venant d'attacher une région n'a que le droit de lecture, il ne peut que la consulter.

## Valeurs de retour

La fonction **attach()** retourne une valeur positive en cas de succès, et une valeur négative en cas d'erreur.

## Succès

La fonction **attach()** aboutit, et a rencontrée l'une des situations suivantes:

| | |
|---|---|
| no_error | La région vient d'être attachée. |
| rgn_already_attached | La région était déjà attachée. |

## Erreurs

La fonction **attach()** n'aboutit pas et aucune modification ne prend effet si l'une des situations suivantes est rencontrée:

| | |
|---|---|
| err_unknown_rgn | La région demandée n'existe pas. |
| err_owner_unknown | Le propriétaire de la région est inconnu. Ceci peut arriver si une région a changé de propriétaire. |
| err_no_mem | L'espace mémoire est insuffisant pour attacher une région de la taille demandée. |

# RgnMirror::create()

Crée une nouvelle région.

## Synopsis

**#include "dream.hxx"**

**int RgnMirror::create( Size size )**

**int RgnMirror::create( Size size, RgnId ident )**

## Paramètres

| | |
|---|---|
| size | La taille de la région à créer. |
| ident | Le nom de la région à créer (optionnel). |

## Description

La fonction **`create()`** crée une nouvelle région de taille *size*. Le système lui donne un nouvel identificateur unique qui permet de la désigner. L'utilisateur peut préciser l'identificateur désiré avec *ident*. Dans ce cas, la création n'a lieu que si l'identificateur n'est pas déjà utilisé. Le processus créateur devient propriétaire de la région et peut immédiatement écrire dedans.

## Valeurs de retour

La fonction **`create()`** retourne une valeur positive en cas de succès, et une valeur négative en cas d'erreur.

## Succès

| | |
|---|---|
| no_error | La région à été créée. |

## Erreurs

La fonction **`create()`** n'aboutit pas si une des conditions suivantes est rencontrée:

| | |
|---|---|
| rgn_already_created | Une région ayant le nom demandé existe déjà. |
| err_no_mem | L'espace mémoire est inssuffisant pour créer une région de la taille demandée. |

# RgnMirror::destroy()

Détruit une région.

## Synopsis

**#include "dream.hxx"**

**void RgnMirror::destroy( )**

## Description

La fonction **`destroy()`** permet la destruction d'une région par son propriétaire. Toutes les ressources de la région sont détruites: d'abord le descripteur global, puis les régions miroirs secondaires et la région miroir primaire.

Les processus ayant attaché la région ne sont pas avertis de cette destruction, les références qu'ils détiennent sur la région deviennent caduc ce qui peut entraîner des erreurs de type violation de la mémoire (SIGSEGV).

Un processus peut être notifié de la destruction d'une région miroir en spécifiant une fonction qui sera appelée lors de la destruction de cette région miroir.

## Voir aussi

RgnMirror::setNotifyFct()

# RgnMirror::detach()

Détache la région miroir de l'espace du processus

## Synopsis

**#include "dream.hxx"**

**int RgnMirror::detach( )**

## Description

Un processus peut détacher une région de son espace mémoire à l'aide de la fonction **`detach()`**. La zone de mémoire locale et le descripteur local sont détruits. Si le processus est le propriétaire, son titre est transféré à l'une des régions miroir encore existante. Si le processus possédait la dernière région miroir, la région est détruite, ainsi que son descripteur global (ce qui revient à une destruction de la région).

Le processus recevant le titre de propriétaire est chargé de la gestion de la région. Il n'a pas automatiquement le droit d'écriture.

## Valeurs de retour

La fonction **`detach()`** retourne une valeur positive en cas de succès, et une valeur négative en cas d'erreur.

## Succès

| | |
|---|---|
| rgn_destroyed | La région à été détruite. |
| rgn_detached | La région à été détachée. |

## Erreurs

La fonction **`detach()`** n'aboutit pas si une des conditions suivantes est rencontrée:

| | |
|---|---|
| err_unknown | Ne doit jamais arriver. |

# RgnMirror::findAgain()

Recherche d'une région par son nom ou son adresse.

## Synopsis

**#include "dream.hxx"**

**int RgnMirror::findAgain( RgnId ident )**

**int RgnMirror::findAgain( Addr addr )**

## Paramètres

| | |
|---|---|
| addr | Une adresse de la région à chercher. |
| ident | Le nom de la région à chercher. |

## Description

La fonctions **`findAgain()`** permet à un processus de retrouver une région qui a déjà été attachée. La recherche de la région se fait soit par son nom *ident*, soit par une de ses adresses *addr*. La recherche s'effectue uniquement dans le cache local au processus.

## Voir aussi

RgnMirrorDesc::findLocal()

## Valeurs de retour

La fonction **`findAgain()`** retourne une valeur positive en cas de succès, et une valeur négative en cas d'erreur.

## Succès

| | |
|---|---|
| no_error | La région à été créée. |

## Erreurs

La fonction **`findAgain()`** n'aboutit pas si une des conditions suivantes est rencontrée:

| | |
|---|---|
| err_unknown_rgn | La région n'est pas attachée localement. |

# RgnMirror::flush()

Met à jour le contenu de la région miroir.

## Synopsis

**#include "dream.hxx"**

**int RgnMirror::flush( )**

## Description

La fonction **`flush()`** synchronise le contenu de la région miroir locale avec la région. Si la région miroir est primaire, toute les copies (régions miroirs) sont synchronisées. Si la région miroir est secondaire, seule la copie locale est synchronisée avec la région miroir primaire.

Lors d'une synchronisation, l'état gelé d'une région miroir distante est pris en compte. Cela signifie que la synchronisation avec une région miroir distante ne se fait que si elle n'est pas gelé. L'état gelée de la région miroir locale n'est pas pris en compte.

Chaque synchronisation réussie d'une région miroir entraîne la mise à l'heure de sa date de synchronisation. Cette date est consultable avec la fonction **`flushDate()`**.

## Voir aussi

RgnMirror::freeze()

RgnMirror::flushDate()

## Valeurs de retour

La fonction **`flush()`** retourne une valeur positive en cas de succès, et une valeur négative en cas d'erreur.

## Succès

| | |
|---|---|
| no_error | La région à été mise à jour. |

## Erreurs

La fonction **`flush()`** n'aboutit pas si une des conditions suivantes est rencontrée:

| | |
|---|---|
| err_unknown_rgn | La région n'existe pas ou n'existe plus. |
| err_owner_unknown | Impossible de retrouver le propriétaire de la région |

# RgnMirror::flushDate()

Date de la dernière mise à jour de la région miroir.

## Synopsis

**#include "dream.hxx"**

**TimeVal RgnMirror::flushDate( )**

## Description

La fonction **`flushDate()`** renvoie la date de la dernière mise à jour de la région miroir. La classe *TimeVal* encapsule la structure Unix *"struct timeval"*. Elle fournit les opérations de base ('+', '-', '>', '==') et des méthodes de conversion entre *TimeVal* et *long* et de *TimeVal* vers *struct timeval**.

## Valeurs de retour

**`flushDate()`** renvoie un objet de type *TimeVal* contenant la date de la dernière mise à jour.

# RgnMirror::Free()

# RgnMirror::Malloc()

Allocation - libération de zones mémoires dans la limite de l'espace mémoire d'une région.

## Synopsis

**#include "dream.hxx"**

**void RgnMirror::Free( Addr addr )**

**Addr RgnMirror::Malloc( Size size )**

## Description

Les fonctions **`Free()`** et **`Malloc()`** permettent d'allouer et de libérer des sous-zones mémoire dans la limite de l'espace mémoire de la région. La région est alors gérée comme un tas.

La fonction **`Malloc()`** prend comme paramètre la taille *size* de la sous-zone à allouer dans la région, et retourne une adresse libre. Si il n'y a plus de place dans la région, **`Malloc()`** retourne zéro.

La fonction **`Free()`** prend comme paramètre une adresse *addr* retournée précédemment par **`Malloc()`**.

La première allocation se fait toujours à l'adresse de base de la région (celle retournée par **`startAddr()`** ). Il n'est pas possible de prévoir l'emplacement des allocations suivantes.

Une région peut être pleine sans que la taille cumulée des objets qu'elle contient soit égale à la taille demandée à la création. Ceci vient du fait que l'algorithme gérant l'espace d'une région alloue en réalité des objets de taille sensiblement différente de celle demandée: il arrondit la taille à la puissance de 2 immédiatement supérieure.

## Voir aussi

RgnMirror::startAddr()

## Valeurs de retour

La fonction **`Malloc()`** retourne une adresse dans la région en cas de succès, et zéro en cas d'erreur.

# RgnMirror::freeze()

Gel / dégèle la région miroir.

## Synopsis

**#include "dream.hxx"**

**void RgnMirror::freeze( )**

**void RgnMirror::unfreeze( )**

## Description

Une région dans l'état gelé n'accepte plus de demande de synchronisation venant de l'extérieur du processus. Si la région miroir est secondaire, elle ne sera plus mise à jour par la région miroir primaire. Si la région miroir est primaire, les régions miroirs secondaires ne pourront plus demander de mise à jour.

Le gel d'une région miroir n'interdit pas les mises à jour locales au processus. Celui-ci peut alors effectuer explicitement des synchronisations.

La fonction **`freeze()`** permet de faire passer une région miroir dans l'état gelé. Si la région est primaire, les régions miroirs secondaires ne sont plus synchronisées automatiquement par le système, et elle ne peuvent plus se synchroniser. Si la région miroir est secondaire, elle n'est plus synchronisée automatiquement avec la région miroir primaire. Si elle reçoit un ordre de mise à jour, celui-ci est différé jusqu'a ce que la région miroir repasse dans l'état non gelé.

La fonction **`unfreeze()`** permet de faire passer une région miroir dans l'état non gelé. Si la région miroir est secondaire, le dernier ordre de mise à jour reçu est exécuté.

# RgnMirror::waitWriteAccess()

Acquisition du droit d'écriture sur une région.

## Synopsis

**#include "dream.hxx"**

**int RgnMirror::waitWriteAccess( )**

## Description

La fonction **`waitWriteAccess()`** permet d'acquérir le droit en écriture sur la région. Ce droit doit avoir été auparavant relâché par le processus propriétaire. Si le droit est détenu par un autre processus, la fonction bloque et attend l'abandon du droit d'écriture.

## Valeurs de retour

**`waitWriteAccess()`** renvoie une valeur positive en cas de succès, ou une valeur négative en cas d'erreur.

| | |
|---|---|
| no_error | Le droit vient d'être acquis |
| rgn_already_attached | La région était déjà primaire, le droit RW est repositioné (valeur de retour positif). |

## Voir aussi

releaseWriteAccess()

## Erreurs

waitWriteAccess() échoue et aucune modification ne prend effet si l'une des conditions suivante intervient:

| | |
|---|---|
| err_bad_state_rgn | Un autre processus possède déjà le droit d'écriture sur la région. |

# RgnMirror::releaseWriteAccess()

Abandon du droit d'écriture sur une région.

## Synopsis

**#include "dream.hxx"**

**int RgnMirror::releaseWriteAccess( )**

## Description

La fonction **`releaseWriteAccess()`** permet d'abandonner le droit en écriture sur la région. Le processus reste propriétaire de la région tant que le droit n'a pas été acquis par un autre processus.

## Valeurs de retour

**`releaseWriteAccess()`** renvoie une valeur positive en cas de succès. En cas d'erreur, une valeur négative est renvoyée.

## Voir aussi

getWriteAccess()

## Erreurs

**`releaseWriteAccess()`** échoue et aucune modification ne prend effet si l'une des conditions suivante intervient:

| | |
|---|---|
| err_bad_state_rgn | Le processus ne possède pas le droit d'écriture sur la région. |

# RgnMirror::waitUpdate()

Attend que la région miroir soit synchronisée (mise à jour).

## Synopsis

**#include "dream.hxx"**

**void RgnMirror::waitUpdate( )**

## Description

La fonction **`waitUpdate()`** bloque le processus jusqu'a ce que la région miroir soit mise à jour. La région miroir est provisoirement mise dans l'état non gelé afin d'autoriser les mises à jour venant du propriétaire.

La fonction **`waitUpdate()`** appliquée à une région miroir primaire se termine immédiatement.

# RgnMirror::id()

Retourne l'identificateur de la région.

## Synopsis

**#include "dream.hxx"**

**RgnId RgnMirror::id( )**

## Description

La fonction **`id()`** retourne l'identificateur de la région référencée. Cet identificateur est unique au sein de l'application, il peut être transmis aux processus (par message ou partage). Sa connaissance permet à un processus d'attacher la région correspondante.

Le type prédéfini *RgnId* est un sous-type des entiers non signés, la conversion peut se faire dans les deux sens.

## Valeurs de retour

La fonction **`id()`** retourne l'identificateur de la région en cas de succès, et 0 en cas d'erreur.

# RgnMirror::startAddr()

Retourne l'adresse de départ de la zone de mémoire partagée.

## Synopsis

**#include "dream.hxx"**

**Addr RgnMirror::startAddr( )**

## Description

La fonction **`startAddr()`** retourne l'adresse de départ de la zone de mémoire partagée de la région. Cet adresse est identique dans tous les processus ayant attaché la région. Elle peut être transmise aux processus (par message ou partage).

Le type prédéfini *Addr* peut être converti avec les différents type de pointeurs.

## Valeurs de retour

La fonction **`startAddr()`** retourne l'adresse de départ en cas de succès, et NULL en cas d'erreur (région non existante ou non attachée).

# RgnMirror::size()

Retourne la taille de la zone de mémoire partagée.

## Synopsis

**#include "dream.hxx"**

**Size RgnMirror::size( )**

## Description

La fonction **`size()`** retourne la taille de la zone de mémoire partagée de la région telle qu'elle a été définie lors de la création.

Le type *Size* peut être converti avec les entiers non signés.

## Valeurs de retour

La fonction **`size()`** retourne la taille de la zone de mémoire partagée en cas de succès, et 0 en cas d'erreur.

# RgnMirror::split()

Divise la région miroir en plusieurs régions miroirs.

## Synopsis

**#include "dream.hxx"**

**int split( Size offset, RgnId ident );**

**int split( Size offset );**

## Description

La fonction **`split()`** divise la région miroir en deux régions à part entière. La première aura la taille de l'offset, et la seconde la taille de la région d'origine diminué de cet offset. La division se fait à l'adresse de la région d'origine, additionnée de l'offset moins un.

La région commençant à l'adresse de la région d'origine garde le nom de cette dernière. La région commençant à partir de l'offset se voit attribuer un nouveau nom, ou le nom *ident* passé en paramètre. Pour le nom, les règle sont les mêmes que lors de la création d'une région.

La fonction split ne fonctionne que si la région n'a pas encore été attachée. Dans le cas contraire, la division n'a pas lieu et une erreur est retournée.

## Voir aussi

RgnMirror::create()

## Valeurs de retour

La fonction **`split()`** retourne une valeur positive ou *nulle* en cas de succès, et une valeur négative en cas d'erreur.

| | |
|---|---|
| no_error | La région à été créée. |

## Erreurs

La fonction **`split()`** n'aboutit pas si une des conditions suivantes est rencontrée:

| | |
|---|---|
| rgn_already_created | Une région ayant le nom demandé existe déjà. |
| err_no_mem | L'espace mémoire est inssuffisant pour créer une région de la taille demandée. |

# RgnMirror::isReadOnly()

# RgnMirror::isReadWrite()

# RgnMirror::isPrimary()

# RgnMirror::isSecondary()

---

Indique le type de droit d'accès sur une région ou l'état d'une région.

## Synopsis

**#include "dream.hxx"**

**int RgnMirror::isReadOnly( )**

**int RgnMirror::isReadWrite( )**

**int RgnMirror::isPrimary( )**

**int RgnMirror::isSecondary( )**

## Description

La fonction **`isReadOnly()`** retourne vrai si le processus n'a que le droit de lecture sur la région, et faux dans le cas contraire.

La fonction **`isReadWrite()`** retourne vrai si le processus a les droits de lecture et d'écriture sur la région, et faux dans le cas contraire.

La fonction **`isPrimary()`** retourne vrai si le processus est propriétaire de la région, et faux dans le cas contraire.

La fonction **`isSecondary()`** retourne vrai si le processus n'est pas propriétaire de la région, et faux dans le cas contraire.

Un processus peut être propriétaire d'une région et ne pas avoir le droit d'écriture sur cette région, notamment si il a relâché ce droit.

## Voir aussi

releaseWriteAccess(), waitWriteAcess().

## Valeurs de retour

Les fonctions retournent 1 si l'assertion est vrai, 0 dans le cas contraire

# checkAndAllocate

# delete

# halt

---

Initialisation et destruction explicite de la DSM.

## Synopsis

**#include "dream.hxx"**

**int dream_checkAndAllocate( )**

**int dream_delete( )**

**int dream_halt( )**

## Description

**`CheckAndAllocate()`** initialise Dream si il n'existe pas déjà.

**`Delete()`** détache toutes les régions encore attachées au processus, puis termine Dream.

**`Halt()`** détruit toutes les régions encore attachées au processus avant de terminer Dream.

## Valeurs de retour

La fonction **`checkAndAllocateDsm()`** retourne 1 en cas de succès, et 0 en cas d'erreur (Dsm déjà initialisé).

La fonction **`delete()`** retourne 1 en cas de succès, et 0 en cas d'erreur (Dsm déjà détruite).

La fonction **`halt()`** retourne 1 en cas de succès, et 0 en cas d'erreur (Dsm déjà détruite).

# dream_waitFdInput()

Attente d'événements en entrée sur des descripteurs de fichier.

## Synopsis

**#include "dream.hxx"**

**int dream_waitFdInput( int *fds, int nb, TimeVal *tmout=NULL );**

**int dream_waitFdInput( DreamWaitInputOpt opt, TimeVal *tmout=NULL );**

## Paramètres

| | |
|---|---|
| fds | Tableau de descripteurs de fichier sur lesquels un événement est attendu. |
| nb | Nombre de descripteurs de fichier contenu dans le tableau. |
| opt | types de descripteur. |
| tmout | time out. |

## Description

La fonction **`dream_waitFdInput()`** permet d'attendre l'arrivée d'un événement sur un descripteur de fichier, tout en assurant la gestion de la dsm. La fonction se termine quand l'un des descripteurs passés en paramètre est prêt à être lu.

Si le paramètre *tmout* est spécifié, la fonction se termine quand le quantum de temps est écoulé, auquel cas la fonction retourne 0, ou quand l'un des descripteurs passés en paramètre est prêt à être lu, la valeur retournée est alors supérieure à 0.

Le passage des descripteurs peut se faire de deux façons différentes:

- soit en passant les descripteurs dans un tableau, il faut alors aussi indiquer le nombre de descripteurs.
- soit en précisant le ou les types de descripteur dans le paramètre *opt*. Plusieurs descripteurs peuvent être spécifiés simultanément en appliquant un ou logique sur les types. Actuellement, les types reconnus sont:
    - DREAM_WAIT_STDIN attente sur l'entrée standard
    - DREAM_WAIT_PVM attente sur les descripteurs utilisés par PVM.

Le programmeur doit utiliser **`dream_waitFdInput()`** avant chaque appel à une fonction attendant un événement sur un descripteur de fichier, par exemple avant la saisie de caractères au clavier, ou avant l'attente d'un événement graphique. **`dream_waitFdInput()`** attend l'arrivée de cet événement tout en assurant une bonne gestion de la dsm. Une fois l'événement arrivé, le programmeur peut appeler sa fonction consommant l'événement, comme par exemple *scanf()* ou *XtAppProcessEvent()*

Exemple 1 : Attente d'une saisie clavier (fd=0)

```
char str[16];
dream_waitFdInput( DREAM_WAIT_STDIN );
scanf( "%s", str );
```

Exemple 2 : Boucle de gestion des événements graphiques.

Le descripteur de fichier utilisé par X est mis dans la liste des descripteurs à surveiller. **XtAppPending()** retourne 0 si il n'y a pas d'événement graphique. Dans ce cas, **dream_waitFdInput()** permet d'attendre un événement tout en assurant la gestion de la dsm. **XtAppProcessEvent()** consomme un événement graphique, mais cette fonction bloque si l'événement n'est pas présent. On s'assure donc de sa présence par un appel à **XtAppPending()**.

```
  XtAppContext app;
  Widget frame;
      ..... Widgets initialization
  int Xfdes[1] = ConnectionNumber(
XtDisplay(frame) );
  while( TRUE )
    {
    while( XtAppPending( app ) == 0 )
      {
      dream_waitFdInput( Xfdes, 1);
      }
    XtAppProcessEvent( app, XtIMAll
);
    } //end main loop
```

## Valeurs de retour

La fonction **dream_waitFdInput()** retourne une valeur positive en cas de succès, négative en cas d'erreur, et 0 si le temps spécifié par *timeout* s'est écoulé.

| | |
|---|---|
| > 0 | Pas d'erreurs. |
| 0 | Le temps spécifié par timeout s'est écoulé, sans qu'il y ait eu d'événement sur l'un des descripteurs demandé. |
| < 0 | Erreur retournée par la fonction **select()**. |

# dream_recv()

Réception de message.

## Synopsis

**#include "dream.hxx"**

**int dream_recv( int from, int tag )**

## Description

Cette fonction permet d'attendre un message PVM sans bloquer la gestion de la DSM.

Les arguments sont les mêmes que les arguments de la fonction **`pvm_recv()`**.

Dream utilise quatre numéro de *tag* pour sa gestion interne. Si **`dream_recv`**() est appelée avec l'un de ces numéro, la fonction retourne le message qui est normalement à destination de Dream ! Cette utilisation est déconseillée.

Les tags utilisée sont :

MSG_CACHE_REQUEST=10,

MSG_CACHE_REPLY=20,

MSG_DSM_REQUEST=30,

MSG_DSM_REPLY=40

## Valeurs de retour

La valeur retourné par la fonction **`pvm_recv()`**.

# dream_trecv()

Réception de message avec timeout.

## Synopsis

**#include "dream.hxx"**

**int dream_trecv( int from, int tag, struct timeval *tmout )**

## Description

Cette fonction permet d'attendre un message PVM sans bloquer la gestion de la DSM.

Les arguments sont les mêmes que les arguments de la fonction **`pvm_trecv()`**.

Dream utilise quatre numéro de *tag* pour sa gestion interne. Si **`dream_trecv`**() est appelée avec l'un de ces numéro, la fonction retourne le message qui est normalement à destination de Dream ! Cette utilisation est déconseillée.

Les tags utilisée sont :

MSG_CACHE_REQUEST=10,

MSG_CACHE_REPLY=20,

MSG_DSM_REQUEST=30,

MSG_DSM_REPLY=40

## Valeurs de retour

La valeur retourné par la fonction **`pvm_trecv()`**.

# Dsm::find*()

Recherche d'une région par son nom ou son adresse.

## Synopsis

**#include "dream.hxx"**

**RgnGlobalDesc* Dsm::findGlobal( RgnId ident )**

**RgnGlobalDesc* Dsm::searchGlobal( RgnId ident )**

**RgnMirrorDesc* Dsm::findLocal( RgnId ident )**

**RgnMirrorDesc* Dsm::findLocal( Addr addr )**

## Paramètres

| | |
|---|---|
| addr | Une adresse de la région à chercher. |
| ident | Le nom de la région à chercher. |

## Description

Les fonctions de recherches permettent de retrouver le descripteur local ou global d'une région.

Les fonctions **`findGlobal()`** et **`searchGlobal()`** permettent de retrouver le descripteur global d'une région. Elles renvoient un pointeur sur le descripteur global qui est contenu dans la liste partagée des descripteurs globaux.

La fonction **`searchGlobal()`** recherche le descripteur global de la région uniquement dans la copie local de la liste des descripteurs globaux.

La fonction **`findGlobal()`** recherche le descripteur global en commençant par la copie locale de la liste des région. Si le descripteur n'est pas trouvé, la liste est mise à jour, et une seconde recherche est entreprise.

La fonction **`findLocal()`** permet de rechercher le descripteur local d'une région, soit par son nom *ident*, soit par une de ses adresses *addr*. Elle renvoie un pointeur sur ce descripteur local.

Ces fonctions renvoyant des références sur des structures internes de Dream, leur utilisation est déconseillée. Le programmeur utilisera de préférence la fonction **`RgnMirror::findAgain()`** qui permet de retrouver le descripteur d'une région déjà attachée.

## Voir aussi

RgnMirror::findAgain()

## Valeurs de retour

**`findGlobal()`** et **`searchGlobal()`** renvoient le pointeur sur le descripteur global en cas de succès, ou NULL en cas d'échec.

**`findLocal()`** renvoie le pointeur sur le descripteur local de la région en cas de succès, ou NULL en cas d'échec.

# dream_setUpdateTimer

Positionne le *timer* pour effectuer régulièrement la mise à jour.

## Synopsis

**#include "dream.hxx"**

**void dream_setUpdateTimer( long t )**

**void dream_setRealUpdateTimer( long t )**

## Paramètres

*t* Intervale entre deux appel. En micro-secondes.

## Description

Les fonctions **`setUpdateTimer()`** et **`setRealUpdateTimer()`** installent un *handler* appelant régulièrement la fonction **`work()`**. L'intervalle entre deux appel est exprimé en micro-secondes.

Avec la fonction **`setUpdateTimer()`**, le temps se réfère au temps du processus, alors qu'avec **`setRealUpdateTimer()`**, le temps se réfère au temps réellement écoulé (à l'horloge de la machine).

## Voir aussi

dream_work()

# dream_update

gestion interne de Dream.

## Synopsis

**#include "dream.hxx"**

**void dream_update();**

## Description

La fonctions **`dream_update()`** synchronise les régions dont le temps écoulé depuis la dernière synchronisation est plus grand que l'intervalle de synchronisation.

# dream_work

gestion interne de Dream.

## Synopsis

**#include "dream.hxx"**

**void dream_work();**

## Description

La fonctions **`dream_work()`** permet à Dream d'effectuer sa gestion interne. **`Dream_work()`** effectue un appel à **`update()`** et à **`pendingRequest()`**.

## Voir aussi

dream_update(), dream_pendingRequest().

# dream_pendingRequest

Gestion interne de Dream : prise en compte des messages à destination de la DSM.

## Synopsis

**#include "dream.hxx"**

**void dream_pendingRequest();**

## Description

La fonctions **`dream_pendingRequest()`**"consomme" tous les messages à destination de Dream.

L'appel régulier de cette fonction est nécessaire afin de garantir le bon fonctionnement de Dream.

# Annexe 3
# Listing des applications

Les codes sources des applications citées dans cette thèse sont disponibles à l'adresse suivante : http://www.lifl.fr/~dumoulin/dream/dream.html

# Bibliographie


**[Agarwal88]** A. Agarwal ,R. Simoni, J. L. Hennessy, M. Horowitz, "An Evaluation of Directory scheme for Cache Coherence", ISCA88, Proc. of the 15th Annual Int'l Symp. on Computer Architecture, p280--289, may 1988.

**[Agarwal91]** A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, B-H. Lim, G. Maa, D. Nussbaum, M. Parkin, D. Yeung, "The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor", Proc. of the 1st Workshop on Scalable Shared Memory Multiprocessors, 1991.

**[Ahamad90]** M. Ahamad, P.W. Hutto, R. John, "Implementing and Programming Causal Distributed Shared Memory",

**[Ahamad94]** M. Ahamad, J.E. Burns, P.W. Hutto, G. Neiger, P. Kholi, "Causal Memory: Definitions, implementation and Programming", TR GIT-CC-93/55, Georgia Institute of Technologie, July 94.

**[Alewife95]** A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, D. Yeung, "The MIT Alewife Machine: Architecture and Performance", Proc. of the 22th Annual Int'l Symp. on Computer Architecture, ISCA'95, jun 1995.

**[Amoeba90]** Voir [Tanenbaum90]

**[Amza95]** C. Amza and A. L. Cox and S. Dwarkadas and P. Keleher and H. Lu and R. Rajamony and W. Yu and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations", Accepted for publication in IEEE Computer, 1995.

**[Bal90]** H. E. Bal ,M. F. Kaashoek, A. S. Tanenbaum, "Experience with Distributed Programming in Orca", Proc. of the 1990 Int'l Conf. on Computer Languages, pages 79-89, March 1990

**[Bal92]** H.E. Bal, M.F. Kaashoek, A.S Tanenbaum, W.G. Levelt, "A comparison of two paradigms for distributed shared memorys", IEEE Trans. on Software Engineering, vol. 18, p. 190-205, March 1992.

**[Bal94]** C. J. H Jacobs, "Orca Runtime-system Implementors Guide", Tech. Rep., Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam.

**[Beauquier90]** J. Beauquier, B. Bérard, "Systèmes d'exploitation", Editions Mc Graw-Hill, 1990.

**[Bekele94]** D.Bekele, "Comtribution à l'étude de la répartition d'applications écrites en langage Ada 83", Thèse de doctorat en informatique, IRIT, Toulouse, October 1994.

**[Bennett90]** J. K. Bennett and J. B. Carter and W. Zwaenepoel, "Adaptive Software Cache Management for Distributed Shared Memory Architectures", ISCA90, Rice

University, Dept. of Computer Science technical report COMP TR90-109, May 1990.

**[Bershad91]** B. N. Bershad and M. J. Zekauskas, "Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors", School of Computer Science, Carnegie-Mellon University, CMU-CS-91-170, September 1991.

**[Bershad93]** B. N. Bershad and M. J. Zekauskas and W. A. Sawdon, "The Midway Distributed Shared Memory System", Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93), pp. 528-537, February 1993.

**[Bhoedjang93]** R. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, H. Bal, "Panda: A Portable Platform to Support Parallel Programming Languages," Symposium on Experiences with Distributed and Multiprocessor Systems IV, San Diego, Sep. 1993, pp. 213-226.

**[Birrell84]** A.D. Birrell, B.J. Nelson, "Implementing Remote Procedure Calls", ACM Transactions on Computer Systems, pp 321-374, 2(1) 1984.

**[Bisiani90]** R. Bisiani, M. Ravishankar, "PLUS: A distributed Shared Memory System", The 17th Annual International Symposium on Computer Architecture, 1990, pp. 115-124.

**[Brunie94]** L.Brunie, L.Levèvre, "Dosmos: A Distributed Shared Memory Based on PVM", Publication du LIP Ecole Normale Supérieure de Lyon. 1994

**[Carriero89]** N.Carriero, D. Gelernter, "Linda in context", Commun. of the ACM vol.32, pp 444-458, april 1989.

**[Carter90]** J. K. Bennett, J. B. Carter, W. Zwaenepoel, "Adaptive Software Cache Management for Distributed Shared Memory Architectures", ISCA90, pp125-135, may 1990.

**[Carter91]** J.B. Carter, J.K. Bennett, W. Zwaenepoel, "Implementation and performance of Munin" Proc of the 13th ACM Symposium on Operating Systems Principles, pp 152-164, October 1991.

**[Carter92]** J.B. Carter, A.L. Cox, D.B Johnson, W. Zwaenepoel, "Distributed Operating Systems Based on a Protected Global Virtual Address Space", Appeared as a position paper at the Third Workshop on Workstation Operating Systems, April 1992.

**[Carter93]** J. B. Carter, "Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency", Department of Computer Science, Rice University, Thesis, September 1993.

**[Carter95]** J. B. Carter, "Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems", ACM Transactions on Computer Systems, p. 205-243, vol.13, no.3, August 1995.

**[Chieh95]** Tzi-Cker Chieh, Manish Verma, "A Compiler-Directed Distributed Shared Memory System", Computer Science Department, State University of New York at Stony Brook. 1995

**[Corbin90]** J.R. Corbin, "The Art of Distributed Applications, Springer-Verlag, 1990.

**[CrayT3D]** Cray Research Inc., "Cray T3D System Architecture", Technical Report HR-

04033, September 1993.

**[Day83]** J.D. Day, H. Zimmerman, "The OSI Reference Model", Proc. IEEE, vol 71, pp. 1334-1340, December 1983.

**[Demeure95]** R. C. Dantard, I. Demeure, P. Meunier, V. Bartro, "Phosporus: a Tool for Shared Memory Management in a distributed Environment", Ecole Nationale Supérieure des Télécommunications, Département informatique, December 1995.

**[Denneulin96]** Y. Denneulin, J.M. Geib, J.F. Méhaut, "Solving irregular problems in Parallel", Parallel Optimization Colloquium 96, pp75-83, Versailles France, march 1996.

**[Dubois86]** M. Dubois, C. Scheurich, F.A. Briggs, "Memory Access Buffering in Multiprocessors", Proc. 13th Annual Int'l Symposium on Computer Architecture, ACM, pp 434-442, 1986.

**[Dubois90]** M. Dubois, C. Scheurich, "Memory Access Dependencies in Shared-Memory Multiprocessors", IEEESE, pp660--673, jun 1990

**[Dubois88]** M. Dubois, C. Scheurich, F.A. Briggs, "synchronization, Coherence, and Event Ordering in Multiprocessors", IEEE Computer, vol. 21, pp9-21, February 1988.

**[Dumoulin92]** C. Dumoulin, "Un ramasse-miettes pour les Composants Actifs de Communication", Mémoire de DEA, Communication interne, september 1992.

**[Dumoulin93]** C. Dumoulin, J.F. Roos, J.F. Mehaut, "Le Ramasse-Miettes du projet PVC-BOX", RenPar'5, 5 émes Rencontres du Parallélisme, Brest, France, juin 1993.

**[Dumoulin95a]** C. Dumoulin, "Un modèle de mémoire partagée répartie pour supports d'exécution d'applications parallèles", 7 émes Rencontres du Parallélisme, Mons, Belgique, June 1995.

**[Dumoulin95b]** C. Dumoulin, "DREAM: A Distributed Shared Memory model using PVM", 2nd EuroPVM UG Meeting, Ecole normale superieure de Lyon, France, September, 1995.

**[Ferreira96]** P. Ferreira, "Larchant : ramasse-miettes dans une mémoire partagée répartie avec persistance par atteignabilité", Thèse de doctorat, Université Pierre et Marie Curie, Paris, Mai 1996.

**[Fleish89]** B. Fleish, G. Popek, "Mirage: A coherent distributed shared memory design", in Proc. of 12thACM Symposium on Operating Systems Principles, pages 211-223 Litfield Park, AZ, December 3-6 1989.

**[Flynn72]** M.J. Flynn , "Some Computer Organizations and Their Effectiveness", IEEE Trans. on Computers, vol. C-21, pp 948-960, Sept. 1972.

**[Geist94]** A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, "PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing", The MIT Press, 1994.

**[Gharachorloo90]** K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors", in Proc. of the 17th Annual International Symposium on Computer Architecture, pages 15-26, Seatle, Washington, May 1990.

**[Gransart95]** C. Gransart, "BOX : {U}n modèle et un langage à objets pour la programmation parallèle et distribuée", PhD Thesis, Université de Lille I, janvier 1995.

**[Grenot95]** C. Gransart, C. Grenot, P.Merle, "TOSCA : Support de Threads et d'Objets pour Applications Coopératives", Actes des Journées des Jeunes Chercheurs en Systèmes Informatiques Répartis, IRISA, Rennes, october 1995.

**[Griffiths90]** M. Griffiths, M. Vayssade, "Architecture des systèmes d'exploitation", Traité des Nouvelles Technologies, Série Informatique, Editions Hermes 1990.

**[Goodman89]** J. R. Goodman, "Cache Consistency and Sequential Consistency", IEEE Scalable Coherence Interface Working Group, March 1989.

**[Hemery94]** F. Hemery, "Etude de la répartition dynamique d'activités sur architectures décentralisées", PhD thesis, Université des Sciences et Technologies de Lille, Laboratoire d'informatique Fondamentale de Lille, Juin 1994.

**[Hoare78]** C.A.R. Hoare, "Communicating Sequential Processes", Communications of the ACM, Vol 21, Num 8, pp 666-677, August 1978.

**[Hutto90]** P.W. Hutto, M. Ahamad, "Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories", Proc. 10th Int'l Conf. on Distributed Computing Systems, IEEE, pp 302-311, 1990.

**[Johnson95]** K. L. Johnson, M. F. Kaashoek, D. A. Wallach, "CRL: High-Performance All-Software Distributed Shared Memory", Proc. of the Fifth Workshop on Scalable Shared Memory Multiprocessors, Jun 1995.

**[Kafura91]** D. Kafura and D. Washabough, "Progress in the Garbage Collection of Active Objects", ACM OOPS Messager, 2(2), pp. 55-58, April 1991.

**[Keleher92]** P. Keleher and A. L. Cox and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory', ISCA92, pp. 13-21, May 92.

**[Keleher95]** P. Keleher, "Lazy Release Consistency for Distributed Shared Memory", Department of Computer Science, Rice University, Thesis, January 1995.

**[Kermarrec95]** A.-M. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, I. Puaut, "A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability", Proc. of the 25th Annual Int'l Symp. on Fault-Tolerant Computing (FTCS-25), pp289-298, jun 1995.

**[King94]** Chung-Ta King, Wen-Yew Liang, "ADSMITH: A Structure-Based Heterogeneous Distributed Shared Memory on PVM", Institute of Computer Science National Tsing Hua University, June 1994.

**[King96]** William W. Y. Liang, "Adsmith 1.0: An Efficient Object-Based DSM Environment on PVM", & Information Engineering National Taiwan University, Taipei, Taiwan, February 1996.

**[Lamport79]** L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs", IEEE Trans. on Computers, vol. C-28, pp 690-691, September 1979.

**[Levelt92]** W. G. Levelt, M. F. Kaashoek, H. E. Bal, A. S. Tanenbaum, "A Comparison of Two Paradigms for Distributed Shared Memory", in Software-Practice and Experience, vol 22, no11, p. 985-1010, November 1992.

**[Li86]** K. Li, "Shared Virtual Memory on Loosely Coupled Multiprocessors", Ph.D. Thesis, Yale University, 1986.

**[Li89]** K. Li and P. Hudak, "Memory coherence in shared virtual memory systems", ACM Transactions on Computer Systems, 7(4):321-359, November 1989.

**[Lipton88]** R.J. Lipton, J.S. Sandberg, "PRAM: A Scalable Shared Memory", Tech. rep. CS-TR-180-88, Princeton University, September 1988.

**[Mossiere94]** J. Mossière, X. Rousset de Pina, "Single address space or private address spaces ?", Bull-Imag//systèmes, 1994.

**[MPI95]** Message Passing Interface Forum V1.1, Document for a standard message-passing interface, (from http://www.mcs.anl.gov/Projects/mpi/index.html), june 1995.

**[MPI96]** Message Passing Interface, SC'96 MPI-2-Birds-Of-a-feather, Supercomputing'96.

**[Namyst95]** R. Namyst, J.F. Méhaut,"$PM^2$ Parallel Multithreaded Machine.A computing environment for distributed architectures", Parco95, Gent Belgium, September 1995.

**[Occam88]** "OCCAM 2 Reference Manual", Prentice-Hall International Series in Computer Science. Prentice-Hall, Englewood Cliffs, 1988.

**[Ortega93]** M.I. Ortega, "La mémoire virtuelle partagée répartie au sein du système Chorus" Thèse de l'université de Paris VII, février 1993.

**[Perez96]** E. Pérez-Cortés, J. Han, J. Mossière, "Construction de protocoles de cohérence sur une interface générique de mémoire partagée répartie", MPR'96, Bordeaux, May 1996.

**[Quarks95]** Dilip Khandekar, "Quarks: Portable DSM on Unix", Computer Systems Laboratory, Department of Computer Science, University of Utah, 1995.

**[Raynal91]** A. Maddi, M. Raynal, "Implementing Semaphores on a Distributed Memory Parallel Machine", Proc. of the Int'l Conf. on Parallel Computing (ParCo91), pp407--412, september 1991.

**[Renesse93]** Robbert van Renesse, Robert Cooper, Bradford Glade, Patrick Stephenson "A RISC Approach to Process Groups", Proceedings of the 5th ACM SIGOPS Workshop, Sep. 21-23, 1992, Rennes, France.

**[Rifflet95]** Jean-Marie Rifflet, "La communication sous Unix", $2^{nd}$ édition, Ediscience international 1995.

**[Roos94]** J.F. Roos, "Mise au point d' applications distribuées pour environnement de développement basé sur une technologie objet", Thèse de l'université de Lille, 1994.

**[Sieglin96]** C. Siegelin, I. Demeure, U. Finger, P. Meunier, "WARPphos: a Hierarchical Hardware and Software Shared Memory System for a Network of Workstations", International Conference on Telecommunication, Distribution, Parallelism TDP'96, Cagliari, Italy, May 1996.

**[Silberschatz94]** A. Silberschatz, P. B. Galvin, "Principes des systèmes d'exploitation", Editions Addison-Wesley, 1994.

**[Saunier95]** F. Saunier, "Service de Protection d'une Mémoire Virtuelle répartie dans Sirac", Journées des Jeunes Chercheurs, Réseau Doctoral en Architecture des Systèmes et Machines Informatiques, ed, Irisa, Rennes, France, October 1995.

**[Stroustrup93]** B. Stroustrup, "The C++ programming Language", InterEditions, 1993.

**[Tanenbaum88]** A.S. Tanenbaum, "Computer Networks", Prentice Hall International editions, 1988.

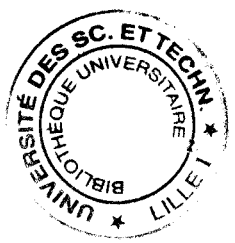
**[Tanenbaum90]** A.S. Tanenbaum, R. Van Renesse, H. Staveren, H. Van Sharp, S.J. Mullender, J. Jansen, G. Rossum, "Experiences with the Amoeba Distributed Operating System", Comm. of ACM, vol. 33, pp.46-63, December 1990.

**[Tanenbaum95]** A. S. Tanenbaum, "Distributed Operating Systems", Prentice Hall International editions, 1995.

**[Wu90]** K.-L. Wu, W. K. Fuchs, "Recoverable Distributed Shared Memory", IEEE transaction on Computer, 39(4), pp460-469, april 1990

**[Zhou]** S. Zhou, "Load Sharing and Batch Queueing Software", Tecnical Report, Platform Computing Coporation, Ontario, Canada. http://www.platform.com

**[Zucker94** R. N. Zucker and J-L. Baer, "Software versus Hardware Coherence: Performance versus Cost", HICSS-27, Proc. of the 27th Hawaii Int'l Conf. on System Sciences, pp163--172, january 1994.




Une bibliographie complète sur les mémoires partagées est disponible à l'adresse suivante :

http://www.cs.ualberta.ca/~rasit/dsmbiblio.html