Conception et réalisation d'un noyau d'administration d'un système réparti à objets persistants

Ibaa Oueichek

THÈSE

présentée par

Ibaa Oueichek

pour l'obtention du grade de

Docteur de l'Institut National Polytechnique de Grenoble

(Arrêté ministériel du 30 mars 1992)

Spécialité : **Informatique**

Conception et réalisation d'un noyau d'administration pour un système réparti à objets persistants

présentée devant la commission d'examen le 23 octobre 1996

Composition du jury

Président :	Jacques MOSSIERE
Rapporteurs :	Mansour FARAH Rachid GUERRAOUI
Examinateur :	Roland BALTER
Directeur de thèse :	Xavier ROUSSET de PINA

Je tiens à remercier

Monsieur Jacques Mossière, Professeur à l'Institut National Polytechnique de Grenoble, qui m'a fait l'honneur de présider le jury de thèse,

Monsieur Mansour Farah, Directeur de recherche et directeur du département d'Informatique à l'Institut Supérieur de Sciences Appliquées et de Technologies de Damas (ISSAT), et Monsieur Rachid Guerraoui, premier assistant à l'École Polytechnique Fédérale de Lausanne, qui ont accepté d'être les rapporteurs de mon travail, et dont les remarques ont considérablement contribué à l'amélioration du document,

Monsieur Xavier Rousset de Pina, Professeur à l'Institut National Polytechnique de Grenoble, qui m'a encadré pendant les années de ma thèse et dont le soutien scientifique, mais aussi moral, m'a permis d'aller jusqu'au bout de cette thèse,

Monsieur Roland Balter, Professeur à l'Université Joseph Fourier et responsable du projet Sirac, qui a accepté d'être membre du jury de thèse,

Monsieur Guy Mazaré, Professeur à l'Institut National Polytechnique de Grenoble, et directeur de l'Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble, qui m'a accueilli au sein de son équipe pendant les deux premières années de ma thèse,

Monsieur Roland Balter, et Monsieur Sacha Krakowiak, Professeurs à l'Université Joseph Fourier, pour la confiance qu'ils m'ont accordé en m'accueillant dans l'Unité Mixte Bull-IMAG et ensuite dans le projet Sirac.

Je tiens également à remercier toutes les personnes qui m'ont aidé et encouragé et plus particulièrement Michel Riveill, Professeur à l'Université de Savoie, pour son expertise sur bien des problèmes, ainsi que la direction de l'ISSAT qui a financé mes études.

Merci aussi à Alain Knaff, compagnon des bons et des mauvais jours, et un inconditionnel de Linux, pour ses nombreux services et son humour. Sans oublier tous les membres de l'ex-Unité Mixte Bull-IMAG devenue projet Sirac.

1

Introduction

La complexité et le degré de sophistication des outils d'administration ont suivi l'évolution des systèmes d'exploitation depuis les systèmes centralisés jusqu'aux systèmes répartis en passant par les réseaux d'ordinateurs. L'administration dans les premiers systèmes centralisés était réduite à quelques tâches simples et répétitives : gestion des utilisateurs, gestion des sauvegardes, et facturation de l'utilisation des ressources. Le passage vers les réseaux d'ordinateurs s'est accompagné de la définition de protocoles et d'outils essentiellement tournés vers l'audit (monitoring) des unités intervenant dans les couches basses du réseau (répéteurs, ponts, routeurs). Beaucoup de ces unités sont instrumentées pour répondre aux requêtes du protocole SNMP [SFDC90] (Simple Network Management Protocol), protocole de la famille Internet pour l'administration des réseaux. Ainsi, les systèmes d'administration basés sur SNMP fonctionnent essentiellement en collectant des informations sur les éléments constituant le réseau et sur leur utilisation, ainsi que des rapports sur les divers événements intervenant sur le réseau. Les données collectées sont utilisées par les administrateurs pour prendre les décisions appropriées. Ces systèmes utilisent souvent des solutions *ad hoc* qui ne permettent pas l'administration des systèmes répartis. Une des carences des systèmes actuels est qu'ils ne permettent pas en général de tenir compte des dépendances existant entre les divers éléments qui les constituent. Ceci a pour conséquence qu'en cas de panne d'un élément, les éléments qui en dépendent risquent d'être traités comme étant défaillants alors qu'il ne le sont pas. Cela peut déclencher des fausses alertes en cascade[Ba94], ce qui peut être délicat à gérer. Les relations de dépendance entre éléments sont nombreuses dans un système réparti, et il est donc indispensable d'en tenir compte. D'autre part, les outils sont presque entièrement tournés vers la tâche d'audit (monitoring) et donc de surveillance au détriment de l'activité de *contrôle*.

La suite de ce chapitre est structurée de la manière suivante. Nous dégageons tout d'abord les problèmes généraux qui doivent être résolus pour administrer un système réparti. Nous présentons alors les services d'administration tels qu'ils sont définis dans le modèle OSI (Open System Interconnection) de l'ISO (International Standard Organisation) et un modèle classique d'architecture de système d'administration le mettant en œuvre. Nous nous consacrons ensuite aux services de base qui ont fait l'objet principal de cette étude car ils permettent la définition d'une couche minimale d'administration qui à son tour peut être utilisée pour réaliser les autres services.

Nous terminons par une description de ce que nous estimons être les apports de notre travail et par la présentation du plan du reste de ce document.

1 Problèmes posés par l'administration des systèmes répartis

L'administration des système répartis est particulièrement difficile si on la compare à celle des système centralisés. Nous donnons une liste non-exhaustive des sources de difficultés qui compliquent la réalisation du système d'administration :

Problème de vulnérabilité. La vulnérabilité est due aux pannes des éléments qui constituent le système. Le nombre important de ces élément augmente considérablement la probabilité d'avoir un ou plusieurs d'entre eux en panne. Cette vulnérabilité est accrue par les problèmes de fiabilité des réseaux de communication (les machines peuvent très bien fonctionner normalement et pourtant être considérées comme non opérationnelles à cause d'une coupure ou d'un dysfonctionnement du réseau). De plus, les réseaux constituent une grande source de problèmes de sécurité et d'intrusion. Ils rendent la protection physique des machines et des données qu'elles stockent (restriction d'accès aux salles) complètement illusoire. Un autre risque apparaît au niveau des informations transmises sur le réseau. Il faut donc développer des outils permettant de contrôler l'accès aux informations stockées sur les machines ainsi qu'aux informations transmises pour garantir leur confidentialité et leur intégrité.

Problème de dispersion géographique. Cette dispersion rend la construction d'une vision cohérente d'un état global du système difficile et coûteuse. Il en résulte une grande difficulté pour prendre des décisions administratives et pour synchroniser les actions concernant les différents composants du système.

Problème de facteur d'échelle et de hétérogénéité. Le nombre de ressources à gérer (sites, volumes de stockage, périphériques, etc.) peut atteindre des valeurs très importantes (le service d'administration du projet Athena [Ra88] gère des centaines de sites et des milliers d'utilisateurs). Il est donc impossible de traiter chaque ressource ou composant comme une entité indépendante. Un tel problème se pose avec SNMP, où le manque de règles de classification et de regroupement s'est traduit par un nombre impressionnant de variables. On compte en effet que plus de vingt milles variables ont été définies [Pra95]. Il est clair que l'administration d'un tel système nécessite un grand effort d'apprentissage et que les risques d'erreurs sont très grands.

Problème d'efficacité. Les propriétés souhaitables dans ces systèmes, comme celle consistant à cacher la localisation géographique des ressources, ou à offrir une bonne tolérance aux pannes ne peuvent être obtenues que grâce à des mécanismes qui sont souvent sources d'inefficacité et qui donc compliquent la tâche de l'administrateur. De plus, les services d'administration utilisent aussi les ressources du système, l'exemple le plus évident est la bande passante utilisée

par le service d'audit qui produit ses rapports d'événements sous forme de messages. Il faut donc veiller à ce que les ressources allouées à l'administration restent dans les limites du raisonnable et que l'activité d'administration ne perturbe pas le fonctionnement normal du système.

Problème d'organisation. Un système réparti a souvent besoin de plusieurs administrateurs. C'est parce que la taille du système et la diversité des tâches administratives peuvent dépasser les capacités d'une seule personne. C'est aussi parce qu'un système réparti peut être dispersé sur plusieurs sites géographiques voire plusieurs organisations indépendantes. Il faut donc définir une stratégie d'administration qui permette d'organiser les interactions entre les administrateurs et le système, ainsi que les interactions entre les administrateurs eux-mêmes afin d'éviter les conflits.

Si l'on veut remédier à ces difficultés, il faut poser les problèmes d'administration *pendant la phase de conception* du système et non pas après. Le(s) service(s) d'administration devient alors une partie intégrante du système et non pas une couche qui vient s'ajouter à la fin. Les analyses faites pendant la phase de conception du système doivent dégager des réponses aux questions suivantes:

1. Quelles sont les tâches du service d'administration?
2. Quels sont les éléments à administrer?
3. Quels outils offrir à l'administrateur du système?
4. Quels sont les mécanismes que le système lui même doit offrir pour faciliter la réalisation de ces outils? En d'autres termes, le système doit être conçu de façon à être *administrable.*

Il ne s'agit donc pas de construire une simple boîte à outils qui fonctionne au dessus d'un système quelconque sans prendre en compte les capacités du système, ni d'offrir un service d'administration complètement lié à un système particulier et donc non portable. Mais de chercher à établir un modèle générique pour l'administration des systèmes répartis, qui cherche à incorporer un petit nombre de mécanismes dans le corps du système pour rendre les tâches d'administration plus simples.

2 Les fonctions d'administration

Pour diminuer la complexité du problème, on peut diviser la tâche d'administration en plusieurs fonctions administratives. C'est l'approche qui a été adoptée dans le modèle OSI [Kle88], et qui est reconnue comme la décomposition la plus complète. Les principales fonctions définies par l'OSI sont:

La gestion de la configuration. La gestion de la configuration est concernée par l'initialisation du système et de ses composants ainsi que par l'ajout et le retrait de composants au sein du système. Elle comprend des tâches telles que l'installation des composants matériels et logiciels dans un système ou une

application répartie, la création des composants logiciels et l'allocation de ces composants au matériel, le service de désignation (passage nom symbolique $\leftrightarrow$ objet physique), le contrôle des composants d'un serveur. En plus on doit prévoir l'évolution du système afin de pouvoir incorporer de nouvelles fonctions.

La gestion des performances. Ce service s'intéresse à l'optimisation des performances pour améliorer le service fourni aux utilisateurs que ce soit au niveau du temps de réponse ou de la fiabilité. L'administrateur s'intéresse aux points suivants :

1. Quel est le niveau d'utilisation des ressources du système?
2. Existe-t-il des goulots d'étranglement ou une concentration de l'activité en un point particulier?
3. Est-ce que le temps de réponse est acceptable?

La gestion des pannes. Il s'agit de s'assurer que l'ensemble du système ainsi que chaque composant individuel fonctionne correctement. Quand une faute se produit, il faut que le service responsable de la gestion des pannes effectue les traitements suivants le plus rapidement possible :

1. Détecter et localiser la faute.
2. Confiner et isoler la faute pour empêcher sa propagation.
3. Effectuer un recouvrement de l'erreur provoquée par la faute. Ceci peut nécessiter la reconfiguration du système.
4. Réparer ou remplacer le composant défaillant pour revenir à l'état normal.

Le traitement doit être rapide et fiable. La plupart des utilisateurs peuvent tolérer des pannes non fréquentes. Mais quand les pannes se produisent, ils demandent qu'elles leurs soient signalées *immédiatement* et que le problème soit résolu rapidement.

La gestion de la sécurité. La gestion de la sécurité constitue l'une des fonctions les plus importantes dans un système d'exploitation. Elle concerne l'ensemble des mécanismes de sécurité au sein du système, comme le contrôle d'accès, le cryptage et la sécurité physique. Ce service comporte des fonctions comme la distribution des clés de cryptage, la gestion et la protection des informations de contrôle d'accès et la surveillance des tentatives d'accès non autorisées.

La gestion de la comptabilité. Ce service permet aux utilisateurs d'obtenir une information sur l'utilisation des ressources et permet aux fournisseurs de facturer leur utilisation de ces ressources. Il permet aussi de s'assurer que les ressources sont utilisées de façon équitable et qu'il n'y a pas d'utilisateurs qui abusent de leurs privilèges et privent les autres des ressources du système. Les informations ainsi récupérées peuvent être utilisées pour planifier l'évolution du système.

La gestion de l'audit. Toutes les fonctions citées ci-dessus ont besoin pour leur mise en œuvre d'une fonction qui ne fait pas partie des services définis dans le modèle OSI. Il s'agit de la **gestion de l'audit**. Elle est concernée par la récupération, l'analyse et la surveillance des informations sur l'état, les erreurs, les performances et l'utilisation du système. Cette fonction permet de vérifier le bon fonctionnement de tous les autres services de l'administration.

3 Architecture du service d'administration

La figure 1.1 présentée par [Sta93] présente une vision générale de l'architecture du service d'administration telle qu'elle est normalement admise. Ce service se décompose en trois couches :

- l'interface utilisateur,
- les fonctions d'administration,
- et la couche de support de l'administration.

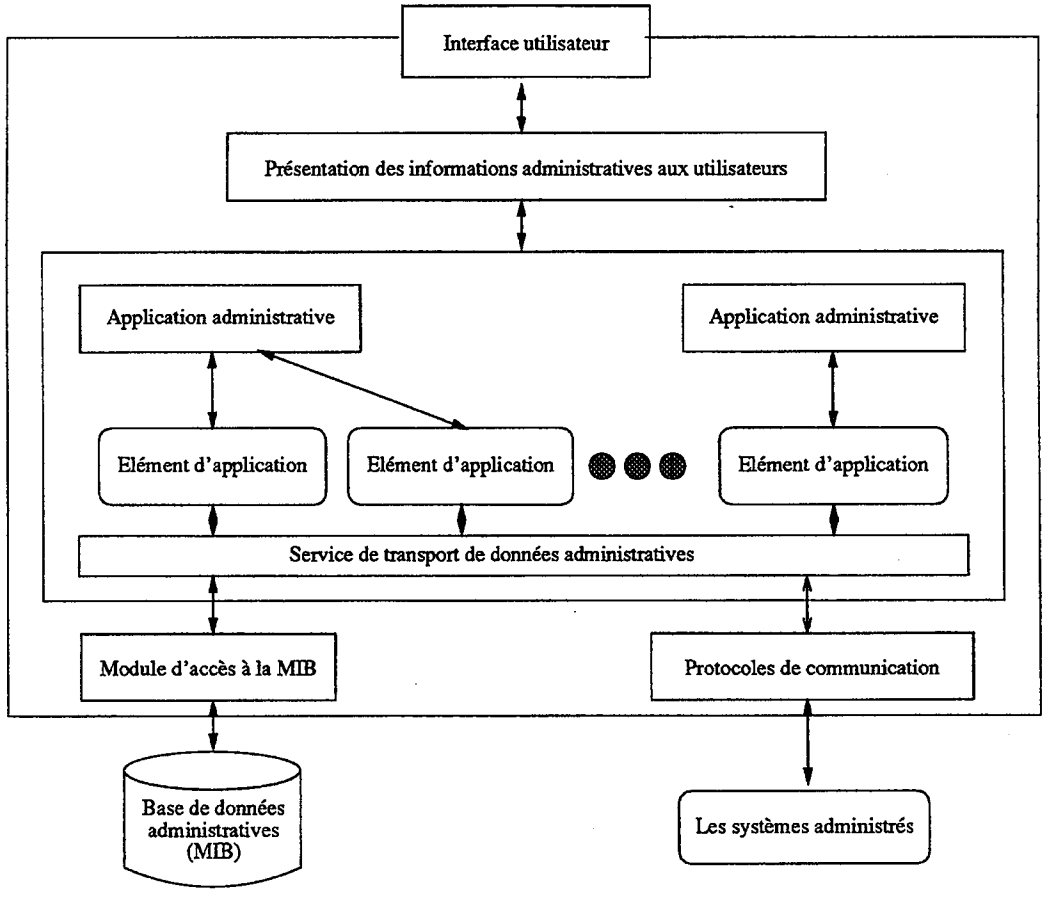

FIG. 1.1 – *Architecture générique pour un service d'administration*

Interface utilisateur L'interface utilisateur est un élément clé dans la conception d'un service d'administration. Cette interface remplace la console traditionnelle des systèmes monolithiques. La particularité de cette interface résulte du volume énorme des informations produites par le service de gestion de l'audit et qui sont à gérer. Ces informations doivent être regroupées et simplifiées au maximum sans perdre pour autant de données importantes.

Les fonctions d'administration La couche logicielle qui fournit l'ensemble des fonctions d'administration est à son tour décomposée en trois couches. La couche la plus haute est constituée des applications administratives qui fournissent les services d'administration aux utilisateurs (par exemple, la gestion des comptes). Ces applications sont basées sur un ensemble *d'éléments d'application.* Il s'agit là d'un ensemble de modules primitifs qui réalisent les fonctions de base comme la génération des notifications et la récupération des données d'audit ; la distinction entre les deux couches permet la réutilisation de ces modules dans plusieurs applications administratives. La couche la plus basse est celle du transport des données administrative. Cette couche est responsable de l'échange d'informations entre les différents éléments d'application d'un côté, et les entités administrées d'un autre. Elle offre souvent un service très primitif, comme la récupération d'informations, la mise à jour des paramètres, et la génération des notifications.

Couche de support à l'administration Cette couche offre l'accès aux informations qui décrivent l'ensemble des entités administrées et leur comportement. Ces informations sont stockées dans une base de données administratives (MIB : Management Information Base). La structure des données utilisées dans la MIB et leur format a une grande influence sur la complexité et la diversité des services offerts par les fonctions d'administration. Cette couche définit aussi les protocoles de communication qui sont utilisés pour communiquer entre les différents sites du réseau (TCP/IP, OSI).

4 Le sujet de notre travail

Notre travail se situe dans le domaine des services d'administration tels qu'ils sont présentés dans le modèle OSI. Plus spécifiquement, nous nous sommes intéressé à la gestion de la configuration et à la gestion de l'audit, et nous avons défini et mis en œuvre une couche minimale d'administration d'un système réparti à objets. Cette section a pour but de justifier ce choix en montrant l'importance de ces deux fonctions.

4.1 La gestion de la configuration

Ce service s'occupe essentiellement des tâches suivantes :

- La représentation et la spécification des entités administrées et de leur comportement. Cette représentation sera utilisée par les utilisateurs et les applications (y compris les autres services d'administration) pour accéder aux informations

dont ils ont besoin sur les ressources du système. Pour pouvoir décrire cette configuration, on a besoin d'un *langage de spécification de configuration.*

- La gestion des changements qui peuvent se produire concernant les ressources et les éléments du système. Un exemple de tels changements est la déconnexion volontaire d'une machine, le démarrage et l'arrêt d'un site et la mise à jour d'un composant. Ces changements sont la conséquence naturelle des évolutions économiques, géographiques et technologiques de l'environnement du système.

- La gestion du service de désignation. Ce service permet d'offrir un nom à chaque objet du système, et de retrouver l'objet à partir de son nom.

Vu l'ensemble de tâches de ce service, on voit clairement qu'il s'agit d'un service de base auquel tout autre service a recours à un moment ou à un autre. Prenons par exemple le service de gestion des performances. Ce service doit s'occuper de l'allocation des ressources aux différents processus en cours d'exécution. Pour pouvoir prendre des décisions, il a besoin d'informations *contractuelles* concernant ces ressources (comme le taux d'utilisation, la localisation géographique, la disponibilité, etc.). Ces informations seront fournies par le service de gestion de la configuration, grâce à la représentation des entités administrées que ce service exporte. Le service de gestion des pannes peut demander l'arrêt d'un élément défaillant du système et le démarrage d'un élément de réserve pour remplacer l'élément défaillant, il aura alors recours au mécanisme de reconfiguration.

4.2 La gestion de l'audit

L'autre service de base est celui de gestion de l'audit. En effet, comme nous l'avons déjà mentionné, le service d'audit a souvent pris une place très importante dans les systèmes d'administration de réseaux. Tous les services d'administration auront besoin à un moment ou un autre d'utiliser les informations produites par ce service. Il suffit de prendre l'exemple de la gestion de la configuration qui doit mettre à jour la configuration du système en fonction des événements qui se produisent au sein du système et qui modifie sa configuration ; c'est le service de gestion d'audit qui doit lui envoyer les rapports qui décrivent ces événements.

Le modèle décrit dans [MSS93] et [FE89] définit quatre activités qui constituent le service de l'audit :

Génération : Il s'agit de produire des rapports concernant l'état des entités du système, et les événements qui affectent ces entités.

Traitement : Les informations produites sont validées, filtrées, et stockées.

Distribution : Les rapports sont distribués aux utilisateurs et aux agents administratifs qui les réclament.

Présentation : Les informations sont présentées aux applications selon le format demandé par ces applications (présentation textuelle, graphes, diagrammes temporels).

Etant donné que nous nous intéressons à la réalisation d'un service *minimal* d'administration, nous réduirons l'activité de l'audit au sein de ce service à la production des rapports d'audit. Cette activité est réalisée grâce à un mécanisme de gestion d'événements qui a été spécifié et réalisé par Emmanuel Lenormand [BLM96].

4.3 Couche minimale d'administration

Une bonne solution est donc de définir un ensemble de services de base pour l'audit et la configuration qui à leur tour sont utilisés par les autres services d'administration. Notre travail consiste à concevoir et à mettre en œuvre cette *couche minimale d'administration*[ORdP94]. Le principe de fonctionnement d'une telle couche, qui est décrit dans la figure 1.2, est simple : le comportement du système est observé grâce au service d'audit. Les informations collectées par le service d'audit sont utilisées pour prendre les décisions d'administration qui vont se traduire par l'émission d'un certain nombre de commandes de contrôle permettant de reconfigurer le système afin de garder un fonctionnement correct. Notons toutefois que notre contribution ne concerne pas le processus de prise de décision, qui est en lui même un sujet de recherche très vaste.

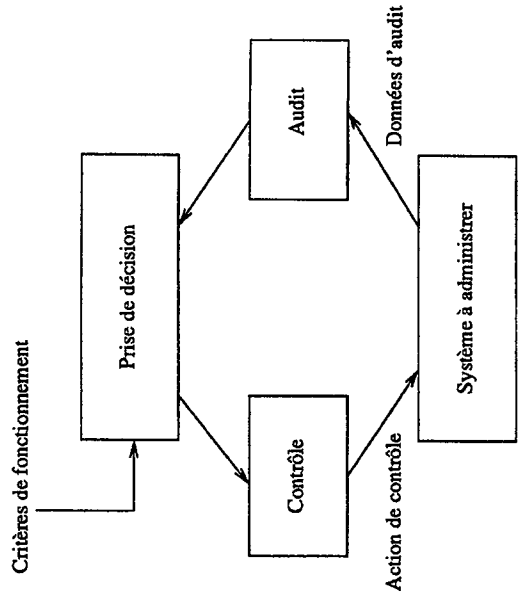

FIG. 1.2 – *Couche minimale d'administration*

La couche minimale d'administration est réalisée au dessus de OODE [Ca94], une plate-forme à objets dont le but est de faciliter le développement d'applications réparties et coopératives utilisant des objets persistants. OODE offre au programmeur le langage OC++ ; il s'agit d'une extension du C++, qui inclut les fonctions de distribution et de persistance. Nous précisons dans la suite notre contribution au domaine de l'administration des systèmes répartis.

5 Contribution apportée

Les principaux apports de notre travail sont les suivants :

- Une étude approfondie des fonctions des langages utilisés pour décrire l'ensemble des ressources composant le système. Cette étude nous a permis de proposer une solution basée sur un langage de haut niveau orienté objet pour fournir une représentation à base d'objets persistants, appelés aussi *objets administrés* (OA). Cette solution est basée sur le langage OC++ de la plateforme que nous avons utilisée, auquel nous avons ajouté plusieurs extensions spécifiques à l'administration. Ces extensions permettent la spécification des relations entre les différents OA et celle des événements qui pourront être émis pour signaler un changement de l'état d'un OA.

- La définition et la réalisation d'un arbre d'héritage complet, qui comporte toutes les classes d'OA nécessaires pour le fonctionnement correcte d'un système réparti. Chacune de ces classes possède une *interface opérationnelle* qui lui est propre (par opposition à l'interface d'administration qui elle est uniforme). Cette interface permet d'accéder aux services offerts par cette ressource. Il s'agit d'une représentation *unique et unifiée* des entités administrées. Ce qui permet d'aborder les problèmes d'administration de chaque ressource au moment où l'on définit son interface opérationnelle, permettant ainsi une meilleure intégration des tâches d'administration au sein du système[ORdP96b].

- Une étude approfondie des problèmes liés à la gestion des changements qui peuvent se produire pendant la durée de vie d'un système. Nous nous sommes consacrés à la classification des changements selon leur nature et le grain des entités sur lesquels ils portent. Nous avons également étudié les critères qui peuvent être pris en considération lors de la conception d'un système de gestion de changement. Cette étude se termine par l'analyse des algorithmes existants.

- Un algorithme qui permet une gestion efficace des modifications de configuration. L'algorithme que nous proposons vise à éviter le ralentissement constaté dans le fonctionnement des algorithmes existants. Il permet au système d'accepter les changements de type structurel (ajout, retrait, connexion et déconnexion des éléments). Le système peut alors évoluer d'une configuration à une autre sans mettre en cause la cohérence des applications qui s'exécutent dessus.

- L'objectif d'efficacité que nous nous étions fixé est atteint dans la mesure où le ralentissement du système causé par les opérations modifiant sa configuration est très inférieur à celui observé en utilisant d'autres algorithmes. De plus, l'algorithme s'adresse à une classe plus large d'applications réparties que celle considérée par les algorithmes existants, et ne pose pas de contraintes particulières aux programmeurs [ORdP96a, ORdP96b].

6 Plan

Ce document se décompose en deux parties et une conclusion. La première partie traite du problème de *spécification de la configuration*; elle est constituée des chapitres 2 et 3. La seconde traite du problème de *l'évolution dynamique de la configuration*; elle est constituée des deux chapitres 4 et 5. Chacune de ces parties couvre une fonction distincte de la gestion de la configuration telle qu'elle a été décrite dans la section précédente. C'est la spécificité du problème de l'évolution dynamique de la configuration qui nous a poussé à opérer cette division en deux parties séparées. Ce dernier domaine n'a jamais été, à notre connaissance, abordé dans les systèmes d'administration. Les personnes qui s'intéressent à l'évolution dynamique de la configuration appartiennent plutôt à la communauté du «génie logiciel» qu'à celle des concepteurs de systèmes d'exploitation. Les problèmes posés par la gestion des modifications sont très différents de ceux posés par la spécification de la configuration. Ainsi avons nous craint de créer de confusion si nous intégrions les deux parties ensembles.

Le chapitre 2 met l'accent sur le problème de spécification de la configuration. On représente l'ensemble des fonctions que le langage de spécification doit offrir à travers un état de l'art des langages utilisés dans d'autres modèles d'administration.

Le chapitre 3 décrit la solution adoptée pour le langage de spécification et la mise en œuvre de cette solution. On met l'accent sur les extensions nécessaires à notre langage de départ (OC++ [San95]) pour pouvoir répondre aux exigences introduites dans le chapitre 2.

Le chapitre 4 présente le problème de la gestion *dynamique* de la configuration (par opposition à la spécification statique qui est décrite dans les chapitres 2 et 3). Après une présentation des différents problèmes et contraintes concernant la gestion des changements, on offre une classification des différentes approches basée sur le degré de changement que les différents modèles peuvent accepter.

Le chapitre 5 présente notre algorithme de gestion dynamique de la configuration. L'algorithme cherche à réaliser un compromis difficile entre les problèmes de préservation de cohérence, le support d'une classe importante d'applications, les dégradations des performances et la transparence pour les applications qui s'exécutent sur le système. Un exemple d'illustration permet de bien distinguer l'originalité de notre algorithme de gestion dynamique de la configuration par rapport aux autres solutions existantes. L'exemple nous a également servi de plate-forme d'essais qui a permis de montrer par des mesures les bonnes performances de cet algorithme.

Le chapitre 6 conclut le rapport.

2

Spécification de la configuration

1 Introduction

Dans le chapitre précédent, nous avons parlé du besoin de disposer d'un langage de spécification de la configuration. Dans ce chapitre, nous analysons de plus près les différentes fonctions que doit fournir un tel langage en justifiant l'intérêt de ces fonctions.

Pour effectuer cette analyse, nous allons considérer les langages utilisés dans plusieurs systèmes d'administration : Moira, le service d'administration du projet Athena [Ra88] ; SNMP [Sta93], le protocole d'administration des réseaux IP ; CMIP, le service d'administration défini dans le modèle OSI [Kle88] [Sta93].

Le langage doit permettre de décrire un système contenant un nombre important de ressources interconnectées, ainsi que les relations qui existent entre ces ressources et les interactions diverses qui peuvent avoir lieu entre elles. On trouve ici une similitude importante avec les objectifs des MIL (Module Interconnection Languages) ; ces langages ont été élaborés pour résoudre les problèmes posés par la spécification de gros logiciels contenant un grand nombre de modules interconnectés. C'est à cause de cette similitude que nous nous sommes aussi intéressés aux MIL et aux techniques qu'ils utilisent.

Les trois familles de fonctions que doit permettre d'exprimer le langage que nous souhaitons utiliser sont : la modélisation du système, la description de son interface, et la possibilité de vérifier la cohérence du système [ORdP96b, SD94]. La suite de ce chapitre est consacrée à la définition de ces fonction et à la justification de leur intérêt, ainsi qu'à la présentation de leur mise en œuvre dans les systèmes d'administration que nous analysons.

2 La modélisation du système

La modélisation du système est généralement définie comme l'utilisation de notations formelles ou informelles pour obtenir une représentation abstraite du système. Cette représentation est utilisée par les différentes applications pour communiquer avec les ressources représentées. Il s'agit d'abord de modéliser les ressources et les éléments dont le système est composé (stations, imprimantes, utilisateurs, etc.). Il

faut ensuite modéliser les relations qui existent entres ces éléments. Ce processus de modélisation comporte donc deux aspects :

Spécification et création d'instances de types abstraits. Le but du processus de modélisation est d'associer à chacune des entités gérées une instance d'un type abstrait. Ces types peuvent correspondre à des enregistrements d'une base de données comme dans Moira, à de simples variables comme dans SNMP ou à des classes d'objets comme dans CMIP. Nous analysons dans la suite le degré de sophistication et de puissance de ces types.

Spécification des relations. Pour illustrer ce besoin, Dean et al [Dea93, Da92] donnent l'exemple du problème de l'installation des logiciels dans une organisation regroupant un grand nombre d'utilisateurs et plusieurs administrateurs. Dans une telle organisations, on peut s'intéresser aux relations suivantes :

- la responsabilité qui lie un programme à son ou ses administrateurs. Les usagers ont besoin de connaître cette relation pour éviter de s'adresser au mauvais administrateur lorsqu'ils ont des problèmes.
- la dépendance qui lie un programme avec l'ensemble des ressources qu'il utilise. Les administrateurs ont besoin de connaître cette relation pour éviter les catastrophes lors d'un changement de version.

On peut multiplier les exemples démontrant la nécessité de pouvoir spécifier explicitement les relations qui existent entre les différentes entités. Les mécanismes permettant cette spécification constituent un des éléments importants de différenciation entre les modèles que nous analysons.

De plus, le processus de modélisation doit tenir compte de deux facteurs :

La complexité. Le grand nombre d'entités à administrer et la grande diversité de leurs types respectifs compliquent la tâche de modélisation. On retrouve ici le problème du facteur d'échelle dont nous avons parlé dans l'introduction. Nous présentons dans la suite des techniques qui permettent de réduire la complexité.

Les types de base disponibles. Les types de base (entiers, chaînes de caractères, pointeurs, etc.) sont les blocs qui sont utilisés pour la construction des types abstraits qui représentent les ressource. La définition des types de base fournis par les différents modèles ainsi que les mécanismes de définition de nouveaux types ont des répercussions sur le mécanisme de modélisation. Par exemple, un langage ne permettant pas l'utilisation des pointeurs ou des références des objets comme type de base rend la mise en œuvre des relations extrêmement difficile.

2.1 Modélisation dans Moira

Une des premières tentatives de définition d'un système d'administration réparti a vu le jour dans le cadre du projet Athena sous le nom de Moira [Ra88]. Il s'agit

d'un serveur avec son protocole d'interface. Le serveur gère une base de données relationnelle qui contient les informations concernant l'ensemble des ressources du système. Il s'agit donc d'un gestionnaire centralisé de la MIB. Les entités administrées sont mises en œuvre sous forme d'enregistrements (lignes) d'une table dans la base de données. La base contient les informations décrivant les entités suivantes : Les utilisateurs, les machines, les serveurs RVD [1], les clusters (association machine $\leftrightarrow$ imprimante et serveur RVD par défaut), les ports des services, les serveurs NFS, les imprimantes, les gestionnaires des boites aux lettres, les listes de diffusion, les alias, et finalement les serveurs qui ont besoin d'être mis à jour lors de la modification des données de la base. Les informations stockées dans le serveur permettent de générer et de mettre à jour les fichiers de configuration nécessaires au fonctionnement des serveurs du système (serveur de nom, serveur NFS, etc.).

Si l'on prend l'exemple des informations stockées sur les machines, on trouve les informations suivantes : nom, type, modèle, statut, numéro de série, et la version du système d'exploitation.

Le fait que Moira soit réalisé en utilisant le SGBD relationnel Ingres, aurait pu permettre de définir des relations entre les différentes entités, puisque c'est l'une des fonctions essentielles du langage de spécification utilisé (SQL). Cette possibilité n'est pourtant pas exploitée, et il n'existe aucune relation définie explicitement entre les différents types d'entités que nous avons citées ci-dessus.

Les choix concernant la modélisation et la représentation des entités administrées dans Moira constituent une première évolution permettant d'échapper aux fichiers de configuration utilisés par un système centralisé comme UNIX. Il s'agit d'une évolution dans le sens où le système utilise des champs typés pour stocker les informations ce qui permet un meilleur accès à ces informations et un meilleur contrôle de leur intégrité. Un autre aspect intéressant est celui du stockage efficace permettant un accès plus rapide aux informations ; cet aspect ne fait pas vraiment partie des points clés du processus de modélisation, mais constitue quand même un avantage non négligeable. On peut noter toutefois plusieurs inconvénients de Moira :

1. La création de nouveaux types d'entités administrées, qui correspond à la définition et à la création d'une nouvelle table de la base, se fait manuellement en mode interactif (à ne pas confondre avec l'ajout ou le retrait d'éléments définis, qui correspondent à la modification d'une table déjà créée), ce qui constitue une perte importante de souplesse et restreint les possibilités d'introduction de nouveaux types d'éléments au sein du système.

2. La modélisation des entités administrées par les tuples d'une base de données relationnelle restreint sévèrement les types de données que l'on peut utiliser. Il interdit en particulier l'utilisation de types structurés. Cette restriction complique la tâche des programmes clients qui ont besoin d'exécuter des opérations sophistiquées.

3. La définition de relations entre les différentes entités administrées n'est pas possible et ceci malgré les possibilités offertes par le SGBD. Cette sous-exploitation

1. RVD : Remote Virtual Disk, c'est un serveur qui gère l'accès aux SGF's partagés en mode lecture uniquement

est essentiellement due à la fonction même de Moira, qui a été conçu comme un service de gestion d'un réseau de systèmes indépendants, où le partage des ressources et les interactions entre les différents éléments du réseau sont limités. C'est pour cette raison que les concepteurs n'ont pas trouvé d'intérêt à la description de relations explicites entre les différents éléments.

2.2 Modélisation dans SNMP

Dans SNMP [Sta93], les entités administrées sont appelées *objets* (à ne pas confondre avec la notion classique des langages orientés objets). Ces objets sont décrits en utilisant la notation ASN.1 [Kal91], un langage formel spécifié par le CCITT et l'ISO. Chaque site maintient sa propre MIB, qui contient l'ensemble des objets représentant les ressources résidant sur ce site.

L'architecture de SNMP décrite dans la figure 2.1 est composée des éléments suivants :

- Une station d'administration qui contient les applications administratives (traitement d'erreurs, gestion de comptabilité, etc.).
- Sur chacune des stations du réseau, un agent d'administration responsable de répondre aux messages envoyés par la station d'administration.
- Une base de données administratives (MIB) [MR90] par station administrée (donc par agent), elle contient l'ensemble des objets représentant les entités administrées qui résident sur cette station.
- Un protocole de communication entre la station d'administration et les agents, c'est le protocole SNMP [SFDC90].

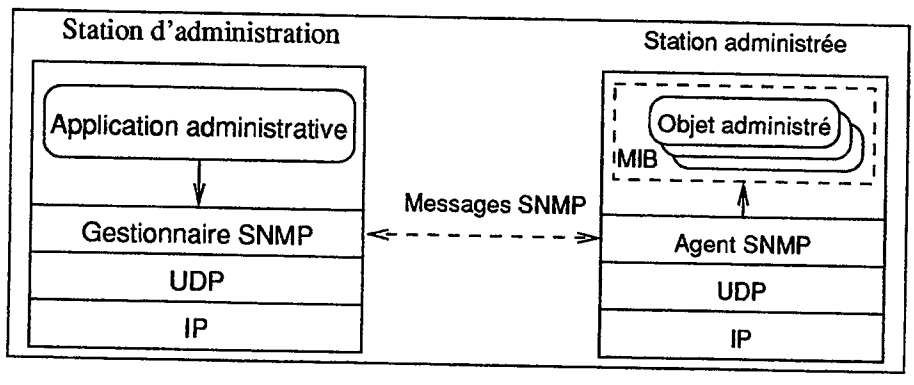

FIG. 2.1 – *Architecture de SNMP*

La MIB de SNMP est organisée sous forme d'arbre, et chaque objet possède un identificateur de type OBJECT IDENTIFIER qui détermine sa place dans cet arbre ; cet identificateur est unique pour un objet et pour une machine donnée. La valeur de l'identificateur est constituée d'une séquence d'entiers ; cette valeur est construite

d'une façon hiérarchique : l'identificateur d'un objet est obtenu en ajoutant un entier à l'identificateur de l'objet supérieur dans la hiérarchie de la MIB.

Les premiers niveaux de la hiérarchie de la MIB sont décrits dans la figure 2.2. Les objets utilisés pour administrer une machine connectée sur l'Internet appartiennent au sous-arbre identifié par le nœud *mgmt*. Ce nœud possède l'identificateur suivant :

mgmt OBJECT IDENTIFIER : := iso(1) org(3) dod(6) internet(1) 2.

Tous les objets du sous arbre commençant à ce nœud ont donc le préfixe commun : 1.3.6.1.2

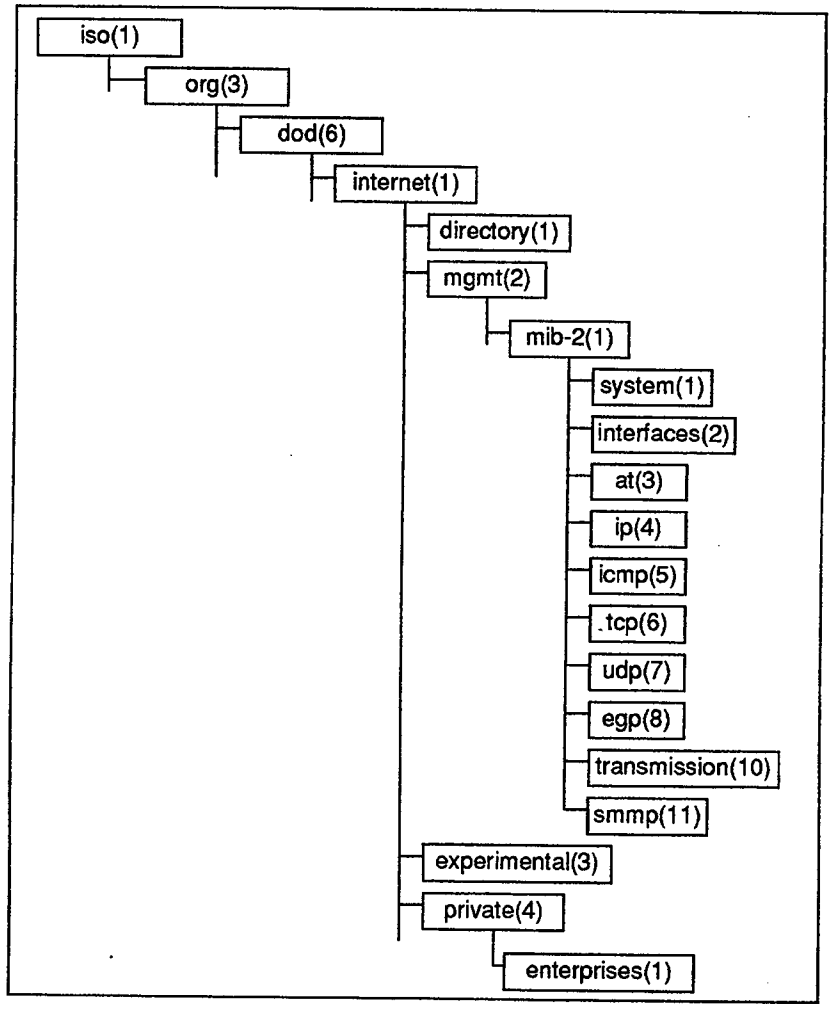

FIG. 2.2 – *Structure de la MIB dans SNMP*

La MIB est divisée en un ensemble de *groupes* qui représentent un regroupement des ressources du système par catégorie. Ces groupes sont les suivants :

System : contient les variables qui décrivent le système en général (nom, lieu, système d'exploitation, type de machine, etc.).

Interfaces : contient des informations sur l'ensemble des interfaces qui lient la machine aux sous-réseaux.

IP : contient des informations concernant la réalisation et le fonctionnement du protocole IP sur la machine.

ICMP, TCP, UDP, EGP : ces groupes contiennent des informations concernant la réalisation et le fonctionnement des protocoles correspondants sur la machine.

Transmission : ce groupe contient des informations sur le médium de transmission utilisé par chacune des interfaces du système.

Chaque objet de la MIB possède un *type* et une *valeur*. Le type de l'objet définit une classe particulière d'objets. Une instance d'un objet est une instantiation d'un type d'objets à laquelle on a associé une valeur spécifique. Un type ne peut pas avoir plus d'une seule instance, ceci est dû au fait que l'arbre utilisé pour stocker les objets de la MIB est utilisé aussi pour définir les types de ces objets.

Pour définir les types des objets de la MIB, on exploite le fait que ASN.1 dispose d'un ensemble de types de base et d'une grammaire permettant de définir de nouveaux types à partir de ces types de base.

Afin de faciliter la construction des MIB et l'interoperabilité entre les clients et le serveur SNMP, les concepteurs du protocole fournissent un ensemble de macro fonctions qui encapsulent le schéma général de définition des nouveaux types. Ce schéma est composé des étapes suivantes :

Définition de macro : Définition des instances légales d'une macro, et spécification de la syntaxe permettant de définir les types, qui sont des instance de cette macro.

Instance de macro : Une instance est générée à partir d'une macro en substituant les paramètres dans la définition formelle par leur valeur, ce qui permet de générer une définition de type.

Une valeur d'une instance : représente une entité spécifique ayant une valeur spécifique.

La macro OBJECT-TYPE utilisée pour définir les types d'objets de la première MIB de SNMP (mib-1) est décrite dans la figure 2.3. Les mots clefs importants dans cette définition sont les suivants :

- SYNTAX : La syntaxe abstraite du type d'objet. Cette syntaxe doit être construite en utilisant les types de base autorisés qui sont décrits dans la même figure.

- ACCESS : Définit les options d'accès à une instance d'un objet. Les valeurs possibles sont *read-only, write-only, read-write,* et *not-accessible.*

- STATUS : Indique le niveau d'importance d'un objet. Les valeurs possibles sont *mandatory, optional,* et *obsolete* qui signifie qu'une machine n'a plus besoin d'avoir une instance de cet objet dans sa MIB.

```
--definition des types d'objets
OBJECT-TYPE MACRO ::=
BEGIN
        TYPE NOTATION ::=  "Syntax" type(TYPE ObjectSyntax)
                           "ACCESS" Access
                           "STATUS" Status
        VALUE NOTATION ::= value(VALUE ObjectName)
        Access ::= "readonly"|"write-only"|"read-write"|"not-accessible"
        Status  ::= "mandatory"|"optional"|"obsolete"|deprecated"
END
ObjectName ::=OBJECT IDENTIFIER  -- Noms des objets de la MIB
--Syntax des Objets de la MIB
ObjectSyntax ::= CHOICE {simple SimpleSyntax,
                         applicationwide ApplicationSyntax}
SimpleSyntax ::= CHOICE {number INTEGER,
                         string OCTET STRING,
                         object OBJECT IDENTIFIER,
                         empty NULL}
ApplicationSyntax ::= CHOICE  {address NetworkAddress,
                               counter Counter,
                               gauge Gauge,
                               ticks TimeTicks,
                               arbitrary Opaque}
-- Les Types d'application
NetworkAddress ::= CHOICE {internet IpAddress}
IpAddress ::=[APPLICATION 0]
                    IMPLICIT OCTET STRING (SIZE (4))
Counter ::= [APPLICATION 1] IMPLICIT INTEGER (0..4294967295)
Gauge   ::= [APPLICATION 2] IMPLICIT INTEGER (0..4294967295)
TimeTicks  ::= [APPLICATION 3] IMPLICIT INTEGER (0..4294967295)
Opaque ::= [APPLICATION 3] IMPLICIT INTEGER (0..4294967295)
```

FIG. 2.3 – *Définition des types des objets de la MIB*

- VALUE NOTATION : indique le nom utilisé pour accéder à cet objet via SNMP. Il s'agit d'un identificateur ASN.1 de type OBJECT IDENTIFIER que nous avons déjà introduit.

Comme la figure 2.3 le montre, les types de base que l'on peut utiliser pour décrire la syntaxe des types de objets administrés dans la version actuelle de la MIB (mib-2) sont très limités, ce qui permet de simplifier le protocole. Trois classes de types peuvent être utilisées : Les types primitifs (Universal types), les types composés (Application types), et les tableaux. Aucun autre type de base ne peut être utilisé pour construire sans passer par un nouveau RFC.

2.2.1 Les types primitifs

Les types primitifs (Universal types) qui ont été retenus parmi ceux proposés par ASN.1 sont : les entiers (INTEGER), les chaînes d'octets (OCTET STRING), les identificateurs d'objets (OBJECT IDENTIFIER), et les séquence (SEQUENCE, SEQUENCE OF).

2.2.2 Les types composés

Les types composés (Application types) définis pour le moment sont les suivants :

- *NetworkAddress :* ce type est construit en utilisant le constructeur CHOICE, pour permettre la sélection d'un format d'adresse parmi ceux des différents protocoles. Pour le moment, le seul protocole reconnu est IP.

- *IpAddress :* c'est une adresse sur 4 octets qui utilise le format standard de IP.

- *Counter :* un compteur est un entier non-négatif que l'on peut augmenter mais pas diminuer. Une valeur maximale de $2^{32} - 1$ est spécifiée ; quand le compteur atteint cette valeur, il est remis à zéro.

- *Gauge :* un indicateur est un entier non-négatif dont la valeur peut augmenter ou diminuer, avec une valeur maximale de $2^{32} - 1$. Si cette valeur maximale est atteinte, l'indicateur garde cette valeur jusqu'à ce qu'il soit remis à zéro.

- *TimeTicks :* le compteur de battements d'horloge est un entier non-négatif qui donne le temps avec une précision d'un millième de seconde.

- *Opaque :* il s'agit d'une séquence d'octets, utilisée pour transmettre des données codées.

2.2.3 Les tableaux

Les tableaux constituent le seul type structuré utilisé dans SNMP. Ils n'ont été introduits que dans la deuxième version de la MIB. Il s'agit de tableaux unidimensionnels (pas de tableaux de tableaux).

Si l'on veut reprendre l'exemple de la représentation d'une machine, on voit que dans SNMP on utilise l'ensemble de variables appartenant au groupe *system.* La figure 2.4 illustre cette description. On voit dans cet exemple l'un des inconvénients majeurs de SNMP. Pour définir une nouvelle variable, il faut systématiquement définir un nouveau type (sauf s'il s'agit d'un élément d'un tableau). Le fait que l'on ne puisse pas utiliser de types structurés nous oblige à utiliser la composition hiérarchique des types pour représenter des entités plus complexes sous forme de groupes de variables. Mais on est obligé de traiter chacun des types regroupés comme un type séparé et donc fournir une définition complète de type. On doit donc répéter pour tous les types qu'ils sont accessibles en lecture uniquement, que leur présence dans la MIB est obligatoire, et leur attribuer un nouvel identificateur, alors que l'on devrait pouvoir spécifier cette propriété au niveau du groupe *system.*

On constate que SNMP offre une représentation simple des entités, basée sur un ensemble limité de types de base, tous simples à l'exception d'une utilisation limitée des vecteurs, et sur un mécanisme de macro fonctions permettant la définition de nouveaux types. L'ensemble des types ainsi construits restent simples et surtout d'un niveau d'abstraction assez bas.

```
sysDescr OBJECT-TYPE                sysObjectID OBJECT-TYPE          sysUpTime OBJECT-TYPE
SYNTAX DisplayString (SIZE (0..255)) SYNTAX OBJECT IDENTIFIER        SYNTAX TimeTicks
ACCESS read-only                    ACCESS read-only                 ACCESS read-only
STATUS mandatory                    STATUS mandatory                 STATUS mandatory
::= { system 1 }                    ::= { system 2 }                 ::= { system 3 }

sysContact OBJECT-TYPE              sysName OBJECT-TYPE
SYNTAX DisplayString (SIZE (0..255)) SYNTAX DisplayString (SIZE (0..255))
ACCESS read-only                    ACCESS read-only
STATUS mandatory                    STATUS mandatory
::= { system 4 }                    ::= { system 5 }

sysLocation OBJECT-TYPE             sysServices OBJECT-TYPE
SYNTAX DisplayString (SIZE (0..255)) SYNTAX INTEGER (0..127)
ACCESS read-only                    ACCESS read-only
STATUS mandatory                    STATUS mandatory
::= { system 6 }                    ::= { system 7 }
```

FIG. 2.4 – *Description d'une machine dans SNMP*

On peut reprocher à SNMP les problèmes suivants :

- La confusion entre les notions de *type* et de *variable*. Cette confusion est due au fait que pour définir une nouvelle variable de la MIB, il faut systématiquement définir un nouveau type, la syntaxe devenant alors trop lourde. Les spécifications des MIB de SNMP sont très longues et assez difficile à lire.

- Le niveau bas de représentation, essentiellement dû à la limitation de l'ensemble des types de bases que l'on peut utiliser. Il est vrai que la simplicité de réalisation de la MIB est un but fondamental pour les concepteurs du protocole, mais le manque d'une représentation de haut niveau réduit l'intérêt des informations présentes dans la MIB.

- L'absence totale de représentation des relations entre les entités. Il n'existe aucun moyen de déclarer des relations, à l'exception de la relation de contenance induite par le groupement hiérarchique des types.

2.3 Les modèles orientés objet

Les dernières années ont vu l'apparition des systèmes d'administration basés sur l'approche orientée objet. Qu'il s'agisse de Tobias [Abd91], de la DME [Fou92], ou de CMIP [Sta93], ces trois systèmes sont tous basés sur une idée commune : la représentation des entités administrées par des objets. Vu la similitude entre les trois modèles, nous avons décidé de nous concentrer sur le plus répandu parmi eux, il s'agit de CMIP, le protocole d'administration du modèle OSI.

Dans CMIP, les ressources du système sont représentées par des *objets administrés*. Les concepts de cette représentation sont dérivés de ceux de la méthode de conception orientée objet. Notons que les spécifications ne précisent pas un langage

particulier de réalisation ; elles ne stipulent même pas que les MIB doivent être réalisées en utilisant un langage orienté objet. La notation utilisée par ce modèle pour spécifier les objets administrés s'appelle GDMO (Guidelines for the Definition of Managed Objects) [ISO92]. Chaque site du système contient sa propre MIB, comme c'est le cas dans SNMP. Les applications administratives n'ont pas d'accès direct à la MIB et doivent passer par un processus local appelé *agent*. Si l'application qui a besoin d'accéder à une MIB ne s'exécute pas sur la même station que cette MIB, elle doit envoyer un message à l'agent qui en est responsable.

Le document qui décrit l'organisation et la représentation des données administratives et de la MIB introduit des notions qui ne font pas partie des notions classiques des langages orientés objet. Il s'agit d'un ensemble d'extensions introduites dans le but d'adapter ces langages à l'administration. Ces notions sont : les attributs, les notifications, et les paquetages. Nous allons dans la suite définir ces notions à travers des exemples.

2.3.1 Les attributs d'un objet

Un attribut est une variable faisant partie des données encapsulées dans l'objet. C'est une extension de la notion classique d'attributs dans les langages orientés objets car on peut associer à ces attributs des mots clefs permettant d'effectuer des opérations d'adressage associatif. Chaque attribut correspond à une propriété de la ressource que l'objet représente, et c'est en scrutant ces attributs que l'ont peut réaliser les fonctions d'audit. L'attribut peut être de type entier, réel, booléen, chaîne de caractères, ou un type structuré construit à partir de ces types de base. L'exemple montré dans la figure 2.5 permet d'illustrer la définition des attributs.

```
eventTime ATTRIBUTE
  WITH ATTRIBUTE SYNTAX Attribute-ASN1Module.EvenTime ;
  MATCHES FOR EQUALITY, ORDERING ;
  BEHAVIOR timeOrdering
  REGISTERED AS smi2Attribute ID 13 ;
timeOrdering BEHAVIOR DEFINED AS
  "The year, month, day, hour, minute, and seconds fields
  are ompared in order to determine whether the specified
  value is greater or less than the value of the attribute" ;
```

FIG. 2.5 – *Exemple de la définition des attributs dans GDMO*

Le constructeur WITH ATTRIBUTE SYNTAX permet de spécifier les attributs non dérivés d'un autre attribut déjà défini. Le constructeur alternatif DERIVED FROM permet de spécifier les attributs dérivés, et dans ce cas l'attribut dérivé hérite des paramètres de filtrage et du rôle de l'attribut père.

Le mot clef MATCHES FOR définit l'ensemble des tests de filtrage que l'on peut effectuer sur la valeur de l'attribut. Les objets qui passent un test seront sélectionnés pour l'exécution d'une opération. Il s'agit là d'un mécanisme d'addressage

associatif très puissant. Les mots clefs définissant les tests possibles sont: EQUALITY pour un test d'égalité, ORDERING pour une comparaison, SUBSTRINGS pour vérifier la présence d'une sous-chaîne de caractère, SET-COMPARISON pour vérifier l'inclusion dans un ensemble, et SET-INTERSECTION pour vérifier si l'intersection entre un attribut de type ensemble et un ensemble donné n'est pas vide. Le mot clef BEHAVIOR, définit le rôle de l'attribut sous forme d'une description textuelle informelle. Finalement, on peut donner un identificateur ASN.1 à l'attribut avec le mot clef REGISTERED AS, la structure des identificateurs et de l'arbre d'enregistrement sera décrite plus tard. Les concepteurs du protocole offrent un ensemble assez large de types d'attributs prédéfinis, il est donc peu probable que l'on ait besoin de définir de nouveaux types d'attributs.

2.3.2 Les notifications

les notifications sont des messages émis par un objet suite à certains changements dans sont état. Ils constituent le mécanisme de base pour l'audit dans les service d'administration de l'OSI. La figure 2.6 illustre la spécification d'une notification.

```
communicationsAlarm NOTIFICATION
  BEHAVIOR communicationsAlarmBehavior ;
  WITH INFORMATION SYNTAX Notification-ASN1Module.AlarmInfo
    AND ATTRIBUTE IDS
    probableCause probableCause,
    psecificProblems specificProblems,
    perceivedSeverity perceivedSeverity,
    backUpStatus backedUpStatus,
    backUpObject backUpObject,
    trendIndication trendIndication,
    thresholdInfo thresholdInfo,
    notificationIdentifier notificationIdentifier,
    correlatedNotifications correlatedNotifications,
    stateChangeDefinition stateChangeDefinition,
    monitoredAttributes monitoredAttributes,
    proposedRepairActions proposedRepairActions,
    additionalText additionalText,
    additionalInformation additionalInformation ;
  REGISTERED AS smi2Notification 1 ;
communicationsAlarmBehavior
  BEHAVIOR DEFINED AS "This notification type is used to
  report when the object detects a communications error."
```

FIG. 2.6 – *Spécification d'une notification dans GDMO*

D'abord, le mot clef BEHAVIOR permet de décrire (toujours d'une façon informelle) le rôle de la notification, les données qui seront spécifiées avec, les résultats générés, et leur signification. Ensuite, le constructeur WITH INFORMATION

SYNTAX identifie le type ASN.1 correspondant, suivi par une suite d'attributs, chacun appartenant à l'un des types d'attributs définis (notons que les noms des attributs sont les mêmes que de ceux de leurs types), et finalement l'identificateur de la notification.

2.3.3 Les paquetages

Un paquetage est un ensemble d'attributs, de méthodes, et de notifications optionnels qui peuvent être présents dans un objet. Un paquetage exprime la présence d'une certaine option dans la ressource sous-jacente. Les opérations sur les éléments d'un paquetage sont réalisées à travers l'objet qui le contient, sans aucune référence au paquetage lui même. Les paquetages d'une classe sont hérités par ses sous-classes. Un exemple de la définition d'un paquetage est montré dans la figure 2.7.

```
hostInfo PACKAGE
  BEHAVIOR
  hostBehavior BEHAVIOR
  DEFINED AS Ce paquetage contient une partie des informations
  nécessaires pour représenter un site du réseau.
  ATTRIBUTES
    stationName GET-REPLACE,
    hostAddress GET ;
  ACTIONS
    boot,
    halt ;
  NOTIFICATION
    hostDown,
    hostUp ;
  REGISTERED AS smiRien 2 ;
```

FIG. 2.7 – *Spécification d'un paquetage dans GDMO*

Le constructeur BEHAVIOR permet de documenter d'une façon informelle (comme pour les attributs) le comportement du paquetage. Le constructeur ATTRIBUTES fournit une liste des attributs qui font partie de ce paquetage. A chaque attribut sont associées des propriétés optionnelles :

- REPLACE-WITH-DEFAULT : L'attribut a une valeur par défaut que l'on peut forcer grâce à une opération administrative.
- DEFAULT VALUE : Spécifie la valeur par défaut de l'attribut.
- INITIAL VALUE : La valeur initiale de l'attribut.
- PERMITTED VALUES : Une restriction sur la valeur de l'attribut.
- REQUIRED VALUES : L'ensemble de valeur que l'attribut peut prendre.

- GET / REPLACE / GET-REPLACE : Les opérations qui peuvent être effectuées sur cet attribut dans le cas où il s'agit d'un attribut atomique.

- ADD / REMOVE / ADD-REMOVE : Les opérations qui peuvent être effectuées sur cet attribut dans le cas où il s'agit d'un attribut de type ensemble.

Après les attributs viennent les *actions* que l'on peut effectuer sur l'objet qui contient ce paquetage. C'est l'équivalent des méthodes dans les langages orientés objet, les actions peuvent avoir une liste de paramètres. Ensuite on a la liste de notifications qui peuvent être générées ; et finalement l'identificateur du paquetage.

La définition d'un objet administré se décompose en cinq parties ; nous avons déjà vu trois d'entre elles : les attributs d'un objet, ses paquetages, et les notifications qu'il pourrait générer ; il nous reste à voir l'allocation de son identificateur et la spécification de sa classe.

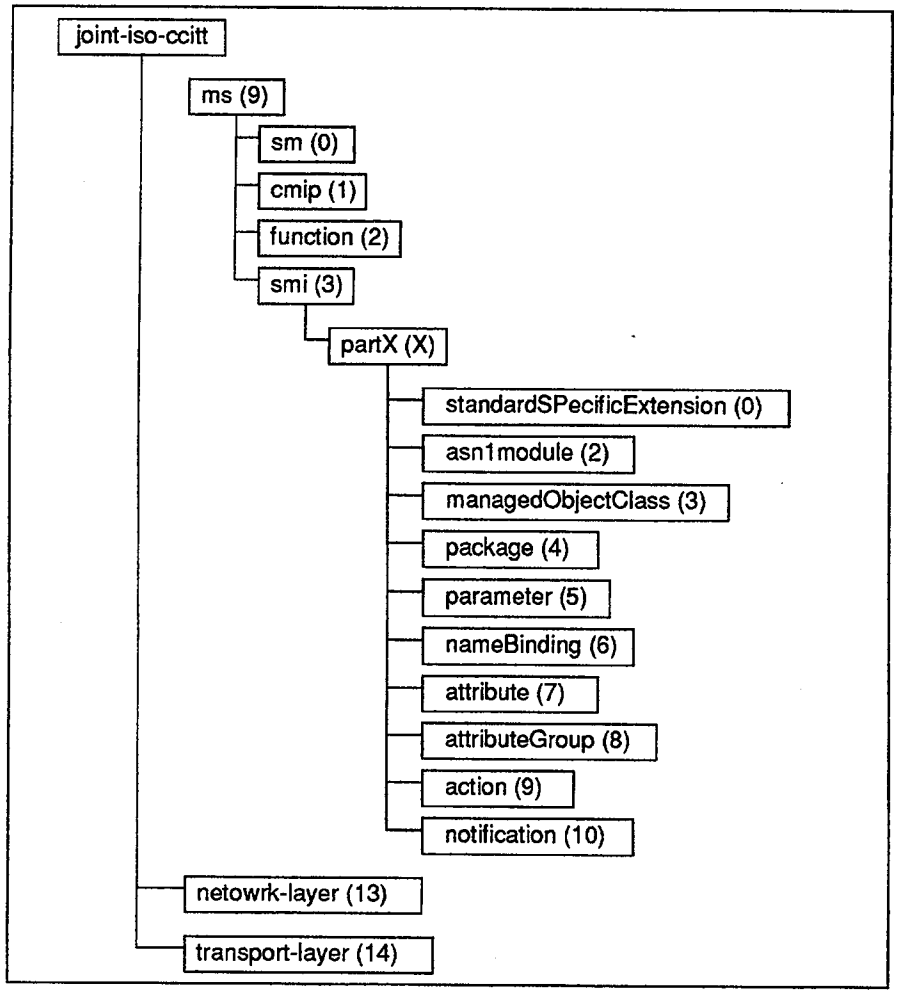

FIG. 2.8 – *L'arbre d'allocation d'identificateurs du modèle OSI*

2.3.4 Allocation d'identificateur

Chaque objet, notification, attribut, paquetage, ou tout autre élément qui fait partie du système OSI possède un identificateur unique défini par sa position dans l'arbre d'identificateurs ASN.1. La figure 2.8 décrit la partie de cet arbre concernant les standards d'administration des systèmes de l'OSI. C'est dans le sous arbre défini par le nœud partX que se trouvent les définitions des attributs, des paquetages, des actions et des classes d'objets.

2.3.5 Spécification de la classe

Pour définir une classe, on a besoin de donner sa classe mère, ainsi que la liste des paquetages spécifiques à cette classe. Tous les attributs, les notifications et les actions qui caractérisent la classe doivent faire partie d'un paquetage. La figure 2.9 permet d'illustrer la spécification d'une nouvelle classe.

```
accountringMeterControlObject MANAGED OBJECT CLASS
  DERIVED FROM "IOSO10165-2" :top
  CHARACTERIZED BY meter-control-info PACKAGE
    BEHAVIOR
    AMControlBehavior BEHAVIOR
    DEFINED AS "When an instance of ..........";
    ATTRIBUTES
    control-object-id GET,
    units-of-usage GET,
    recording-triggers GET-REPLACE, ADD-REMOVE,
    reporting-triggers GET-REPLACE, ADD-REMOVE ; ;
  CONDIONAL PACKAGES
    accounting-Meter-Actions PACKAGE
      ACTIONS start,
        suspend,
        resume ;
      NOTIFICATIONS accountingStarted,
        accountingSuspended,
        accountingResumed ;
      REGISTERED AS
      joint-iso-ccitt ms(9) function(2) part10(10) package(4) 1 ;
      PRESENT IF "the Specific problems parameter is..."
  REGISTERED AS
      joint-iso-ccitt ms(9) function(2)       ;
```

FIG. 2.9 – *Spécification d'une classe en GDMO*

Le constructeur DERIVED FROM indique la (les) classe(s) dont cette classe est dérivée. Le constructeur CHARACTERIZED BY permet de spécifier un en-

semble de paquetages obligatoires; les paquetages conditionnels sont spécifiés avec le constructeur CONDITIONAL PACKAGES.

2.3.6 Structure de la MIB

La MIB est structurée sous forme *d'arbre de contenance.* Chaque instance d'un objet est contenue dans une et une seule instance d'un autre objet. Chaque agent maintient sa propre MIB qui contient l'ensemble des objets dont il a la charge. Cet arbre est utilisé aussi pour la désignation des objets; chaque objet contient un attribut utilisé comme un nom, le nom symbolique d'un objet est construit récursivement en concaténant la valeur de cet attribut au nom symbolique de l'objet contenant.

Si l'on compare la MIB de CMIP à celle de SNMP, on voit que SNMP utilise un seul arbre pour la désignation et l'enregistrement des objets administrés. C'est dû au fait que les objets de la MIB sont tous de type simple et qu'il n'existe qu'une seule instance par type d'objets. Le fait que CMIP utilise deux arbres séparés permet plus de souplesse concernant l'évolution dynamique de la MIB. On peut ajouter et retirer des instances d'objets à la MIB de l'OSI alors que dans SNMP c'est impossible.

2.3.7 Les relations dans l'OSI

Le modèle de OSI se distingue de ses prédécesseurs par l'importance qu'il donne à la spécification des relations. Le système offre des services pour la création, la vérification et l'audit des relations entre les objets administrés.

Les relations sont définies grâce à des *attributs de relation.* A chaque type de relation est associé un ensemble de types d'attributs. La valeur d'un attribut est l'ensemble des noms des autres objets liés par la relation correspondante à l'objet qui contient l'attribut. Le type de l'attribut détermine le *rôle* de l'objet contenant dans la relation, ainsi que la nature ou le type de cette relation. Par exemple, dans une relation de type client-serveur, le client contient un attribut de type *providerObject* qui représente l'ensemble des serveurs dont il est client. Le serveur contient un attribut de type *userObject* qui représente l'ensemble de ses clients. La liste suivante donne l'ensemble des types de relations reconnues par l'OSI ainsi que les noms des attributs et les rôles correspondants.

Relation de service : c'est une relation non-symétrique entre deux objets. L'objet qui contient l'attribut de type **providerObject** a le rôle du client; le serveur lui contient un attribut du type *userObject.*

Relation entre pairs : c'est une relation de pair à pair entre deux objets, chacun d'eux contient un attribut de type *peer.*

Relation de réserve : c'est une relation asymétrique entre deux objets. Le deuxième objet (rôle du secondaire) contient un attribut de type *primary* et peut servir au remplacement du premier objet (rôle du primaire) qui lui contient un attribut de type *secondary.*

Relation de sauvegarde : il s'agit d'une relation entre deux objets, où le deuxième (objet secondaire) contient un attribut de type *backedUpObject* et sert de copie

de sauvegarde au premier objet (objet primaire) qui lui contient un attribut de type *backUpObject.*

Relation de groupe : c'est une relation entre deux objets où l'un avec l'attribut de type *member* appartient au groupe représenté par l'autre objet propriétaire qui lui a un attribut de type *owner* .

2.3.8 Évaluation de la modélisation dans CMIP

Il est très difficile d'évaluer objectivement le modèle de l'OSI. D'un côté le modèle semble complet, avec un grand ensemble de types prédéfinis d'attributs et un mécanisme permettant la définition des types nécessaires. Les notifications constituent un excellent outil pour la réalisation des tâches d'audit. L'arbre de contenance qui constitue la MIB est assez souple. C'est l'un des rares modèles d'administration qui couvre l'aspect «relations». Mais nous avons pourtant quelques problèmes avec ce modèle :

1. Le modèle est assez compliqué et la syntaxe contient beaucoup d'options, ce qui rend le processus d'apprentissage et de compréhension long et difficile. Par conséquent, les concepteurs des applications éprouvent beaucoup de difficultés à être compatibles avec les standards proposés par CMIP [All94]. C'est l'une des raisons pour lesquelles CMIP ne s'est toujours pas imposé face à SNMP malgré sa richesse en fonctions fournies.

2. Une bonne partie du comportement des entités spécifiées est décrite d'une façon informelle. C'est par exemple le cas des notifications, où l'on ne peut pas spécifier comment et à quel moment elles sont engendrées. Cela laisse à la charge du programmeur qui réalise les classes des objets le soin de vérifier si les spécifications informelles sont bien respectées.

3. Il n'est pas possible de définir de nouvelles relations.

4. Le nom symbolique déduit de l'arbre de contenance est le seul identificateur attaché à l'objet. Or, cet identificateur risque de changer si l'objet migre vers un autre système.

5. La hiérarchie des classes permet à un objet d'une sous-classe d'être utilisé à la place d'un objet de l'une de ses super-classes. Il s'agit d'une forme limitée du polymorphisme appelée *allomorphism.* La vérification de la compatibilité utilise les identificateurs des classes, et les objets doivent maintenir une liste des classes avec lesquelles ils sont compatibles. Pour cela, on a besoin d'un registre central de classes. Cette solution marche bien pour un petit réseau avec un nombre réduit de classes mais pas dans le cas où la réalisation et les spécifications des classes sont répartis sur un grand ensemble de machines.

2.4 Modélisation dans les MIL

Dans les MIL (Module Interconnection Languages), deux techniques sont utilisées pour réduire la complexité due à la diversité :

- La décomposition hiérarchique du système.
- Les raffinements successifs des spécifications par ajout d'informations.

Le Langage SySL [TI89] permet la décomposition hiérarchique du système en plusieurs classes d'éléments. Une *classe* est une famille d'entités qui possèdent des attributs communs. A chaque instance de cette classe on peut associer une structure qui peut être composée de sous structures. L'exemple décrit dans la figure 2.10 montre la description d'une station de travail dans SySL :

```
class WORKSTATION is (SUN_WORKSTATION, dcl-sparx1)
structure WORKSTATION is
    PROCESSOR=>(m68010,ns32016,ms68020),
    KEYBOARD,
    {DISPLAY}+,
    [DISK_SYSTEM],
    MEMORY,
    [NETWORK]
end structure
```

FIG. 2.10 – *Définition des classes dans SySL*

L'exemple décrit une classe WORKSTATION qui possède deux membres, une autre classe SUN_WORKSTATION et une réalisation *dcl-sparx1*. A la classe WORKSTATION on associe une structure qui contient plusieurs champs. On peut associer des attributs particuliers aux champs, par exemple DISPLAY+ signifie qu'un ou plusieurs écrans peuvent être présents ; [NETWORK] signifie que le champ NETWORK est optionnel ; Le champ PROCESSOR peut recevoir uniquement une des trois valeurs spécifiées.

Le langage Emanuel [Dea93] résulte des modifications apportées à SySL afin de faciliter son utilisation dans le domaine de l'administration. En plus de la décomposition hiérarchique du système, Emanuel utilise aussi la technique des raffinements successifs. Les éléments peuvent être spécifiés à deux niveaux ; au premier niveau on décrit les structures de données sous forme de *patrons* (templates) ; et au deuxième niveau les champs de ces patrons sont remplis par les informations nécessaires pour spécifier des éléments particuliers. Il est possible de remplir un sous-ensemble des champs d'un patron pour engendrer un nouveau patron.

L'exemple illustré dans la figure 2.11 montre un patron USER_ACCNT décrivant un utilisateur. Un autre patron SE_ACCNT est dérivée du premier, il hérite de tous ses champs, en y ajoutant des contraintes sur les valeurs de certaines variables. D'une façon informelle, la structure SE_ACCNT décrit l'ensemble des utilisateurs d'un groupe dont l'id est 68 ; tous les membres de ce groupe ont le droit à un quota

```
structure USER_ACCNT is
    Name,
    User_ID,
    Group_ID,
    Quota,
    Filesotre,
    Mount_Partition
end structure
```

```
structure SE_ACCNT : USER_ACCNT is
    Name,
    User_ID,
    Group_ID => 68,
    Quota => "40Meg",
    Filesotre,
    Mount_Partition => "/user/se"
end structure
```

FIG. 2.11 – *Raffinement successif des classes dans Emanuel*

de 40 MO de stockage et leur répertoire maison doit être monté sur la partition «/user/se».

Ce qui est particulièrement intéressant dans Emanuel est la possibilité qu'il fournit de spécifier explicitement les relations entre plusieurs entités. Cette possibilité n'existe pas dans les autres langages de spécification. Les relations peuvent être *descriptives* si elles décrivent le système comme il est ; ou *prescriptives* si elles décrivent une configuration souhaitable du système. L'exemple suivant montre comment on peut définir une relation entre deux éléments de classes différentes. La figure 2.12 montre une description simple de deux entités génériques (MACHINE et NETWORK) et la définition d'une relation CONNECTED_TO. Cette relation est définie comme une relation qui pourrait exister entre une entité de type MACHINE et une de type NETWORK, ou comme une relation entre deux entités de type NETWORK.

```
structure MACHINE is
    Name,
    Supplier,
    {Monitor}+,
    Keyboard,
    Processor,
    [Disk],
    Memory,
    Operating_System,
    Network_Interface
end structure
```

```
structure NETWORK is
    Name,
    Topology,
    Speed,
    Medium
end structure

relation CONNECTED_TO is
    domain (NETWORK, MACHINE),
    range NETWORK
end relation
```

FIG. 2.12 – *Modélisation des relations dans Emanuel*

La figure 2.13 montre la réalisation d'un élément dcl-sparx1 de type MACHINE et d'un élément de type NETWORK, les deux sont liés par la relation CONNECTED_TO. La réalisation des éléments ressemble à celle de SySL.

On constate que les MIL possèdent une syntaxe moins lourde que de celle utilisée par SNMP ou l'OSI, que les techniques utilisées pour réduire la complexité du système représenté sont très efficaces, et que ces techniques sont très proches de celles utilisées dans les langages orientés objet.

```
component MACHINE is
    Name => "dcl-sparx1",
    Supplier => "Sun Microsystems"
    Monitor => Sun_Colour,
    Keyboard => Sun_Keyboard,
    Processor => sparc,
    Disk,
    Memory => "16Meg",
    Operating_System => "Solaris 1.0",
    Network_Interface => Ethernet_Controller,
    relation CONNECTED_TO(backbone)
end component
```

```
component NETWORK is
    Name => "Lancaster Campus Backbone",
    Topology => "Ethernet Bus",
    Speed => "10 Mbits/sec",
    Medium => "Co-axial"
end structure
```

FIG. 2.13 – *Réalisation d'une relation dans Emanuel*

3 Spécification des interfaces

Les interfaces constituent le moyen permettant aux différents éléments du système de communiquer entre eux. Il est donc normal que la deuxième étape après celle de la modélisation des entités administrées soit celle de la spécification des ces interfaces. La facilité et l'efficacité des programmes écrits dépendent largement de la qualité et de la richesse des interfaces que le modèle fournit. Les points que nous allons aborder sont :

Le niveau d'interface fournie : La puissance du système ainsi que la simplicité du développement des applications administratives en dépendent.

Les contraintes d'accessibilité : Il s'agit de répondre à la question "qui peut accéder à quoi". On a souvent besoin de restreindre l'accès aux ressources dont on déclare la disponibilité. Cette restriction peut être motivée par des contraintes de sécurité, ou par les règles de la conception modulaire du système.

3.1 Spécification des interfaces dans les MIL

La spécification des interfaces fait partie des fonctions importantes des MIL. Le langage SySL utilise les mots clefs *provides* et *requires* pour définir les ressources qu'un module peut fournir aux autres, et celles dont il a besoin. Mais elle ne spécifie que le nom de la ressource (fonction ou procédure), ce qui limite sévèrement les possibilité de vérifier les erreurs provoquées par un mauvais typage des paramètres. D'autres MIL utilisent le concept d'une interface *concrète* qui constitue un raffinement d'une interface *abstraite*. L'interface abstraite utilise uniquement les noms des ressources, alors que l'interface concrète est décrite d'une façon plus détaillée en donnant une liste de paramètres avec leurs types respectifs.

En ce qui concerne l'accessibilité aux ressources, les MIL offrent un premier niveau de spécification avec les mots clefs *provides* et *requires*. Pour aller plus loin et limiter l'accès à un sous ensemble de modules, il existe plusieurs techniques : on

peut limiter l'accès aux ressources à une section particulière de la décomposition hiérarchique (e.g. accès aux parents uniquement), comme c'est le cas dans Conic [KM85] et Darwin [Ma92]. On peut aussi énumérer d'une façon explicite les modules qui peuvent accéder aux ressources individuelles, c'est le cas de C++ [Lip91] qui utilise le mot clef *friend* (ami) pour spécifier les classes qui auront le droit d'accès aux variables et méthodes déclarées comme protégées *protected.*

3.2 Spécification des interfaces dans Moira

Pour communiquer avec le serveur centralisé, les applications clients comme *chfn* qui modifient le profile d'un utilisateur, ou *nfsmaint* qui gère les serveurs NFS, disposent d'une bibliothèque permettant d'envoyer un ensemble limité de requêtes prédéfinies. Il existe quatre types de requêtes : *retrieve, update, delete, et append.* Le protocole de communication entre le client et le serveur ne prévoit pas de requêtes de type *exécuter une action.* Cette limitation est due aux objectifs de Moira qui cherche uniquement à fournir une alternative aux fichiers de configuration classiques du système UNIX, et non pas à fournir un support pour des tâches administratives de haut niveau, comme la gestion dynamique de la configuration.

En ce qui concerne la visibilité des informations administratives, Moira prévoit un mécanisme limité de restriction de visibilité. Le serveur maintient une liste de contrôle d'accès (ACL) pour chaque objet de la MIB, cette liste contient les ID's des utilisateurs autorisés à accéder à cet objet. Il s'agit là d'un mécanisme de sécurité uniquement.

3.3 Spécification des interfaces dans SNMP

Le système d'administration de SNMP est construit selon le modèle client-serveur. Le serveur est nommé *agent* et il réside sur la station qui contient la MIB. L'application administrative réside sur une *station d'administration* et elle n'a pas accès directement aux variables de la MIB. Les opérations administratives sont effectuées grâce à des messages envoyés par l'agent au manager. Les opération définies sont les suivantes :

Get : La station d'administration récupère la valeur d'un objet depuis l'agent.

Set : La station d'administration modifie la valeur d'un objet de l'agent.

Trap : L'agent envoie une valeur non sollicitée d'un objet à la station d'administration.

Il n'est pas possible de changer la structure de la MIB en ajoutant ou en retirant des instances d'objets. Les seuls objets accessibles sont ceux qui constituent les *feuilles* de l'arborescence hiérarchique de types.

Pour spécifier les contraintes d'accessibilité, SNMP définit la notion de *communauté de stations.* Une communauté est un ensemble de stations d'administration et un agent. L'agent définit pour chaque communauté un *profile* constituée d'un ensemble d'objets visibles, ainsi qu'un mode d'accès (lecture, lecture-écriture) qui

définit les droits des stations d'administration membres de cette communauté sur les objets visibles. SNMP utilise des règles pour combiner les droits d'accès attribués aux stations avec l'attribut ACCESS qui fait partie de la spécification de l'objet. Le nom de la communauté, qui est local à l'agent, est utilisé comme une clé d'authentification ajoutée à chaque requête en provenance de la station d'administration.

L'interface offerte par SNMP présente les limitations suivantes :

- SNMP utilise un schéma d'authentification très rudimentaire. Le protocole assume que si la station utilise un nom correcte, la requête est acceptée par l'agent qui ne maintient pas la liste des stations membres de chaque communauté. Avec ce schéma très faible d'authentification, les administrateurs adoptent souvent une politique pessimiste et bloquent les droits d'écriture, ce qui rend SNMP plus convenable pour des tâches d'audit, où l'on peut se contenter d'opérations de lecture, que des tâches de contrôle où l'on aura besoin de modifier les valeurs des objets.

- Il est impossible de spécifier explicitement une action. En d'autres termes, SNMP n'offre pas de support pour les commandes impératives. La seule façon d'exécuter une commande est de le faire indirectement en demandant à l'agent de modifier la valeur d'un objet. Cette solution est peu flexible et comporte un grand risque d'erreur en l'absence de RPCs structurés avec une vérification des paramètres et un résultat d'exécution.

- Le protocole ne prévoit aucune possibilité de communication entre deux stations d'administration, ce qui ne permet pas la réalisation de programmes administratif où l'on a besoin d'un consensus entre plusieurs administrateurs.

En contre partie, SNMP reste très populaires dans les milieux d'administration des réseaux. La simplicité du protocole a encouragé un développement rapide d'un grand nombre de produits basé sur SNMP.

3.4 Spécification des interfaces dans CMIP

Dans CMIP, pour effectuer des opérations sur les objets administrés, les applications administratives (nommée *managers*) ne peuvent pas communiquer directement avec les objets administrés. L'application doit envoyer un message à un processus *agent* qui lui s'occupe de décoder le message et d'exécuter l'action appropriée. Les communications se font en utilisant le protocole CMISE qui reconnaît les types de messages suivants :

- Get/Replace : Pour lire/modifier la valeur d'un attribut.

- Set : pour remettre un attribut à sa valeur par défaut.

- Add/Remove : permet d'ajouter/retirer un élément à un attribut de type ensemble.

- Create/Delete : creé/détruit un objet administré.

- Action: pour exécuter une des actions de l'objet. Ces actions sont spécifiées pendant la définition de l'objet comme des éléments des paquetages qui constituent la classe de l'objet.

Les contraintes de visibilité des attributs sont définies lors de la déclaration de ces attribut comme une partie de la définition de l'objet administré. Ces contraintes sont décrites sous forme de droit d'accès et de modification avec les mots clefs: GET/ REPLACE/ GET-REPLACE. Le protocole de communication entre l'application et l'agent offre des facilités supplémentaires pour mieux structurer l'exécution. Il est possible d'exécuter une opération sur un groupe d'objets à la fois. Pour définir ce groupe, l'expéditeur du message peut spécifier dans le message la façon de le construire. Le groupe d'objets est obtenu à partir d'une partie du sous-arbre de l'arbre de contenance qui commence à un objet de base spécifié dans le message. L'ensemble des objets de ce sous-arbre est filtré grâce à une opération de filtrage sur un des attributs des objets de l'ensemble (les opérations ont été décrites lors de la définition des attributs). Dans le cas où plusieurs objets sont spécifiés, une option de *synchronisation* permet de dire si l'opération est atomique (soit tous les objets sont traités, soit aucun ne l'est) ou non.

Il est clair que l'interface offerte par CMIP est plus sophistiquée que celle offerte par SNMP, en particulier, le fait de permettre l'exécution des actions sur les objets administrés constitue un avantage considérable. On peut quand même exprimer quelques réserves sur ce modèle.

- L'architecture de l'OSI ne permet pas la communication directe entre l'administrateur ou l'application administrative et les objets administrés distants. Il faut passer par un agent en utilisant le protocole CMISE. C'est dû au fait que l'OSI ne permet les communications qu'entre les entités pairs. Or, les applications et les objets ne sont pas des entités pairs. Il existe donc un manque de transparence.

- Le fait d'accéder aux attributs à travers des méthodes génériques (Set/Get) peut être une source de confusion dûe à d'ambiguité. L'exemple suivant repris de [Zel93] permet d'illustrer cette ambiguité. L'ensemble des standards d'admnistration X.700 ont été écrits en adoptant le principe de «ne pas assumer que ce qui n'est pas indiqué est soit autorisé soit interdit». Prenons un attribut X d'un objet auquel on a attaché le droit de lecture avec le mot clefs GET. Cette information à elle seule ne permet pas de dire si l'agent doit accepter ou refuser les demandes de type SET ou REPLACE. Tout ce qu'elle permet de déduire est que l'agent accepte les requêtes de type GET. Si, pour résoudre cette ambiguïté, on décide d'ajouter un mot-clef GET-ONLY pour spécifier que l'attribut est accessible en lecture uniquement, on réduit l'extensibilité de la classe de l'objet. En effet, l'utilisation d'un tel mot-clef interdit la possibilité d'ajouter des droits supplémentaires dans les classes dérivées. Nous constatons donc que les méthodes génériques utilisées par CMISE ne sont pas compatibles avec la méthodologie de conception orientée-objet.

- Toutes les vérifications se font pendant l'exécution, et non pas à la compilation. Le processus agent doit vérifier les paramètres encapsulés dans les messages

et s'assurer de leur compatibilité. Ce qui complique énormément le code de l'agent, surtout que les sources d'erreur sont nombreuses (demande d'accès à un attribut non-existant, accès à un attribut existant mais le type d'accès demandé n'est pas autorisé, mauvais nombre de paramètres d'une action, etc.).

4 Spécification des contraintes de cohérence

La correction du fonctionnement d'un système informatique dépend largement de celle de sa configuration [Tho96, Tho94]. La dispersion géographique et la taille d'un système réparti rendent sa configuration *fragile*. De plus, ces mêmes facteurs peuvent faciliter la propagation des erreurs qui peuvent à leur tour en provoquer d'autres. La spécification de contraintes de cohérence qui peuvent être testées pour vérifier la correction de la configuration permet de réduire la vulnérabilité du système et permet donc un fonctionnement plus correct et moins risqué. Voici quelques exemples de ces contraintes :

- Un utilisateur doit avoir un mot de passe non vide et contenant au moins un chiffre et deux lettres.
- Dans la solution présentée par Dijkstra [Dij72] au problème des philosophes, le nombre des philosophes doit être supérieur ou égal à deux, et le système doit contenir un et un seul philosophe gaucher (qui demande la fourchette à sa gauche avant celle à sa droite). Cet exemple est particulièrement intéressant puisqu'il exprime des contraintes qui doivent être testées avant de déclencher un changement de configuration pour vérifier que la nouvelle configuration reste correcte. Ces contraintes doivent être vérifiées avant d'autoriser le retrait ou l'ajout d'un philosophe.

Les contraintes sont en général décrites d'une façon informelle en utilisant le langage naturel, qui est certainement l'outil le plus puissant pour les exprimer. En revanche, le langage naturel n'est pas compréhensible par l'ordinateur et il est donc difficile d'automatiser la procédure de vérification. D'où l'intérêt d'utiliser la notation formelle ou semi-formelle fournie par le langage de spécification de la configuration pour spécifier ces contraintes.

Cette fonction n'est présente dans aucun des modèles d'administration que nous avons présentés (ce qui est assez étonnant d'ailleurs). En revanche, elle fait partie des fonctions fournies par les MIL.

Le langage SySL, permet d'imposer plusieurs contraintes sur les composants. L'exemple de la figure 2.14 nous montre une contrainte qui stipule que toutes les instances de la classe WORKSTATION sont soit connectées au réseau, soit dotées de leurs propres disques durs. Le compilateur du langage vérifie cette contrainte et informe l'utilisateur des problèmes éventuels.

Mais les MIL ne permettent pas d'imposer de contraintes inter-modules ; en plus, ils s'intéressent uniquement à la description *statique* du système et n'offrent aucun mécanisme permettant d'imposer ces contraintes sur un système qui évolue dynamiquement.

```
assert WORKSTATION :
   forall i : member (i,WORKSTATION)
                and not (not_present(NETWORK)
                and not_present(DISK_SYSTEM))
```

FIG. 2.14 – *Spécification des contraintes d'intégrité dans SySL*

5 Conclusion

Ce chapitre nous a permis d'identifier les différentes fonctions que le langage de spécification doit fournir. Nous avons analysé les solutions présentées dans trois systèmes d'administration: MOIRA, SNMP et CMIP. Nous les avons choisis parce qu'ils constituent une bonne représentation de l'évolution historique des systèmes d'administration, et parce que SNMP et CMIP sont les standards reconnus pour l'administration des réseaux. Nous avons aussi analysé les fonctions que peuvent nous offrir les MIL qui ont un objectif commun avec le notre, celui de la modélisation des systèmes complexes à grand nombre d'éléments hétérogènes. Ces fonctions nous seront utiles dans la suite pour décider des meilleurs choix pour le langage que nous allons utiliser. Les fonctions que nous avons identifiées et que le langage doit fournir sont les suivantes:

- Le regroupement des entités administrées en classes d'entités ayant un comportement commun. Pour ce faire, il peut avoir recours aux techniques de décomposition hiérarchique du système et de raffinement successif de la représentation.

- La spécification des relations qui lient ces entités entre elles, la représentation devant être assez souple pour permettre l'évolution de ces relations.

- La spécification des contraintes d'intégrité d'une entité.

- La génération dynamique des instances des types abstraits qui représentent les classes d'entités administrées.

- Les utilisateurs et les applications administratives doivent être capables de surveiller l'évolution de l'état de l'entité représentée. Pour ce faire, le langage doit fournir un mécanisme de spécification de notifications asynchrones qui seront utilisées pour réaliser les tâches d'audit. Si un tel mécanisme n'existe pas, les applications seront obligées de scruter l'état des objets administrés de façon active, ce qui peut être source d'inefficacité car les messages engendrés consomment une partie non négligeable de la bande passante.

- La spécification des interfaces permettant de communiquer avec la ressource représentée. Ces interfaces peuvent être abstraites (sans paramètres) ou concrètes. Elles peuvent être génériques comme les méthodes Set/GetAttribute fournies dans CMIP, ou spécifiques comme dans un vrai modèle à objets où sont définies explicitement pour chaque variable les méthodes permettant son accès.

- Les interfaces doivent être de haut niveau. En particulier, elles doivent permettre de spécifier l'exécution des actions autres que la simple lecture ou écriture d'une variable.

Le tableau sur la page suivante montre l'ensemble de ces fonctions et leur présence dans les différents modèles présentés.

Systèmes	Moira	SNMP	CMIP	MILs
Langage	SQL	ASN.1	GDMO	varie
Types de base	int, string	limité par RFC1212	types ASN.1	dép. du lang.
Création de classes	difficile (manuellement en mode interactif)	oui, mais pas dans partie standard où il faut un nouveau RFC	oui	oui
Génération dynamique des instances	oui	non	oui	oui
Désignation des instances	indice dans une table	identificateur ASN.1	nom symbolique à construction hiérarchique	varie
Spéc. des relations	non	non	limité	oui
Spéc. des contraintes d'intégrité	non	non	non	oui
Spéc. des notifications	non	limité	oui	non
Interface	librairie C→SQL	protocole générique SNMP	protocole générique CMISE	interfaces abstraites
Spéc. d'actions dans l'interface	non	non	oui	oui
Difficulté d'apprentissage	bas	bas	élevé	acceptable
Possibilité de traiter un grand nombre de ressources	réduite	réduite	bonne, grâce à la technologie orientée objet	bonne
Description d'une MIB	intégrée, non extensible	intégrée, extensible	absente, mais fournit les mécanismes de construction	absente

3

Le langage de spécification

1 Introduction

Dans ce chapitre, nous présentons l'architecture de notre système d'administration. Nous nous intéressons en particulier au langage choisi pour spécifier la configuration de notre système.

Le Premier critère de choix est celui des fonctions fournies par le langage. Le chapitre précédent nous a permis de présenter l'ensemble des fonctions requises pour un langage de spécification de configuration telles qu'elles sont fournies par les solutions existantes (Moira, SNMP, OSI, MIL).

Il est clair qu'aucune des solutions présentées ne fournit l'ensemble des fonctions décrites dans le chapitre précédent. En effet, les langages orientés objets offrent la quasi totalité de ces fonctions, mais pas toutes. En particulier, ces langages ne permettent pas de spécifier les relations (la solution adoptée par l'OSI pour spécifier un ensemble limité et non extensible de relations est difficilement acceptable), ni les contraintes d'intégrité. En revanche, ils proposent de très bonnes solutions à des problèmes comme la décomposition hiérarchique, la généricité, le raffinement successif des spécifications et la spécification des interfaces. D'un autre coté les MIL constituent un excellent outil de description des systèmes hétérogènes à grand nombre de ressources, avec des techniques semblables à celles des langages orientés objets en ce qui concerne la décomposition hiérarchique, la modélisation générique et son raffinement, et la spécification des interfaces. Ils ont l'avantage de fournir les fonctions que les langages orientés objets n'offrent pas d'habitude, en particulier la spécification des relations et des contraintes d'intégrité.

Un autre critère de choix est celui de l'effort nécessaire pour mettre en œuvre le langage adopté. Il est en effet difficile de s'engager dans un projet de développement d'un langage spécifiquement conçu pour l'administration des systèmes. Ce n'est pas uniquement pour des raisons liées à l'effort nécessaire pour le développement, mais aussi parce qu'un tel langage risque de ne pas convaincre les programmeurs de son intérêt. Il risque aussi d'être difficile à comprendre pour les administrateurs qui en général ne s'intéressent pas à l'apprentissage de nouveaux langages. Cet argument réduit considérablement l'intérêt des MIL dont l'utilisation ne dépasse guère la communauté de génie logiciel où ils sont utilisés extensivement.

La solution idéale serait donc d'enrichir un langage orienté objet afin qu'il offre les

fonctions supplémentaires spécifiques aux MIL, comme la modélisation des relations fournie par Emanuel. Aussi avons nous décidé d'utiliser un modèle à base d'objets pour spécifier les entités du système administré et d'enrichir le langage à objets OC++ qui est le langage d'écriture d'applications réparties du système que nous voulons administrer. OC++ est une version du langage C++ auquel on a rajouté la persistance et la répartition [San95].

Nous présentons dans la suite de ce chapitre la plate-forme OODE (Object Oriented Distributed Environment) et son langage de programmation OC++. Ensuite, nous présentons comment OC++ est utilisé pour modéliser le système et notamment les extension apportées.

2 L'environnement de développement OODE

L'environnement de développement est la plate-forme OODE réalisée à l'unité mixte Bull-IMAG. Le but de cette plate-forme est de faciliter le développement et l'exécution d'applications coopératives manipulant des objets persistants et distribués. Les principales fonctions offertes par OODE sont :

- Un modèle d'exécution distribué offrant une distribution totalement transparente : une méthode est appelée sur un objet de la même façon, que l'objet appelé soit local ou distant.
- La programmation persistante : la durée de vie de l'objet ne dépend pas de celle de l'activité qui l'a créé. Le stockage et le chargement des objets en mémoire sont pris en charge par le système de façon transparente.
- La possibilité de partage simultané d'objets entre plusieurs applications.
- Des mécanismes de protection et de sécurité orientés objet.
- Une interface compatible avec les spécifications de CORBA [Sol92], qui est la norme actuelle pour les systèmes à objets.

2.1 Modèle d'exécution distribuée dans OODE

Un objet OODE constitue l'unité de partage. Il est caractérisé par un état et un ensemble d'opérations. Les objets OODE sont passifs et exécutés via des structures d'exécution indépendantes des objets. L'unité d'exécution, appelée *domaine* (pour domaine d'exécution), offre au programmeur une machine virtuelle multi-processus dans laquelle le parallélisme est apparent et la distribution cachée. Un domaine, qui correspond en général à une session d'une application pour un utilisateur, est composé d'un ensemble de flots d'exécution séquentiels (appelés activités). Les objets utilisés dans un domaine sont chargés et liés dynamiquement dans un espace d'exécution distribué appelé *contexte*. Un objet est automatiquement chargé dans la mémoire d'exécution lors du premier appel de l'une de ses méthodes.

Le modèle d'exécution de OODE est illustré par la figure 3.1 représentant une application agenda se déroulant pour le compte d'un utilisateur appelé John. Cette

application s'exécute dans un domaine d'exécution distribué sur les postes de travail de John et Jim. John effectue simultanément l'établissement d'un rendez-vous avec Jim via l'appel d'une méthode sur l'agenda de Jim, et la consultation dans son agenda d'un autre rendez vous avec Jim, ce qui se traduit par l'exécution en parallèle de deux flots de contrôle dans ce domaine, et l'appel par chacun de ces flots d'objets sur chacun des postes de travail. Le modèle d'exécution offert par OODE généralise

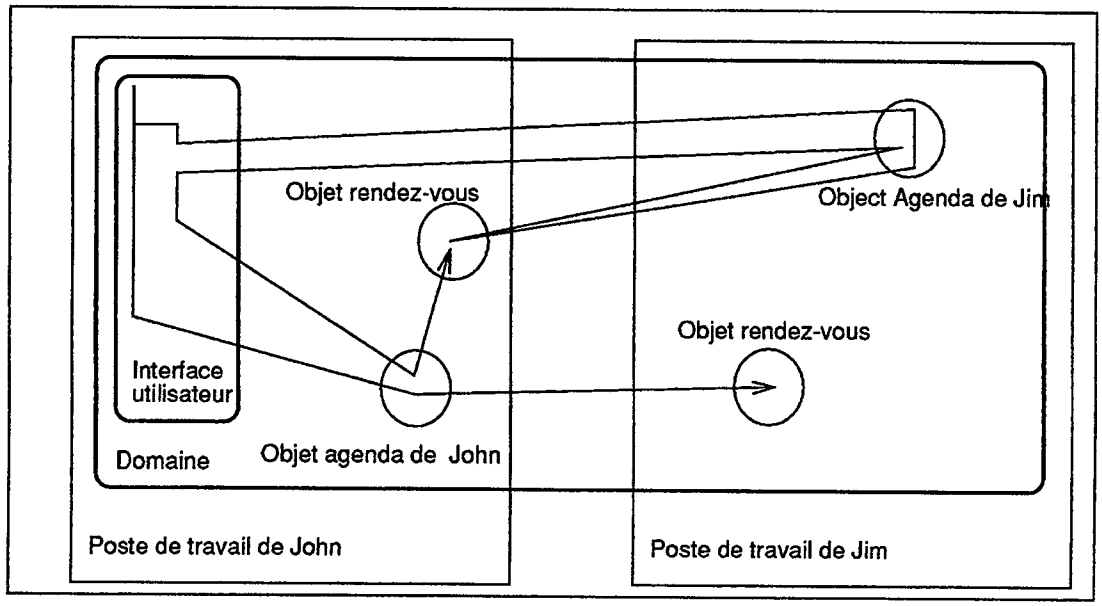

FIG. 3.1 – *Modèle d'exécution et objets distribués dans OODE*

donc à un ensemble de machines le modèle classique (mono-machine) de processus et flots d'exécution offert par tous les systèmes d'exploitation modernes. OODE offre au programmeur un modèle uniforme (processus et flots d'exécution distribués) qui remplace avantageusement le modèle client-serveur où le client et le serveur doivent être programmés de façon spécifique. Avec OODE une application distribuée se programme exactement comme une application centralisée, la distribution étant définie non-plus par programmation, mais implicitement suivant la localisation des objets utilisés par l'application.

2.2 Programmation persistante dans OODE

Un objet est dit persistant si sa durée de vie ne dépend pas de celle de l'activité qui l'a créé. Dans les langages de programmation classiques, ces deux valeurs sont identiques. Le programmeur doit donc programmer explicitement, en utilisant des fichiers ou une base de données, le stockage et la restauration des objets persistants. Le programmeur doit également effectuer la conversion entre les deux représentations : celle en mémoire d'exécution, et celle en mémoire secondaire.

Dans OODE tout objet accessible à partir d'un autre objet persistant est persistant et est donc sauvegardé automatiquement par le système en mémoire secondaire. Les objets sont chargés automatiquement depuis la mémoire secondaire lors de

l'appel de leurs méthodes. Le programmeur n'a pas à gérer les opérations de lecture et de conversion correspondantes.

Les objets qui risquent d'être utilisés en même temps sont chargés en même temps pour des raisons de performance et de taille de grain de chargement. Un tel ensemble d'objets est appelé *grappe*. Une grappe constitue donc l'unité de chargement en mémoire. Le regroupement des objets en grappes est effectué par le système en fonction des directives du programmeur. La gestion de la persistance dans OODE est

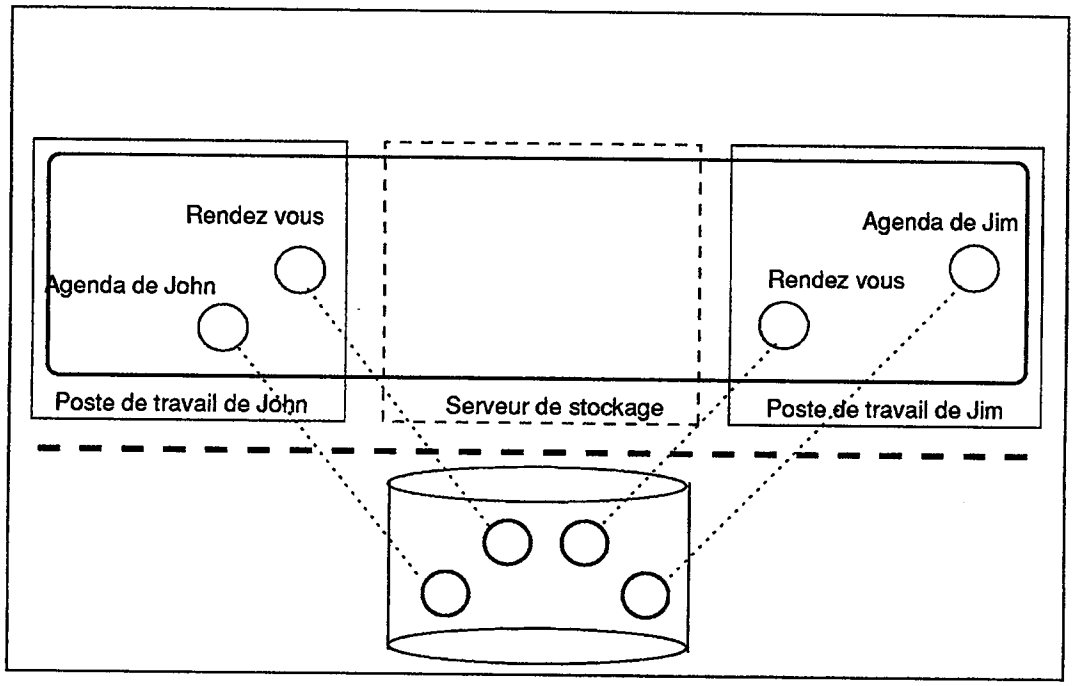

FIG. 3.2 – *Les objets persistants dans OODE*

illustrée par la figure 3.2, reprenant l'exemple de la section précédente, en supposant que la politique de stockage définie pour l'application agenda est de stocker tous les objets dans un serveur de stockage, et que la politique d'exécution est d'exécuter les méthodes des objets appartenant à chacun des utilisateurs sur son poste de travail (dans l'exemple on suppose qu'un objet rendez-vous est créé par John et un autre par Jim).

On accède à un objet à l'aide d'un pointeur distribué qui permet de l'identifier de manière unique et de le localiser. Un pointeur distribué est valide à la fois en mémoire d'exécution et en mémoire secondaire ce qui permet d'éviter les conversions lors du chargement et du stockage des objets. L'état d'un objet peut contenir des pointeurs distribués sur d'autres objets, ce qui permet de construire des structures complexes d'objets. Nous verrons dans la suite le mécanisme de déclaration et de conversion des pointeurs distribués.

2.2.1 Interopérabilité avec CORBA

En plus de la définition écrite en OC++, chaque classe distribuée doit impérativement être décrite en utilisant le langage IDL de CORBA. Cette interface possède exactement le même nom de la classe ; elle fournit la liste des attributs et des opérations exportées par la classe en question. Le compilateur vérifie que les deux représentations sont identiques. L'exemple suivant montre la description d'une classe OC++ et son interface IDL :

```
// Definition OC++
struct Date {long m, y, d;};
struct Transaction {
      float amount;
      Date date;
};
distributed class Account {
  private:
      Transaction history[10];
  public:
      long number;
      char name[32];
      const char addresse[32];
      const Date creation;
      double balance;
      virtual void credit (float amount);
      virtual void debit  (float amount);
      virtual void summary(double &balance,
                           Transaction history[]);
}

// Définition IDL
struct Date { long m, y, d;};
struct Transaction {
      float amount;
      Date  date;
}

typedef char string32[32];

interface Account{
attribute long number;
attribute string32 name;
readonly  attribute string32 address;
readonly  attribute Date     creation;
attribute double balance;

void credit (in float amount);
```

```
void debit  (in float amount);

void summary(out double balance,
             out Transaction history[10]);
}
```

Dans la suite de ce rapport, quand nous avons besoin d'une fonction fournie par OODE, nous en donnerons une brève description. La réalisation de OODE et de OC++ ne font pas partie de notre contribution, nous nous contenterons donc de définir les fonctions fournies sans expliquer leur mise en œuvre. Pour plus d'informations sur OODE et le langage associé OC++, le lecteur peut consulter [Ca94] et [San95].

3 La modélisation du système

Les entités à administrer sont représentées par des *objets persistants*, qui sont des instances de *classes distribuées*. Les classes distribuées sont similaires aux classes classiques du langage C++, sauf que leurs déclarations sont précédées par le mot clef *distributed*.

Les objets persistants sont instances des classes distribuées ; ils peuvent être utilisés dans un contexte distribué, accepter des appels distants à partir des programmes ou des objets chargés sur d'autres sites. Ils sont aussi persistants et leur durée de vie ne dépend pas de la durée d'exécution du programme qui les a créés.

3.1 Les types de base

Le langage OC++ permet l'utilisation de tous les types de base que l'on peut utiliser dans C++, à savoir les types simples (entiers, réels, caractères, et void) et les types structurés (enregistrements, unions et vecteurs). En plus de ces types classiques, OC++ introduit un type de base supplémentaire, appelé pointeur distribué, qui est la référence à un objet persistant. Il s'agit d'un identificateur unique attribué à un objet au moment de sa création et qui ne change pas pendant la durée de vie de l'objet. La déclaration des pointeurs distribués est identique aux pointeurs classiques, le fait qu'un pointeur soit distribué ou pas dépend de la classe de l'objet référencé.

Exemple :

```
// Définition d'une liste distribuée
distributed class Item {
public :
  SomeType value ;
  Item *next ;
};
```

La manipulation des pointeurs distribués est différente de celle des pointeurs classiques. Le compilateur du langage impose certaines restrictions à l'utilisation de ces pointeurs. Ces restrictions ont pour but de protéger le programmeur contre les

erreurs de manipulation. Il ne faut donc pas les voir comme une limitation, mais plutôt comme une protection utile, les opérations interdites n'ayant en effet aucune signification dans le cas d'un pointeur distribué. Il s'agit des restrictions suivantes:

- La conversion des pointeurs est strictement interdite sauf dans les cas suivants: il s'agit du pointeur nul (constant 0), il s'agit d'une conversion d'un pointeur distribué vers dvoid*, il s'agit d'un pointeur d'une classe distribuée A que l'on veut convertir vers le pointeur d'une classe B dérivée de A.
- Les opérations arithmétiques ne s'appliquent pas sur les pointeurs distribués (++,−−,+,−,+ =,− =).
- La comparaison entre les pointeurs distribués est limitée au test d'égalité.
- Il n'est pas possible d'avoir des pointeurs vers les attributs des objets persistants, ni d'obtenir de tels pointeurs en ajoutant un déplacement au pointeur qui référence l'objet.
- Les opérateurs new et delete ne peuvent être utilisés pour créer des vecteurs d'objets distribués.

```
distributed FstClass ;
distributed ScndClass :FstClass ;
FstClasse *fstptr ;
ScndClass *scndptr ;
int *intptr ;
dvoid *any ;

int f(){
  // Conversion des pointeurs distribués

fstptr = 10 ;                    //Error
fstptr = NULL ;                  //Ok
fstptr = scndptr ;               //Ok
any = scndptr ;                  //Ok
scndptr = (ScndClass *)intptr ;  //Error
}
```

3.2 La représentation des ressources

Chaque ressource du système est représentée par un objet distribué[ORdP94], appelé *objet administré*. Les objets sont des instances de classes, et les objets de la même classe constituent une famille ayant les caractéristiques communes définies par leur classe. Les classes des objets administrés héritent un ensemble d'attributs et une interface uniforme de la classe de base ManagedObject. La figure 3.3 présente l'ensemble des classes utilisées pour la représentation des ressources du système.

On peut distinguer deux grandes catégories de classes:

Objets actifs: ces objets représentent les flots d'exécution qui communiquent entre eux à travers des objets passifs. Ils sont surtout caractérisés par leur

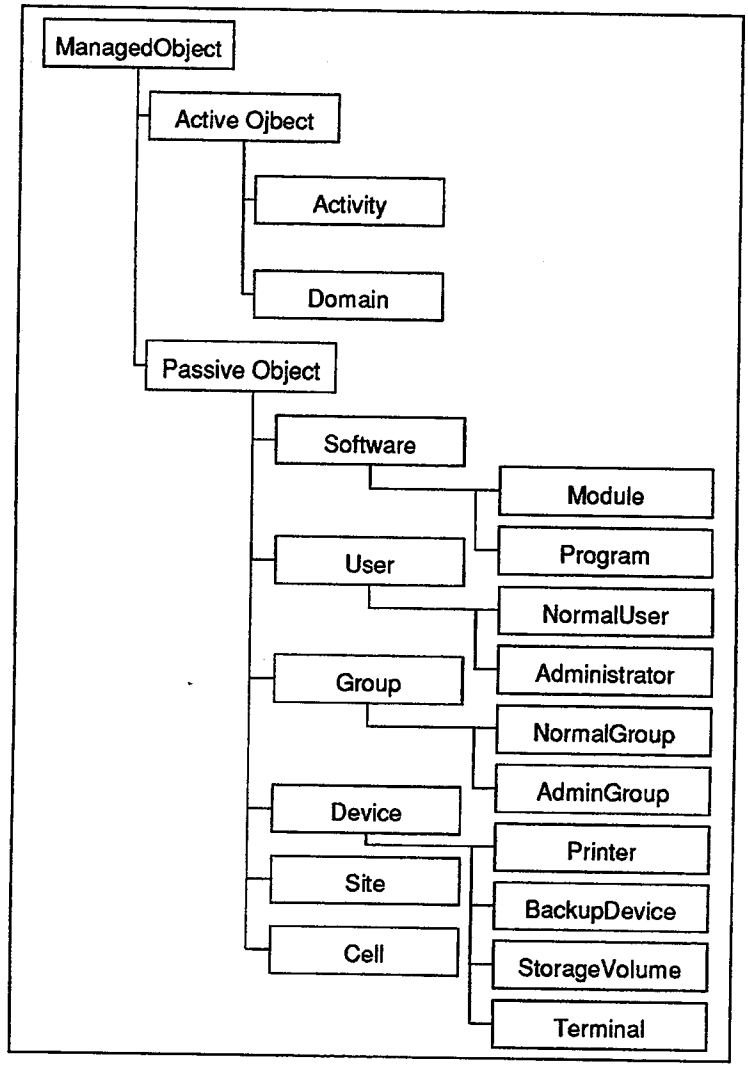

FIG. 3.3 – *Arbre d'héritage des objets administrés*

durée de vie relativement courte, qui est égale à la durée d'exécution des flots correspondants. Les objets actifs sont à leur tour divisés en deux sous-classes. L'unité d'exécution est le *domaine* qui correspond à une session d'une application pour un utilisateur. Un domaine est composé d'un ensemble de flots d'exécution séquentiels appelés *activités*. Les activités sont des flots d'exécution répartis qui représentent l'unité de concurrence.

Objets passifs : ces objets ne représentent pas des flots d'exécution, mais des ressources physiques du système. Ils ont une durée de vie longue qui peut aller jusqu'à plusieurs années, ce qui pose un problème concernant leur mise à jour et la gestion de leur évolution. Ce problème sera abordé en détail dans le chapitre suivant. Le seul moyen d'exécuter le code de ces objets est leur appel par un objet actif.

Les classes des objets administrés sont évidentes et ne nécessitent pas d'expli-

cation particulière à l'exception de la classe «software». Cette classe regroupe l'ensemble des «exécutables» du système. Les instances de cette classe sont caractérisées par la méthode «run» qu'elles exportent. La classe «software» est décomposée en deux sous-classes: les *modules* et les *programmes*. Les modules sont associés aux activités, et la création d'activités correspond à la création d'un flot d'exécution qui exécute la méthode «run» d'un module. Ce flot d'exécution peut se propager vers d'autres modules et d'autres objets administrés via des appels de leurs méthodes. La classe «program» représente les applications telles qu'elles sont vues par les utilisateurs. Le compilateur OC++ génère en effet un fichier exécutable UNIX, cet exécutable doit avoir une représentation au niveau OODE. L'appel de la méthode «run» d'un objet de la classe «program» correspond à l'exécution du programme UNIX associé. Les programmes sont associés aux domaines, et la création d'un domaine correspond à la création d'un premier flot d'exécution qui exécute la méthode run d'un objet de la classe program.

Notons que les objets administrés ne sont pas tous persistants. En effet, les objets distribués doivent être enregistrés et dotés d'un nom symbolique pour devenir persistants; les objets non enregistrés seront détruits par un ramasse miettes. Dans certains cas, en particulier pour les objets actifs la persistance n'a aucun intérêt puisque ces objets représentent des entités volatiles (les domaines et les activités) et il est normal qu'ils soient détruits une fois que l'exécution des activités correspondantes soit terminée.

3.3 La structure de la MIB

L'unité d'administration est la *cellule*, un ensemble de machines hétérogènes connectées par un réseau local; chacune de ces machines exécute une instance du démon «oode» et a accès à l'espace de stockage global et persistant. Toute application dans le système peut avoir accès à tous les objets de la MIB (pourvu qu'elle possède les droits nécessaires). La notion d'unité d'administration signifie qu'une seule politique d'administration est appliquée au niveau d'une cellule. Exemple: le compte d'un utilisateur est un compte global pour l'ensemble des machines constituant la cellule et non local à une seule machine.

La MIB est structurée d'une façon hiérarchique qui reflète une relation de contenance. Ainsi, à la racine de l'arbre on trouve la cellule, ensuite l'ensemble des objets représentant les ressources du système groupés par catégorie. Nous avions le choix de regrouper les objets de la MIB par catégorie d'objets ou par localisation physique comme dans CMIP ou Tobias. La seconde structure reviendrait à considérer que la MIB d'une cellule est le résultat d'un regroupement par le haut de MIBs séparées et indépendantes. En revanche la première donne à la MIB une image globale à la cellule, c'est ce qui nous a amené à opter pour ce choix. Cette structure est conforme à la stratégie des systèmes répartis orientés objets (et les systèmes répartis à mémoire partagée en général) où la répartition est, si on le souhaite, totalement transparente au niveau des applications.

Il serait quand même intéressant de pouvoir connaître la localisation physique d'une ressource. Cette localisation sera définie grâce à une relation de *résidence* qui associe chaque objet au site où réside la ressource correspondante (et non pas au

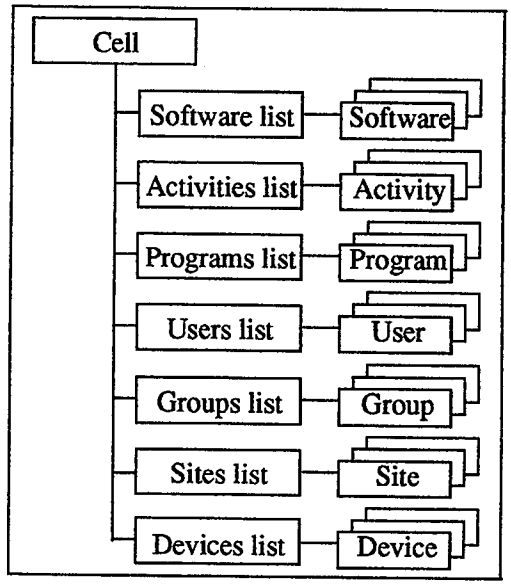

FIG. 3.4 – *Structure de la MIB*

site où l'objet est stocké). La description de cette relation sera donnée à la fin de cette section.

3.4 La désignation des objets de la MIB

La gestion des fonctions de désignation fait aussi partie des tâches que le service de gestion de la configuration doit fournir. Il s'agit de trouver le meilleur mécanisme permettant de désigner les éléments qui composent le système. Étant donné que dans notre modèle chaque ressource est représentée par un objet administré, nous devons discuter du meilleur moyen pour désigner ces objets.

L'utilisation des pointeurs distribués pour identifier les objets administrés a plusieurs avantages sur le schéma d'identification basé sur la structure de contenance de l'OSI. Ces avantages sont les suivants :

1. Elimination des problèmes dûs à la migration. L'identificateur de l'objet ne change pas pendant sa durée de vie, alors que celui utilisé par l'OSI risque de changer si l'objet change sa position dans l'arbre de contenance. Ce qui rend les valeurs des variables de type objectId non fiables.

2. Garantie d'unicité. La valeur d'un pointeur désignera toujours un seul objet. Si jamais cet objet est détruit, cette valeur ne sera pas réutilisée. Alors que rien n'empêche que l'identificateur symbolique d'un objet de l'OSI soit réutilisé pour désigner un autre objet.

3. Protection. Les pointeurs distribués sont protégés par le compilateur. Il est impossible de leur affecter une valeur qui ne soit pas correcte. Alors que les identificateurs de l'OSI ne le sont pas.

Le schéma de désignation de l'OSI présente pourtant deux avantages non négligeables. D'abord, l'identificateur est structuré d'une façon hiérarchique, alors que les pointeurs distribués sont «plats». Deuxièmement, l'identificateur est lisible par l'utilisateur humain, puisqu'il s'agit d'une chaîne de caractères à laquelle on peut attacher une signification quelconque, ce que l'on ne peut pas faire avec les pointeurs distribués.

Le service de désignation essaye d'offrir la meilleure des deux représentations. La majorité des systèmes d'exploitation utilisent deux types de noms: des noms *symboliques* constitués de chaînes de caractères pour les utilisateurs, ce qui permet de rattacher un sens aux noms des objets, et une autre représentation interne au système qui utilise une chaîne de bits. Le service de désignation est responsable de la gestion des associations des noms symboliques des objets à leurs pointeurs distribués. La plupart des propositions que nous faisons pour le service de désignation résultent du travail de Nabil Layaida sur le service de désignation dans le système Guide [Lay93].

3.4.1 Construction de l'espace de noms

Pour construire l'espace de noms nous avons à choisir parmi plusieurs possibilités: Soit utiliser la structure déjà existante de la MIB, comme c'est le cas dans CMIP, ou essayer de créer une nouvelle structure spécifique à la gestion des noms. Nous avons écarté la deuxième solution parce que nous voulons éviter d'avoir à gérer plusieurs structures qui risquent d'être redondantes. En effet, si l'on veut utiliser une structure hiérarchique pour construire les noms symboliques, celle fournie par la MIB semble tout à fait adéquate. En plus, le fait que la MIB soit organisée par catégorie d'objets et non pas par site empêche l'apparition du nom du site où réside une ressource dans le nom symbolique associé à cette ressource. Il s'agit ici d'une propriété très importante du système de désignation, celle de la *transparence* de localisation [Ker89].

La deuxième question qui se pose est de savoir où stocker les noms symboliques associés aux objets et comment construire le nom d'un objet. Dans le système d'administration de l'OSI, le nom local d'un objet est rattaché directement à l'objet sous forme d'un attribut. Le nom symbolique d'un objet est construit en suivant la chaîne de contenance entre l'objet et la racine de la MIB (qui est locale à un site). Donc si un objet B est contenu dans un objet A, et si A devient inaccessible, le chemin d'identification de B sera coupé, et on ne pourra plus localiser B depuis son nom symbolique. CMIP ne cherche pas à résoudre ce problème parce que la MIB est locale à un site. La probabilité d'avoir seulement une partie de la MIB inaccessible est quasiment inexistante dans ce cas. En revanche, notre système utilise une MIB totalement répartie, où les objets sont stockés sur plusieurs serveurs de stockage. Le problème d'inaccessibilité d'une partie de la MIB n'est donc pas exclu (ex: coupure réseau, panne, arrêt du site qui contient le serveur de stockage pour effectuer des mises à jour, etc.). Nous avons donc décidé d'adopter une solution basée sur les principes suivants:

1. Stocker le nom symbolique attribué à l'objet à l'extérieur de cet objet, ce qui permet d'identifier l'objet même s'il n'est pas accessible. En plus, cette solution

permet d'attribuer plusieurs noms symboliques au même objet.

2. Étant donné que la MIB est structurée par catégories, l'espace de noms ne sera pas basé sur une relation de contenance.

3. La structure utilisée pour stocker les noms symboliques sera dupliquée afin d'empêcher le problème de non-résolution de noms dû à une rupture dans la chaîne de résolution. Cette duplication constitue une exception à notre approche qui vise à ne pas gérer la tolérance aux pannes au niveau du noyau d'administration. La décision se justifie par l'importance de la fonction de désignation et donc de sa disponibilité.

4. Les copies différentes de la même structure seront placées sur des sites différents afin d'augmenter leur disponibilité.

L'espace de désignation est construit à base de *répertoires*. Un répertoire est un objet qui maintient une liste d'associations entre des noms symboliques «simples» et des pointeurs d'objets, il peut aussi contenir d'autres répertoires. L'espace de noms de la cellule est construit par un chaînage de ces répertoires. L'exemple décrit dans la figure 3.5 illustre cette construction hiérarchique. L'objet qui représente l'utilisateur Alain possède le nom symbolique «/SIRACCELL/USERSFOLDER/ARIAS/Alain».

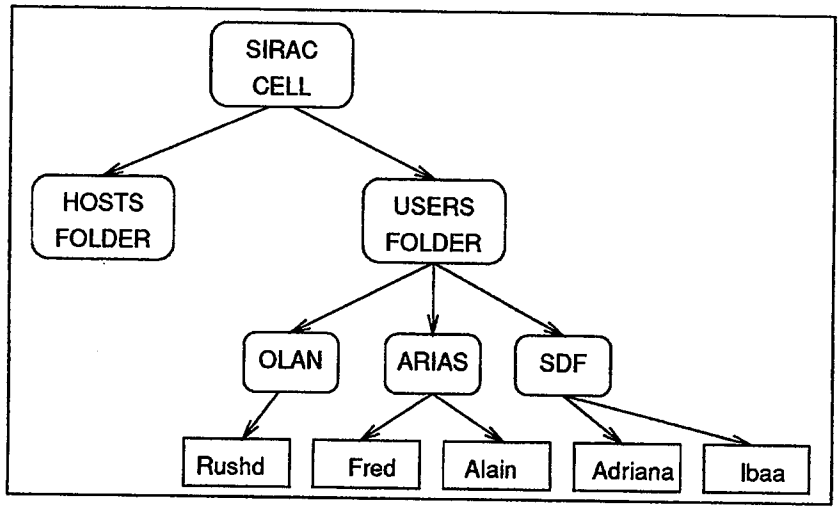

FIG. 3.5 – *Exemple de l'espace de désignation*

L'association entre un objet et son nom est définie par une instance de la classe «ObjectAssociation». L'association entre un répertoire et son nom est définie par une instance de la classe «FolderAssociation». Les deux classes sont dérivées de la classe «Association».

```
distributed class Association
{
  public :
    //Obtention et modification du nom symbolique de l'association
    setName(char symname[MAX_STR]) ;
    getName(char symname[MAX_STR]) ;
    ManagedObject *getObj() ;
  private :
    char     name[MAX_STR] ;
}
```

```
distributed class ObjectAssociation : :Association
{
  public :
    //Récupération de l'objet identifié par l'association
    ManagedObject *getObj() ;
  private :
    ManagedObject *obj ;
}
distributed class FolderAssociation : :Association
{
  public :
    //Récupération du répertoire identifié par l'association
    Folder *getFolder() ;
  private :
    //Copie principale du répertorie.
    Folder *fol ;
    //Liste de copies secondaires du répertoire.
    FolderList *copies ;
    int nbcopies ;
}
```

Un répertoire contient une liste d'associations potentiellement infinie. Cette liste contient les associations qui désignent les objets et les sous-répertoires référencés dans ce répertoire. Il existe donc deux types d'opérations possibles sur un répertoire. D'abord les opérations concernant les objets, qui consistent à ajouter et à détruire des instances d'association, et à chercher le pointeur distribué d'un objet à partir de son nom. Ensuite les opérations concernant les répertoires, qui consistent à ajouter et à détruire des répertoires, et de donner le nombre d'entrées dans le répertoire. La classe « Folder » qui représente les répertoires est décrite ci-dessous :

```
distributed class Folder
{
  public :
    //Opérations sur les objets
    void setFolderName(char folname[STR_MAX]) ;
    void getFolderName(char folname[STR_MAX]) ;
    int addObject(char objname[STR_MAX], ManagedObject *obj) ;
    int deleteObject(char objname[STR_MAX]) ;
    int renameObject(char objold[STR_MAX],objnew[STR_MAX]) ;
    ManagedObject *searchObject(char objname[STR_MAX]) ;
    //Opérations sur les répertoires
    Folder *getParentFolder() ;
    addFolder(char folname[STR_MAX], Folder *foldref) ;
    delFolder(char folname[STR_MAX]) ;
    destroyFolder(char folname[STR_MAX]) ;
    getFolder(char folname[STR_MAX]) ;
  private :
    char foldername[STR_MAX]) ;
    AssocLIST assocs ;
    Folder *parentfolder ;
    int foldersize ;
}
```

3.4.2 Le serveur de noms

Le service de noms est utilisé par les applications pour pouvoir associer des noms symboliques aux objets. L'opération la plus couramment utilisée est la recherche de la valeur du pointeur distribué de l'objet à partir de son nom symbolique. Elle est réalisée sous forme d'une méthode de la classe «NameServer» qui exporte l'ensemble des méthodes suivantes:

```
distributed class NameServer
{
  public :
    Folder *getRootFolder() ;
    Folder *getCurrentFolder() ;
    int setCurrentFolder(char folname[STR_MAX]) ;
    int moveObject(char source[STR_MAX],char destination[STR_MAX]) ;
    int moveFolder(char source[STR_MAX],char destination[STR_MAX]) ;
    ManagedObject *lookupObject(char nom [STR_MAX]) ;
  private :
    Folder *currentfolder ;
    Folder *root ;
    NameCache *cache ;
}
```

A chaque domaine (exécution d'une application distribuée) on associe une ins-

tance de la classe NameServer. Cet objet permet aux activités qui s'exécutent dans le contexte défini par le domaine de naviguer à travers l'espace de noms de la cellule. Les méthodes essentielles exportées par le serveur de noms sont :

- La méthode moveFolder permet de déplacer un répertoire, ainsi que le sous arbre de répertoires contenu dedans vers un autre emplacement.

- La méthode lookupObject permet de retrouver le pointeur distribué d'un objet à partir de son nom symbolique. Pour le faire, la méthode effectue un parcours récursif de la chaîne des répertoires depuis le répertoire racine pour localiser l'objet recherché. Cette procédure est appelée *résolution du nom symbolique.*

Pour accélérer les opérations de recherche, le serveur utilise un *cache de noms.* Il s'agit d'une table qui peut contenir dix entrées, cette table maintient l'association entre le nom absolu d'un répertoire et son contenu. Ainsi l'opération de lookupObject examine d'abord le contenu de cette table avant d'entreprendre une recherche à partir de la racine : si le nom est résolu le résultat est immédiat, dans le cas contraire (cas d'échec ou de succès partiel) un nouveau parcours de l'arbre est nécessaire.

La politique de gestion du cache est LRU (Least recently used), avec une invalidation supplémentaire si le pointeur fourni par le cache s'avère obsolète. La validité des références est donc vérifiée au moment de l'accès. Le cache est créé à la création du serveur de noms, il se remplit au fur et à mesure des opérations lookupObject. Des études expérimentales (voir K.W Shirrif [SO86]) sur le comportement des caches prévoit une stabilisation rapide de son contenu pour une capacité de dix noms, à cause de la localité des noms utilisés par les applications. Ce cache est réalisé par la classe suivante :

```
distributed class NameCache
{
  public :
    folder *lookupCache(char search[STR_MAX],char res[STR_MAX]) ;
    int insertFolder(char folname[STR_MAX], Folder *foldref) ;
    int invalidateFolder(char path[STR_MAX]) ;
    int isFull() ;
}
```

La méthode lookupCache permet de chercher l'existence d'un répertoire dans le cache. Elle envoie en réponse le pointeur distribué du répertoire trouvé et copie le chemin restant dans la chaîne de caractères res. S'il s'agit d'un échec total, le pointeur renvoyé est nul. En revanche, s'il s'agit d'un succès total, la chaîne res est vide. La méthode invalidateFolder permet de retirer un répertoire obsolète du cache ; tous les chemins qui contiennent la chaîne «path» donnée en paramètre sont invalidés et retirés du cache.

3.4.3 Disponibilité des répertoires

La disponibilité du service de nommage est un facteur critique pour le bon fonctionnement des applications. Cette disponibilité est accrue grâce à un mécanisme de duplication. Les répertoires sont dupliqués et chaque association contient une liste

de répertoires équivalents. Ces copies sont vues par le serveur de nom comme un répertoire unique qui possède un seul nom symbolique. Le système gère la modification des copies multiples d'un seul répertoire. Il gère aussi le placement des copies sur les différents volumes de stockage.

Les sources de pannes qui peuvent rendre un répertoire inaccessible sont variées. Au niveau de l'application ce problème se traduit par la génération d'une exception lorsque le serveur de noms tente d'accéder au répertoire indisponible pendant l'exécution d'une opération de résolution de nom. Cette exception est dirigée vers une fonction traitante (exception handler) qui effectue alors un traitement d'erreur. Le traitement consiste à trouver une copie disponible dans la liste des copies du répertoire. Cette copie sera la nouvelle copie primaire. La duplication du chemin d'accès d'un objet se fait lors de l'attribution d'un nom à cet objet (en général, cette attribution est effectuée lors de l'ajout de l'objet dans la MIB, au moment ou l'objet devient persistant).

```
Pour Chaque objet enregistré obj faire :
  volume =getVolId(obj) ;
  Pour chaque répertoire rep du chemin d'accès faire :
    liste← {volumes des différentes copies}
    Si vol ∉ liste alors
    newcopie = clone(répertoire) ;
    ajouter newcopie à la liste des copies ;
    Fsi ; // test d'appartenance à la liste
  Fin_Pour // parcours des répertoires
Fin_algo
```

La fonctions «getVolId» permet d'obtenir l'identité du volume sur lequel est stocké l'objet en question. La fonction «clone» permet d'obtenir une copie exacte d'un objet. Le service de stockage permet de regrouper un ensemble d'objets sur le même volume dans un groupe d'objets appelé *grappe*. La grappe constitue l'unité de chargement des objets persistants. D'où l'intérêt de stocker les objets qui risquent d'être accédés en même temps au sein de la même grappe. Les répertoires stockés sur le même serveur sont alors regroupés dans une seule grappe, afin d'améliorer la vitesse d'accès. L'algorithme est basé sur une idée simple : pour tout objet de la MIB il existe un chemin d'accès construit de répertoires stockés sur le même serveur de stockage que cet objet. Ce qui garantit la propriété ABMA-Résistant (All But Manager Access) : il existe toujours un chemin permettant de localiser un objet à partir de son nom symbolique, sauf si l'objet lui même n'est pas accessible. Le coût de l'algorithme en terme d'espace consommé est très limité, puisqu'il ne peut exister plus qu'une copie d'un répertoire par volume de stockage. Il ne peut donc y avoir plus que N copies des répertoires où N représente le nombre des serveurs de stockage du système.

La mise à jour des répertoires dupliqués se fait en utilisant une copie primaire. Pour chaque répertoire, une copie primaire est définie, et lors de chaque modification du contenu du répertoire, la copie primaire est modifiée en priorité suivie par les autres copies. Si jamais la copie primaire devient inaccessible suite à une panne, une autre est choisie dans la liste des copies secondaires, la copie primaire est alors remise

dans la liste des copies secondaires. Pour éviter que cette copie ne soit pas à jour au moment ou elle redevient accessible, on associe un compteur à chacune des copies. Ce compteur est incrémenté lors de chaque modification du contenu du répertoire. Si pendant la mise à jour d'un répertoire on découvre que le compteur associé à une copie secondaire est inférieur à celui de la copie primaire, la copie secondaire est recopiée à nouveau depuis la copie primaire.

3.5 Les extensions au langage

Le langage OC++ n'offre pas de support à toutes les fonctions que nous avons identifiées dans le chapitre précédent. C'est l'inconvénient du choix d'un langage existant par rapport à un langage sur mesure. En contre-partie, ce choix présente l'avantage d'une économie considérable d'effort de développement. Nous devons donc ajouter certaines extensions au langage pour fournir les fonctions qui nous manquent. Il s'agit de permettre la modélisation des relations et la spécification et la vérification des contraintes de cohérence. Il existe deux techniques pour ajouter des extensions à un langage orienté objet [ELM+92].

- Fournir une bibliothèque de classes d'objets qui offrent les fonctions désirables. C'est le moyen le plus convenable et le plus couramment utilisé pour introduire de nouveaux services aux langages orientés objets. Cette bibliothèque peut aussi contenir des classes *abstraites*, qui ne seront pas utilisées directement pour générer des instances, mais pour générer de nouvelles classes qui vont hériter de ces classes les propriétés désirables. Les instances des classes dérivées auront alors le même comportement et l'ensemble d'opérations que l'on voulait introduire. Cette solution est très utile dans les cas où l'on cherche à ajouter de nouvelles propriétés aux opérateurs ou aux entités du langage, à modifier leur comportement (introduire la distribution, la fiabilité, ou la persistance).

- La dernière technique consiste à modifier la syntaxe du langage pour introduire les nouvelles instructions, types, unités lexicales, et contraintes sémantiques qui correspondent aux fonctions requises. C'est la façon la plus "naturelle" pour étendre un langage, et la solution la plus complète aussi.

Nous avons décidé d'exclure la deuxième technique pour plusieurs raisons. La raison principale est l'état du langage OC++ qui n'est pas encore stable et qui reste en phase de développement. En ajoutant des extensions à la version actuelle, on risquerait d'être incompatible avec les versions suivantes du compilateur. La deuxième raison est la volonté de garder OC++ le plus proche possible de C++, et donc plus familier aux programmeurs habitués à ce langage. Si nous introduisons beaucoup de modifications au corps du langage, ces programmeurs risquent de ne pas se familiariser facilement avec le langage obtenu, et nous perdons l'un des avantages importants de OC++. La dernière raison est liée à la nature même des langages orientés objets, qui sont extensibles par construction grâce aux mécanismes d'héritage et de conformité. Puisque nous utilisons un langage orienté objet, il semble plus logique d'utiliser les techniques d'extension spécifiques à ces langages, à savoir l'utilisation d'une bibliothèque de classes. Le problème avec ce type de solution provient de la

difficulté d'imposer des contraintes syntaxiques et sémantiques. Or, vu le nombre de fonctions supplémentaires que nous comptons ajouter (nous introduisons pour le moment les notifications d'événements et la spécification des relations, il serait imprudent de ne pas vérifier ces contraintes.

Nous avons donc décidé d'adopter une solution hybride. Nous allons fournir une bibliothèque de classes qui représente l'ensemble des types abstraits et structures de données nécessaires, ainsi qu'un ensemble de *macro-fonctions* qui permettent de traduire les spécifications décrites en instructions OC++ grâce à un *préprocesseur*. Ce préprocesseur sera responsable de la vérification de la correction de la spécification des extensions et de la génération du code OC++. Cette technique nous permet d'ajouter les fonctions nécessaires sans avoir besoin de modifier le compilateur de base, le développement des macros nécessaires et du préprocesseur étant une tâche moins compliquée que la modification d'un compilateur. La figure 3.6 illustre le cycle de vie d'un objet administré dans notre système.

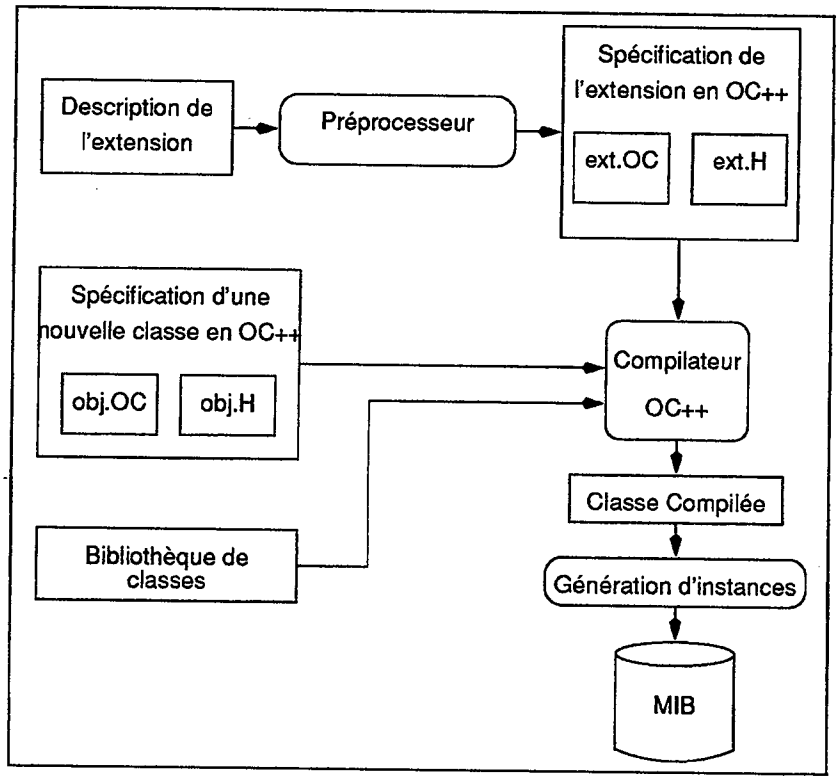

FIG. 3.6 – *Schéma de création d'un objet administré*

Il est vrai que l'utilisation d'un préprocesseur semble contraire aux principes d'extension des langages orientés objets. L'exemple suivant permet d'illustrer l'intérêt de ce préprocesseur : prenons la spécification d'un relation à n éléments. Nous représentons cette relation par une classe «myrelation» dérivée de la classe abstraite **Relationship**. La classe myrelation possède n variables d'état, chacune appartenant à un type OC++.

```
distributed class
```

```
myrelation : Relationship
{
 public:
  type1 var1;
  type2 var2;
  .
  .
  typen varn;
}
```

Pour définir la classe myrelation, nous avons besoin de définir les méthodes permettant l'accès à ces variables. Or, nous ne connaissons pas a priori la dimension des relations définies dans le système, ni les types de leurs champs. Il est donc impossible d'imaginer un protocole générique permettant d'accéder aux champs d'une relation quelconque. De plus, même si l'on arrive à résoudre le problème de dimension d'une relation en utilisant une structure de taille variable pour représenter les relations (une liste, par exemple), on aura toujours le problème de vérification des types des valeurs affectées aux champs de la relation. Il est vrai que l'on peut utiliser la conformité pour les champs de type pointeur distribué en utilisant une liste dont les éléments appartiennent à la classe ManagedObject, mais cette propriété ne s'applique pas aux autres types de OC++ (entiers, caractères, etc.). De plus, l'utilisation de la conformité d'une manière aussi générique affaiblit considérablement la possibilité de vérifier les types des valeurs affectées pendant la compilation puisque toute instance d'une classe dérivée de ManagedObject sera acceptée pour n'importe quel champ de la relation, ce qui augmente considérablement la probabilité d'avoir des erreurs pendant l'exécution.

Vu l'impossibilité de définir des protocoles génériques, nous sommes obligés de définir spécifiquement pour toute relation les méthodes permettant l'accès aux champs de cette relation. Cette tâche est très répétitive, et peut être assez longue. Le préprocesseur permet de générer automatiquement la définition de la classe correspondante en connaissant uniquement la liste des champs de la relation et leurs types respectifs. L'exemple donné dans la section suivante permet de constater que nous réalisons ainsi une économie considérable de lignes écrites. La spécification d'une relation de dimension deux nécessite cinq lignes de description et il en génère plus de quarante. Le préprocesseur adopte aussi une convention uniforme de nommage des méthodes d'accès, ce qui permet au programmeur de se familiariser facilement avec le code généré.

3.6 Spécification des relations

Dans un système réparti, il existe plusieurs relations entre les éléments qui constituent le système. La possibilité de spécifier ces relations constitue un élément important d'appréciation du système d'administration. C'est d'ailleurs un des domaines où les modèles classiques n'offrent quasiment aucun support comme nous avons pu le constater dans le chapitre précédent. Les relations qui nous intéressent sont des relations *descriptives*, dans le sens où l'on s'intéresse à décrire les relations déjà existantes. Ces relations peuvent varier avec le temps, il faut donc tenir compte de cette

possibilité de changement.

Les relations qui nous intéressent sont les relations n-aires, telles qu'elles sont utilisées dans les bases de données relationnelles. Les autres formes de relations telles que la relation one-to-many utilisée dans GDMO peuvent être déduites de cette forme. On peut considérer une relation 1 à n comme une relation binaire avec n instances qui ont toutes le même premier élément.

Une relation est représentée par une instance d'une classe distribuée. C'est l'approche que nous avons adoptée pour introduire les extensions (Librairie de classes + Classes abstraites + Macro fonctions). Nous avons donc introduit la classe abstraite «Relationship» dont toutes les classes qui représentent des relations seront dérivées. La description de cette classe est la suivante :

```
distributed class
Relationship :oodeObject
{
  public :
    void getRelationName(char *namebuffer) ;
    int getRelationDim() ;
  private :
    char [MAX_STR] name ;//Nom de la relation
    int symetric ;    //Propriété de symétrie
    int dimension ;   //représente la dimension de la relation
}
```

L'attribut «name» donne le nom que l'on associe à la relation lors de sa création. L'attribut symetric vaut 1 si la relation l'est et 0 si elle ne l'est pas.

A chaque objet administré on associe une liste des instances des relations dont il est membre. On peut alors ajouter et retirer des instances pour exprimer la création et la dissociation ou la destruction d'une relation. L'exemple suivant montre une macro qui définit une relation binaire «connectedto» :

```
RELATION connectedto IS
{
      ManagedObject source ;
      ManagedObject dest ;
}
```

Le code généré par le préprocesseur est décomposé en deux parties : un fichier entête Relconnectedto.H qui contient la description de la classe connectedtoRelationship générée, ainsi que le constructeur de cette classe qui permet de construire des instances de cette relation, et un fichier Relconnectedto.OC qui contient la réalisation de cette classe. On peut noter que le constructeur ajoute une référence vers l'instance créée à chacun des objets constituants de la relation. Le préprocesseur calcule aussi la dimension de la relation, et il lui associe un nom déduit du nom donné lors de la déclaration. Finalement, le préprocesseur génère les méthodes nécessaires pour accéder aux éléments constituants d'une relation (ex : set_source, get_source).

```
// fichier .H
distributed class
```

```
connectedtoRelationship : Relationship
{
 public:

  virtual connectedtoRelationship (ManagedObject *source_param,
   ManagedObject *dest_param)
  {
   dimension = 2;
   strcopy(name, "connected_to");
   source = source_param;
   dest = dest_param;

   source->addRelationship(this);
   dest->addRelationship(this);
  }

  virtual void set_source (ManagedObject *source_param);
  virtual void set_dest   (ManagedObject *dest_param);
  virtual ManagedObject *get_source();
  virtual ManagedObject *get_dest();
 private:
   ManagedObject *source;
   ManagedObject *dest;
}

// fichier .OC
connectedtoRelationship::set_source(ManagedObject *source_param)
{
   // D'abord retirer cette instance de la liste des
   // instances de relation de l'objet source courant
   // avant de le remplacer.
   if (source)
     source->delRelationship(this);
   source = source_param;
   source->addRelationship(this);
};

connectedtoRelationship::set_dest(ManagedObject *source_param)
{
   if (dest)
     dest->delRelationship(this);
   dest = dest_param;
   dest->addRelationship(this);
};

ManagedObject * connectedtoRelationship::get_source()
{
```

```
    return source;
};
ManagedObject * connectedtoRelationship::get_dest()
{
    return dest;
};
```

Nous avons introduit un ensemble de classes prédéfinies dans le but de représenter quelques relations élémentaires qui existent entre certaines entités. Il s'agit des relations suivantes :

Contenance : c'est une relation entre deux objets, un *contenant* et un *contenu*. Cette relation permet de structurer la MIB sous forme d'arbre. L'objet contenant peut contenir plusieurs objets, mais l'objet contenu ne peut être contenu dans plus d'un autre objet.

Résidence : c'est une relation entre deux objets, le premier est un objet administré quelconque, le deuxième est de la classe **host**. Cette relation exprime le fait que la ressource représentée par le premier objet réside sur le site représenté par le deuxième.

Responsabilité : c'est une relation qui associe deux objets, le premier est un objet administré quelconque, le deuxième est de la classe **administrator**. Cette relation permet de déterminer l'administrateur responsable de chaque ressource du système. L'absence de cette information peut causer des situations où les utilisateurs ne savent pas à qui s'adresser quand un problème concernant la ressource se produit.

Expertise : cette relation associe deux objets, le premier est un objet administré quelconque, et l'autre est un objet de la classe **user**. L'expertise peut être comparée à la responsabilité, avec deux différences importantes qui distinguent l'expertise : l'utilisateur peut être un utilisateur normal et pas forcément un administrateur ; en plus, l'expertise est volontaire.

Intérêt : cette relation associe un utilisateur à un objet administré quelconque pour exprimer le fait que cet utilisateur est intéressé par toute évolution qui pourrait se produire concernant la ressource représentée par l'objet.

Dépendance : cette relation associe deux objets administrés quelconques, le premier est dit *dépendant* du second. Cette relation est utilisée pour propager les changements de configuration sur un objet vers tous les objets qui dépendent de lui. Les détails de cette utilisation sont donnés au chapitre 5, consacré à la présentation de notre algorithme de changement dynamique de la configuration.

4 Spécification des interfaces

Les systèmes d'administration que nous avons étudiés dans le chapitre précédent souffrent d'un inconvénient important en ce qui concerne la spécification des

interfaces. Ces systèmes s'intéressent uniquement à l'aspect «administratif» des ressources administrées, et non pas à leur aspect «opérationnel». L'aspect opérationnel recouvre l'ensemble des fonctions et des services offerts par la ressource. Si l'on prend l'exemple d'une imprimante, on considère que les opérations qui consistent à connecter et déconnecter l'imprimante du système, ainsi que les opérations d'abonnement aux événements engendrés par l'objet qui représente l'imprimante sont des opérations d'administration ; ce sont des opérations qui ne dépendent pas du type de l'imprimante. En revanche, les opérations liées à l'impression proprement dite (chargement de police, choix de la résolution, etc.), font partie du fonctionnement de l'imprimante et dépendent du type de l'imprimante. Cette séparation signifie que les objets administrés sont spécifiés et réalisés en utilisant des outils et des représentations différents de ceux utilisés pour mettre en œuvre les services de la ressource représentée. Notre système utilise une représentation unifiée des ressources du système. Cette représentation permet la spécification et la réalisation du code qui met en œuvre les fonctions d'administration en même temps que celui qui met en œuvre les fonctions opérationnelles. De cette façon, les problèmes d'administration sont discutés dès la phase de spécification du système. Ce qui fait que l'administration est considérée comme faisant partie du système et non pas comme une couche qui vient s'ajouter à la fin.

Un objet administré possède donc trois interfaces qui sont décrites dans la figure 3.7 :

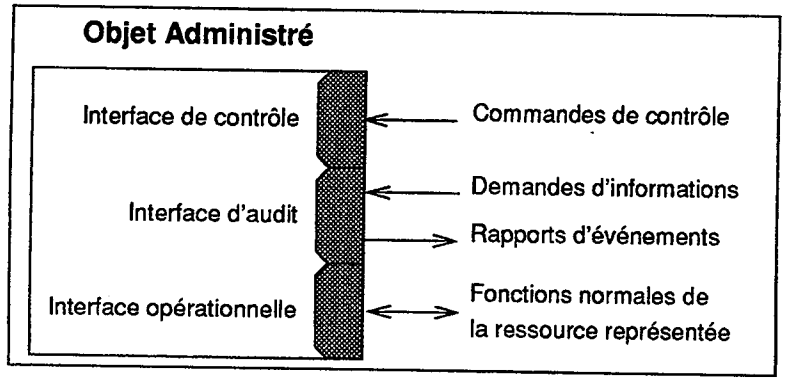

FIG. 3.7 – *Interfaces d'un objet administré*

Interface Opérationnelle : cette interface permet d'accéder aux fonctions fournies par la ressource représentée ; ex : un objet de classe «Printer» fournit la méthode «print». Cette interface dépend entièrement de la ressource représentée.

Interface de Contrôle : cette interface permet d'effectuer des opérations permettant de contrôler le comportement de la ressource représentée. Les opérations de contrôle fournies dans notre système sont l'ajout, le retrait, la connexion et la déconnexion d'une ressource. Elles comprennent aussi les opérations de synchronisation des accès concurrents : acquisition et libération des verrous.

Interface d'audit : cette interface fournit des informations sur le comportement et l'état de la ressource représentée. Ces informations sont fournies sous forme de rapports d'événements.

Les interfaces de contrôle et d'audit constituent ensemble *l'interface d'administration* de l'objet administré. Cette interface est uniforme et héritée par l'ensemble des objets administrés. En ce qui concerne l'interface d'audit, le système d'administration intègre la couche de génération d'événements décrite par Emmanuel Lenormand dans [BLM96].

Interface de contrôle

L'interface de contrôle est constituée des méthodes suivantes :

Add/Remove : Permettent d'ajouter et de retirer des instances d'objets à la MIB. La méthode Add prend en paramètre le nom symbolique de l'objet qui définit aussi sa position dans la MIB.

Connect/Disconnect : Permettent de rendre l'objet accessible/inaccessible aux activités. Une définition plus complète est fournie dans le chapitre 5, avec l'algorithme de gestion dynamique de la configuration qui permet la réalisation correcte des opérations de connexion et de déconnexion.

Lock/UnLock(locktype) : Permettent de poser/libérer des verrous sur un objet administré pour contrôler les accès concurrents. Il existe deux classes de verrous : partagés et exclusifs.

Interface d'audit

L'interface d'audit est basée sur un service de génération d'événements. Un événement est une notification asynchrone émise par un objet au moment où une certaine condition concernant les variables d'état de cet objet est réalisée. Le mécanisme de génération actuelle n'offre aucune spécification formelle pour les conditions de génération des événements. Le programmeur doit donc instrumenter son code manuellement pour engendrer ces événements.

Les événements sont dirigés vers des domaines (unité d'exécution concurrente constituée de plusieurs activités). A chaque domaine on associe un objet *gérant d'événements*. Il s'agit d'une instance de la classe *EventService* qui gère les événements reçus par le domaine. La définition du comportement de cet objet est faite grâce à des *clauses réactives* ; il s'agit d'une section spéciale dans la définition de la classe, introduite par le mot clé REACTIONS et conclue par END_REACTIONS. Une clause est un ensemble de doublets (e,m) où e est le nom d'un événement, et m est la méthode exécutée en réaction à cet événement. Les méthodes sont exécutées par des activitées faisant partie du domaine qui reçoit les événements.

Les méthodes de l'interface d'audit sont :

Subscribe/Unsubscribe(evt) : permet au domaine appelant de recevoir les événements engendrés par l'objet. Le domaine doit avoir au préalable créé un

```
REACTIONS
      CLAUSE clause1 // nom de la clause
          // des clauses réactives définies par
          // un événement et la réaction associée
          ON e1 DO m1 ;
          ON e2 DO m2 ;
      CLAUSE clause2
          ON e3 DO m3 ;
END_REACTION
```

FIG. 3.8 – *Exemple d'un ensemble de clauses réactives*

gérant d'événements qui contient une clause réactive permettant de traiter l'événement.

Enable/Disable(evt) : permet/arrête la génération de l'événement donné en paramètre.

Les événements sont engendrés manuellement grâce à une macro fonction qui a la forme suivante :

```
SEND <nom_evt> INT(0) STR (''STUFF'') OBJECT(this);
```

Le mot-clé SEND indique qu'il s'agit de l'émission d'un événement, dont le nom est passé en paramètre. Ce nom est une chaîne de caractères constante ou une variable en contenant une. Les éléments suivants de cette commande d'émission sont les paramètres de l'événement. INT dénote un paramètre entier, STR un paramètre chaîne de caractères et OBJECT un pointeur distribué. Les paramètres spécifiés sont utilisés comme paramètres d'appel de la méthode appelée lors de la réception de l'événement.

Pour illustrer l'utilisation des événements, nous reprenons l'exemple du problème des philosophes pour montrer la spécification d'un philosophe dans notre modèle. Un philosophe est représenté par une activité qui exécute le code d'un module «philo», et il exécute la boucle classique (réfléchir, récupérer les fourchette, manger, libérer les fourchettes). Pour pouvoir suivre l'évolution de son état, le philosophe engendre des notifications d'événements lors de l'acquisition et de la libération des verrous. Ces notifications seront récupérées par une activité responsable de la surveillance de l'exécution d'une famille de philosophes. La classe philo est dérivée de la classe module que nous avons déjà décrite. La définition de cette classe est la suivante :

```
distributed
class Philo : public Module {

  ManagedObject  *left;
  ManagedObject  *right;
  long id;
public:
  Initinal (long n); // numero du philosophe
  virtual void run();
}

Philo::run()
{
 while (TRUE)
  {
   think();
   left = getLeftFork();
   SEND <gotleftfork> INT(id) OBJECT (this) OBJECT (left);
   right = getRightFork();
   SEND <gotrightfork> INT(id) OBJECT (this) OBJECT (right);
   eat();
   releaseLeftFork();
   SEND <freedleftfork> INT(id) OBJECT (this) OBJECT (left);
   releaseRightFork();
   SEND <freedrightfork> INT(id) OBJECT (this) OBJECT (right);
  }
}
```

5 Conclusion et évaluation

Nous avons présenté dans ce chapitre nos propositions pour la spécification de la configuration. Le modèle proposé est celui d'une représentation à base d'objets. Les objets sont décrits grâce à une extension du langage OC++ fourni par la platforme distribuée OODE. Il s'agit d'un langage qui enrichit C++ en introduisant les notions de persistance et de distribution. Nous avons introduit un mécanisme permettant d'ajouter des extensions supplémentaires à OC++ grâce à un préprocesseur et une bibliothèque de classes abstraites. Cette section conclut le chapitre en donnant une évaluation de l'ensemble des fonctions fournies.

Dans l'absence de tout critère d'évaluation basé sur des performances quantitatives du modèle, l'évaluation que l'on peut faire d'un tel service est largement subjective. Notre évaluation sera basée sur deux points : les atouts du modèle, et le nombre de fonctions fournies.

Même si notre modèle n'introduit pas de nouveaux concepts ou de nouvelle fonctions par rapport à celles introduites dans l'état de l'art, il remplit les objectifs que nous nous sommes fixés. D'un coté, la représentation à base d'objets est largement

reconnue comme la meilleure représentation de haut niveau que l'on peut utiliser pour représenter la configuration d'un système réparti [Gen96]. En effet, la question qui se pose n'est pas d'utiliser une représentation à base d'objets ou pas, mais quelle représentation à base d'objets choisir. La solution que nous proposons représente plusieurs atouts considérables :

- Le langage proposé est très proche de C++, qui est le langage le plus utilisé dans ce domaine. Ceci élimine le besoin d'apprentissage d'une syntaxe lourde et peu répandue en dehors de certains cercles comme celle de ASN.1.

- L'utilisation des objets persistants facilite énormément la mise en œuvre des OA et des applications qui les utilisent. Le programmeur n'a pas à se soucier de gérer les deux représentations des ses données et la conversion entre elles, c'est géré automatiquement par le système.

- La conformité aux spécifications de CORBA, qui est le standard actuel pour les systèmes à objets, nous permet d'espérer que notre modèle ne sera pas remis en cause de façon significative dans l'avenir proche.

- Le mécanisme d'extension permettant d'introduire des fonctions non présentes dans OC++ est simple et très efficace. Nous comptons nous en servir pour introduire d'autres fonctions que nous n'avons pas encore traitées, en particulier celle de la spécification des contraintes d'intégrité.

- La MIB proposée nous permet de couvrir l'ensemble de ressources que nous considérons pour le moment gérer. Le modèle à base d'objets permet une grande souplesse en ce qui concerne l'introduction au système de nouvelles classes de ressources, car il suffirait alors de définir une nouvelle sous-classe d'objets administrés. Les classes que nous avons présentées dans ce chapitre sont toutes dans un état fonctionnel qui permet de développer des applications et des outils d'administrations qui les utilisent. Nous avons d'ailleurs essayé de réutiliser les MIB déjà existantes de SNMP et de CMIP, comme la MIB de l'imprimante définie pour SNMP [SWH+95].

- L'utilisation d'une seule représentation pour les ressources regroupant à la fois les fonctions d'administration et de fonctionnement de la ressource permet une meilleure intégration de l'administration au sein du système. L'administration est vue alors comme une partie intégrante du système et non plus comme un ensemble d'outils ad hoc qui sont ajoutés une fois que le système à été réalisé.

En ce qui concerne le nombre de fonctions, le modèle semble assez complet puisque nous fournissons l'ensemble des fonctions décrites dans le chapitre précédent et qui semble assez exhaustif, à l'exception de deux fonctions : la spécification formelle des conditions de génération d'événements et la spécification des contraintes d'intégrité. Ces deux fonctions sont d'ailleurs étroitement liées. En effet, dans l'absence d'un langage *déclaratif* comme prolog, la seule possibilité qui nous reste pour mettre en œuvre les contraintes d'intégrité est de réaliser une solution à base de détection et de réparation des incohérences. Dans un tel schéma, pour chacun des

objets sur lesquels on veut imposer des contraintes d'intégrité, on définit un ensemble d'événements qui seront engendrés au moment où ces contraintes sont violées. Ces événements seront dirigés vers un domaine «réparateur» qui se chargera de remettre l'objet dans un état cohérent. Une telle solution est décrite dans [Tho94] et nous comptons nous en inspirer dans l'avenir.

4

Gestion des changements

1 Motivation

Comme nous l'avons montré dans le chapitre 1, la seconde tâche du service de gestion de la configuration consiste à gérer les changements qui peuvent avoir lieu pendant la durée de vie du système. Les systèmes répartis sont construits à partir de ressources matérielles et logicielles qui sont physiquement séparées mais qui coopèrent pour réaliser des tâches quelconques. Le système peut subir plusieurs changements d'origines diverses pendant sa vie ; ces changements sont une conséquence naturelle de l'évolution du système. Un système à *configuration statique* tel qu'il est décrit dans la figure 4.1 ne permet pas d'effectuer ces changements de façon acceptable. En effet, pour pouvoir modifier la configuration d'un tel système, il faut l'arrêter et le reconstruire.

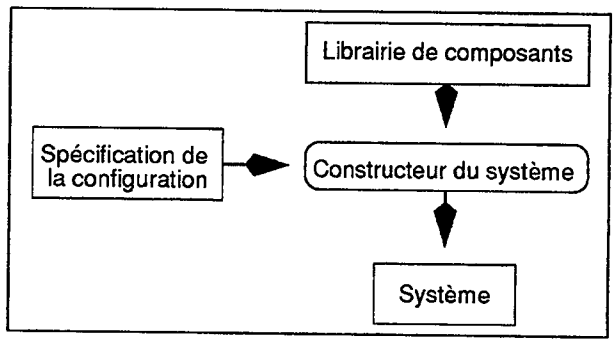

FIG. 4.1 – *Configuration statique d'un système informatique*

Nous avons donc besoin d'un système permettant la *configuration dynamique* afin de réaliser des modifications et des extensions de la configuration sans avoir à reconstruire le système tout entier. La figure 4.2 décrit un tel système, où un système ayant une configuration i évolue vers une configuration $i + 1$. Un fonctionnement correct de cette gestion dynamique de la configuration doit permettre d'effectuer les changements uniquement sur les parties concernées sans perturber le fonctionnement du reste du système. Dans la suite, le terme de *système configurable* désignera un système dont la configuration peut évoluer dynamiquement.

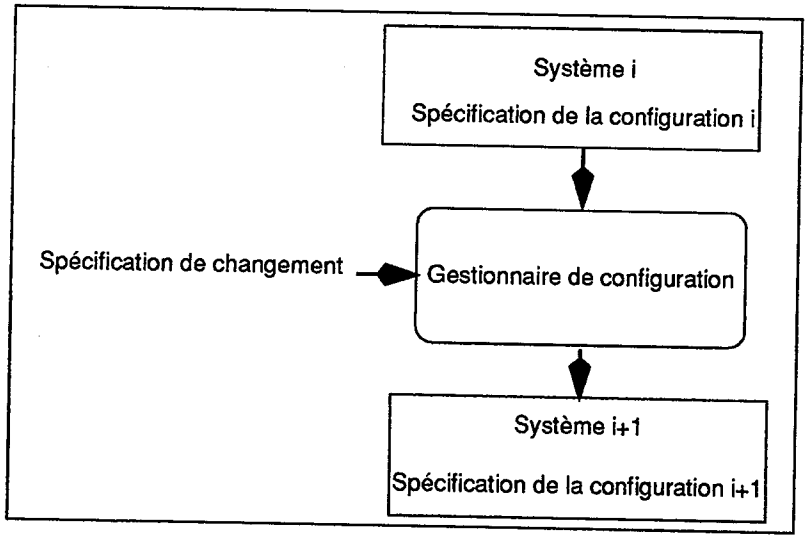

FIG. 4.2 – *Gestion dynamique de la configuration*

Les premiers concepts de la gestion dynamique de la configuration sont apparues avec l'édition de liens dynamique. Ce concept permet de lier les références vers les procédures externes au code de la procédure dans laquelle elles figurent au moment de l'exécution lors du premier appel à ces procédures. Les programmes peuvent alors appeler la version la plus récente d'une procédure stockée dans une bibliothèque de programmes sans avoir besoin d'effectuer une nouvelle édition de liens à chaque fois qu'une nouvelle version est disponible. On peut trouver une description détaillée de ce mécanisme dans [CRO75].

Ce concept n'est plus suffisant pour faire face aux nouvelles exigences. Premièrement, il n'est pas possible de changer les références d'une procédure une fois que ces références ont été liées. Cela ne posait pas de vrais problèmes quand le temps d'exécution des programmes était court relativement au temps de mise à jour des procédures. Mais, c'est de moins en moins vrai avec les systèmes modernes, où certains programmes doivent fonctionner 24h/24 et 7j/7 sur des périodes de plusieurs années. Il reste un deuxième problème plus important qui est celui du manque total de mécanisme de préservation de la cohérence. On n'a aucune garantie sur le comportement d'une application qui exécute une procédure d'une bibliothèque au cas où cette bibliothèque est changée ou retirée pendant l'exécution.

C'est donc principalement pour prendre en compte ces problèmes, que nous avons besoin d'étudier en profondeur le problème de reconfiguration dynamique et de gestion des changements. Ce chapitre sera consacré à l'étude des différents aspects qui nous intéressent dans la gestion des changements. Il s'agit des aspects suivants :

Les catégories de changement. Il existe plusieurs catégories possibles de changement, chacune de ces catégories correspondant à une situation différente. Nous devons déterminer les catégories qui nous intéressent en fonction de nos besoins.

Le grain de changement. Il s'agit de déterminer l'unité de changement dans le

système, de la même façon que l'on s'intéresse à l'unité de concurrence ou de partage des données. Le grain peut être *fin* si l'unité de changement est la procédure ; il peut aussi être *gros* si l'unité est le serveur. Nous allons analyser comment agit ce facteur avant de décider du grain qui nous convient.

Les contraintes concernant la gestion des changements. Quelles sont les contraintes à respecter pour assurer une bonne gestion des changements ? Est-il possible de satisfaire toutes ces contraintes ? Dans le cas d'une réponse négative, quelles sont les priorités à respecter ? Les contraintes dépendent-elles du type des applications qui s'exécutent sur le système ou pas ? C'est à ces questions que nous devons répondre.

Les algorithmes de changement. Les changements doivent s'effectuer sans mettre en cause la cohérence du système. Les algorithmes de changement cherchent à maintenir les conditions permettant des changements *cohérents* sans provoquer de ralentissement du système. Nous présentons les algorithmes utilisés dans les systèmes CONIC [KM90], POLYLITH [HP93], DUS [Hig94], chacun de ces trois algorithmes adoptant une approche différente au problème de maintient de la cohérence.

2 Les catégories de changement

Les sources de changement qui peuvent avoir lieu dans un système réparti sont diverses et variées. Pour mieux les comprendre, il faut les classer ; nous avons pu distinguer quatre catégories de changement :

Changements d'interface : Ce sont des changements de l'interface des objets. Ce type de changement permet de mettre à jour un composant pour fournir de nouvelles fonctions qu'il ne fournissait pas auparavant sous forme d'ajout de méthodes. Il permet aussi de changer l'interface d'accès à des fonctions existantes en modifiant l'interface de méthodes déjà existantes.

Changements de réalisation : La configuration globale du système et les interfaces restent inchangées, mais la réalisation des méthodes est modifiée. Un exemple de ces changements est l'installation d'une nouvelle version d'un logiciel qui corrige d'anciennes bogues ou utilise de meilleurs algorithmes mais sans fournir de nouvelles fonctions.

Changements structurels : Ce sont des changements qui affectent la topologie du système. Les objets peuvent être retirés ou ajoutés pour engendrer une nouvelle configuration (comme c'est le cas avec l'arrêt et/ou le démarrage d'un site).

Changements géométriques : Les objets restent inchangés, mais l'allocation des composants logiciels aux composants matériels est modifiée. Un exemple est la migration des processus.

Pour illustrer ces différents types de changements, on peut considérer le problème des philosophes proposé par Dijkstra [Dij72].

Un exemple de changement d'interface est l'ajout d'une méthode supplémentaire **getNeighbours** à l'interface de la classe Philo permettant de connaître les voisins d'un philosophe. Un changement de réalisation équivaut à changer la classe Philo par une autre version où la boucle réalisée par la méthode «run» fonctionne en mode «verbose», c'est-à-dire dans un mode où les philosophes affichent des messages permettant de connaître l'évolution de l'état de chacun parmi eux.

Un changement structurel équivaut à ajouter et à retirer des activités philosophes au système. Le problème qu'il faut alors résoudre est celui de la redistribution des fourchettes de façon à préserver le fonctionnement correct du programme.

Un changement géométrique équivaut à déplacer l'exécution d'une activité philosophe d'un site vers un autre.

3 Contraintes pour la gestion des changements

Les techniques utilisées pour reconfigurer le système dépendent du domaine des applications qui s'exécutent et des contraintes de performance et de cohérence qui en résultent. Il est clair que les contraintes qui s'imposent dans un environnement de type systèmes de contrôle en temps réel ne sont pas les mêmes que celles que l'on trouve dans un environnement qui gère des bases de données. Il existe cependant plusieurs contraintes qu'un système configurable doit satisfaire, indépendamment de la nature des applications pour lesquelles ce système a été conçu. L'énumération de ces contraintes nous permet dans la suite d'évaluer les modèles que nous allons présenter, ainsi que notre contribution. Ces contraintes sont :

Préserver la cohérence

Un système réparti contient plusieurs applications qui s'exécutent et qui accèdent à ses ressources. La cohérence de ces applications ne doit pas être affectée par les changements qui peuvent avoir lieu pendant leur exécution. Cette contrainte est extrêmement importante car si la cohérence des applications est mise en cause par des changements, les conséquences de ces incohérences peuvent être suffisamment graves pour compromettre l'intérêt de la gestion dynamique de la configuration. La perte de temps et les ressources qui doivent être allouées pour revenir à un fonctionnement correct et réparer les incohérences provoquées par un changement mal géré peuvent dévorer les économies réalisées grâce au passage d'une gestion statique à une gestion dynamique de la configuration.

L'importance de cette contrainte s'accroît du fait que sa satisfaction constitue une base indispensable pour la satisfaction d'autres contraintes, en particulier celle de la réduction de l'intervention humaine décrite plus tard. En effet, si les changements sont effectués sans provoquer d'incohérences, on n'aura pas besoin de l'intervention des administrateurs pour remettre de l'ordre dans le système.

Limiter la dégradation des performances

L'importance de cette contrainte provient du fait qu'il ne s'agit pas uniquement d'une contrainte, mais aussi d'un but de la gestion dynamique des changements. Dans les systèmes statiques, la reconfiguration nécessite l'arrêt du système, ce qui se traduit par une perte de temps et une dégradation considérable des performances. L'un des buts de l'évolution vers une gestion dynamique de la configuration est justement de se débarrasser de cet inconvénient.

De plus, certaines applications peuvent voir leur cohérence mise en cause suite à un ralentissement du système ou à une suspension d'un processus de l'application alors qu'elles ne sont pas concernées par le changement. C'est clairement le cas des applications de type temps réel, mais ce risque apparaît aussi pour d'autres applications. Si l'on considère le cas d'une application client-serveur, le client peut croire à un serveur défaillant alors que le serveur est tout simplement ralenti par une surcharge de la machine où il réside.

Il faut noter que cette notion de «dégradation des performances» est assez large et couvre plusieurs aspects :

1. **Les performances du système pendant le changement.** Il s'agit là de comparer le temps que mettrait le système ayant effectué un changement à exécuter une certaine tâche au temps qu'aurait mis le même système si ce changement n'avait pas eu lieu. C'est le coût direct de la reconfiguration.

2. **La latence introduite indépendamment des changements** Cet aspect semble un peu étrange. Pourrait-on être ralenti par des changements qui n'ont pas lieu? La réponse est malheureusement **oui**. En effet, la conception d'un système configurable impose aux concepteurs des choix qui pourraient se traduire par un ralentissement permanent du système relativement à un système non configurable. Nous devons donc éviter (dans la mesure du possible) de tels choix.

3. **Le temps total nécessaire pour effectuer un changement.** Même si le système n'est pas forcément ralenti par un changement, il est souhaitable que ce temps soit réduit car le système se trouve dans un état transitoire.

Minimiser l'intervention humaine

L'intervention humaine peut être source d'erreurs et de ralentissement. La gestion des changements doit donc être automatisée afin de réduire la nécessité d'une telle intervention.

Il est alors souhaitable que les spécifications de changement soient *déclaratives*; c'est le service de gestion de la configuration qui détermine l'ordre des opérations de changement qu'il faut appliquer et non pas l'administrateur du système.

Garder la transparence

C'est une contrainte qui concerne surtout les programmeurs. Le fait qu'un programme doive s'exécuter sur un système configurable ne doit pas pour autant augmenter sa complexité. Les programmeurs ne doivent donc pas avoir à incorporer des

tests à leur code pour vérifier la disponibilité des ressources. Cependant, on peut accepter qu'il y ait certaines règles de programmation relativement simples qu'il faille respecter. Mais il est hors de question d'obliger les programmeurs à adopter un style de programmation radicalement différent de celui qu'ils utilisent d'habitude.

Réduire les mécanismes de base nécessaires

L'algorithme choisi ne doit pas supposer l'existence de mécanismes qui ne sont pas forcément disponibles, par exemple des protocoles de diffusion comme CBCAST [Ba90]. C'est un problème délicat car l'existence de tels mécanismes peut s'avérer très utile, mais le coût de leur réalisation et de validation peut être excessif. En plus, plusieurs de ces mécanismes posent des problèmes de facteur d'échelle. Dans le cas des protocoles de diffusion, les solutions proposées sont souvent à coût linéaire (n site <=> n messages par diffusion). Un autre exemple concerne l'un des systèmes que nous allons voir, où l'algorithme repose sur un bus de message que l'on peut contrôler pour arrêter la circulation des messages. Il est clair qu'un tel bus est difficile à mettre en œuvre, surtout dans le cas d'un système à grand échelle.

Utiliser des spécifications de changement de haut niveau

Un système réparti est construit à partir d'un ensemble de modules ou de composants (dans notre cas, il s'agit d'un groupe d'objets administrés). Kramer et Magee, les concepteurs du système Conic [KM90] et les précurseurs des travaux dans le domaine des systèmes reconfigurables considèrent que les changements doivent être décrits au niveau des composants élémentaires. L'argument donné est que la spécification de l'évolution à un niveau plus bas serait trop détaillée et n'apporterait pas d'informations claires et compréhensibles sur l'évolution de la structure du système.

La spécification des changements en terme de composants de base du système détermine aussi le *grain* de ces changements.

Valider les changements a priori

Les changements sont soumis au système sous forme de *spécifications de changements*. Un bon algorithme de gestion de changements doit tester dans la mesure du possible si les changements peuvent être accomplis avec succès et s'ils ne violent pas les règles de cohérence du système. Un exemple simple [KA95] est le remplacement d'un objet a par un objet b. Le gestionnaire de configuration doit vérifier que l'objet b possède une interface compatible avec celle de a, sinon le changement doit être refusé.

4 Le grain de changement

Comme nous l'avons dit dans l'introduction de ce chapitre, la détermination du grain de changement est fondamentale car plusieurs contraintes en dépendent. Nous avons distingué trois degrés différents de finesse du grain ; le grain peut être la procédure, le type, ou le serveur [SF93]. Il est clair que la procédure (méthode)

constitue le grain le plus fin parmi les trois niveaux. Ensuite vient le type (classe) qui peut comporter plusieurs procédures d'accès à ses données internes. Le grain le plus gros est le celui du serveur qui contient des procédures faisant partie des interfaces de plusieurs types.

4.1 Le grain est la procédure

Dans les programmes écrits dans un langage procédural comme C, le grain naturel de changement est la procédure. L'édition de liens dynamique décrite au début de ce chapitre constitue une forme primaire de mécanisme de mise en œuvre des changements à un tel grain. Dans un système à objets, le grain comparable est la méthode.

Le système PODUS(Procedure-Oriented Dynamic Updating System) décrit par Frieder et Segal [SF91] a été conçu pour permettre d'effectuer des changements au niveau de la procédure. La mise à jour d'un programme est effectuée en remplaceant les anciennes procédures par de nouvelles pendant l'exécution du programme. Pour accepter ces changements, le programme est instrumenté lors de la compilation. Le compilateur produit pour chaque procédure la liste des procédures qu'elle peut appeler sous forme de graphe d'appels.

Pendant l'exécution d'un programme, l'utilisateur demande que la nouvelle version de l'une de ses procédures soit chargée dans la mémoire. Une fois que cette version est chargée, l'utilisateur envoie une demande de mise à jour. Le système interrompt alors le programme pour vérifier l'état de sa pile d'exécution. L'analyse de la pile et du graphe d'appels produit par le compilateur permet de déterminer si une procédure est active ou non, seules les procédures non-actives peuvent être remplacées. Une fois que la procédure devient inactive elle peut être remplacée ; le système modifie alors la table des adresses de procédures associée au programme pour substituer à l'ancienne l'adresse où la nouvelle version est chargée. Si la procédure contient des données statiques, l'utilisateur doit fournir un module de conversion permettant de convertir les données de l'ancienne représentation vers la nouvelle.

4.2 Le grain est le type

Le deuxième niveau de finesse de grain permet des changements au niveau des types définis par les utilisateurs (en ce qui concerne notre système, un type correspond à une classe). L'idée présentée dans [Sha76] cherche à exploiter le fait que l'on peut remplacer la réalisation d'un type d'une façon transparente si l'on garde la même interface. On cherche alors à pouvoir le faire d'une façon dynamique et pendant l'exécution des programmes qui utilisent des instances de ce type.

Un prototype d'un tel système est le Data Type Replacement (DTR) réalisé par Robert Fabry [Fab76] sur un système qui utilise un schéma d'adressage basé sur les capacités. Un type est décrit par un ensemble de procédures appelé *gestionnaire de type* (correspond à l'interface d'une classe). Quand un type est changé, la nouvelle représentation des données est différente de l'ancienne, et les instances de l'ancienne version ne peuvent plus fonctionner correctement avec le nouveau gestionnaire. Pour résoudre ce problème, un indicateur de version est utilisé pour estampiller chaque

instance d'un type. Le gestionnaire peut alors distinguer les anciennes instances et appeler les procédures de conversion nécessaires.

Le problème avec les changements de type est que l'on ne peut changer que la *réalisation* des procédures d'accès, et non *l'interface* de ces procédures, sans s'exposer à des problèmes compliqués de compatibilité d'interface. Dans notre système, un changement de type équivaut à un changement de classe d'objets. Il nous est en effet difficile de permettre le changement de la signature des méthodes d'accès.

4.3 Le grain est un serveur

Les systèmes basés sur le modèle client-serveur sont similaires aux systèmes permettant un remplacement dynamique des types, car on peut voir le gestionnaire de type comme un serveur spécial. Les deux approches sont quand même différentes en ce qui concerne le grain des objets qui sont modifiés. Un type abstrait a un grain plus fin que celui d'un service, qui est souvent construit à partir de plusieurs types abstraits.

Le système Argus [Lis88] est un bon exemple d'un système où l'unité de changement est le serveur. Argus est un langage et un système d'exploitation qui ont été conçus pour faciliter la réalisation des applications transactionnelles. Un programme Argus est constitué d'un ensemble de serveurs appelés *gardiens* qui réalisent un ensemble logique de fonctions. Les communications entre les serveurs se font par échange de messages.

Pour effectuer les changements, le système utilise le mécanisme de réparation d'erreur fourni par le système transactionnel intégré. Argus permet de maintenir des version cohérentes des données persistantes qui résistent aux pannes du système. Les changements dans Argus s'effectuent sur un ensemble de serveurs, appelé *sous-système*. L'utilisateur simule une panne des serveurs du sous-système à changer, et ensuite effectue un traitement d'erreur en démarrant les nouvelles versions des serveurs.

L'inconvénient de ce modèle provient essentiellement du fait qu'il est très lié au système d'exploitation, surtout au mécanisme de recouvrement d'erreur qu'il fournit, et qui est rarement disponible dans d'autres systèmes.

5 Les algorithmes de changement

5.1 Le système CONIC

Conic [KM85, KM90] est un système qui à été conçu à l'Imperial college ; c'est le précurseur des systèmes configurables. Le système est construit à partir d'un ensemble de *nœuds*, un nœud étant l'unité d'exécution (l'équivalent d'un processus). Les nœuds communiquent par échange de messages à travers des connexions asymétriques. A chaque nœud est associé un ensemble de ports qui lui permettent de recevoir des messages. Les communications se font par *sessions*[2]. Une session est

2. L'article original utilise le terme *transactions* que nous avons préféré éviter pour ne pas confondre avec les transactions classiques utilisées dans les bases de données

un échange d'informations entre **deux et uniquement deux** nœuds ; elle consiste en une séquence de messages échangés entre deux nœuds. Les propriétés qui vont suivre ont été démontrées pour un système à sessions *indépendantes.* Une session indépendante est une session dont la terminaison ne dépend pas d'une autre session. Si l'on veut faire l'analogie avec un modèle client-serveur (un RPC est une session particulière), on dit que le système ne possède que des serveurs *terminaux* (un processus ne peut être à la fois serveur et client).

Les nœuds sont indépendants de la configuration : il n'y a pas de désignation directe des autres modules, mais uniquement des envois et des réceptions de messages sur les ports du nœud. Les connexions sont gérées par un *gestionnaire de configuration* externe aux nœuds ; ce qui constitue alors une séparation totale entre l'application elle même et sa configuration topologique.

Conic offre un très bon support pour les changements structurels. Les changements possibles sont l'ajout **create** et le retrait **remove** des nœuds, ainsi que l'établissement (**link**) et la coupure (**unlink**) des connexions. Le problème essentiel auquel s'attaque Conic est celui de la préservation de la cohérence ; pour ce faire, l'algorithme utilisé cherche à éviter totalement les incohérences.

Comme nous nous sommes largement inspirés de ce modèle, et que dans la suite de notre rapport nous comparons notre propre algorithme à celui de Conic, nous faisons une présentation assez détaillée de l'algorithme utilisé dans Conic dans la suite de cette section.

5.1.1 Les états de configuration d'un nœud

La notion d'état de configuration d'un nœud a été introduite afin de donner suffisamment d'informations sur le comportement de l'application. Ces informations seront utilisées par le gestionnaire de configuration pour décider du moment du changement. Ces états sont les suivants :

État actif : le nœud peut démarrer, accepter et exécuter des sessions.

État passif : le nœud doit continuer à accepter et exécuter des sessions, mais il n'est pas en train de servir une session qu'il a démarrée, et il ne démarrera plus de sessions.

État dormant : un nœud est dans un tel état s'il remplit les conditions de l'état passif, si en plus il n'est pas en train d'exécuter de session, et finalement s'il ne recoit plus de demandes d'exécution de sessions.

La figure 5.1.1 décrit l'ensemble des états de configuration, ainsi que les transitions permettant le passage entre ces états, ces transitions correspondant à des commandes envoyées par le gestionnaire de la configuration.

Pour un nœud dans l'état dormant, son état est **cohérent** et **gelé**. Il est cohérent car il ne contient pas les résultats d'une session qui ne s'est pas encore terminée, et gelé car il ne changera pas puisque le nœud ne recevra plus de demandes de session. L'idée de base de l'algorithme est de diriger le nœud dont la configuration va changer vers l'état dormant.

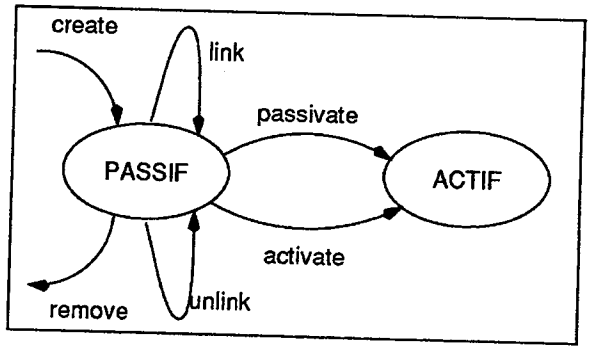

FIG. 4.3 – *États de configuration dans le système Conic*

L'obtention de l'état dormant pour un nœud Q passe par deux étapes. On définit d'abord l'ensemble passif de Q, noté PS(Q), comme l'ensemble des nœuds qui ont des connexions dans la direction de Q (et qui peuvent donc envoyer vers Q des demandes d'exécution de sessions), plus Q lui-même. On peut prouver que Q est dans l'état dormant si tous les nœuds de PS(Q) sont dans l'état passif. On peut aussi prouver que le passage de l'état actif à. l'état passif peut être fait en un temps fini, et de même pour passer vers l'état dormant.

5.1.2 Algorithme de changement

Maintenant que nous avons défini les concepts de base utilisés dans Conic, on peut décrire l'algorithme de changement. L'algorithme se décompose en quatre étapes :

1. Déterminer l'ensemble des nœuds à mettre dans l'état dormant. Cet ensemble dépend du type de changement à effectuer, mais nous allons considérer le cas le plus sensible, celui du retrait. Dans ce cas, cet ensemble comporte le nœud Q, ainsi que tous les nœuds qui sont connectés à Q. L'ensemble ainsi obtenu est noté QS(Q) (Quiescent Set).

2. A partir de l'ensemble QS, il faut déterminer l'ensemble des nœuds qu'il faut mettre dans l'état passif pour que les nœuds de QS passent dans l'état dormant. Cet ensemble contient tous les nœuds du système qui ont une connexion à un nœud de QS. L'ensemble obtenu est noté CPS(Q) (Change Passive Set).

3. Procéder aux changement dans l'ordre suivant :

   - Envoyer l'ordre passivate à tous le nœuds de l'ensemble CPS.
   - Déconnecter les nœuds de QS(Q) de Q.
   - Enlever Q.

4. Activer les nœuds de CPS à l'exception de Q qui a été retiré.

Cet algorithme impose certaines contraintes aux programmeurs des applications. Il faut s'assurer que les nœuds répondent correctement aux directives de changement

comme *passivate*. Cette contrainte constitue une charge pour les programmeurs et réduit la transparence des changements.

L'*exactitude* de l'algorithme dépend des applications qui s'exécutent sur le système. Les preuves qui sont fournies pour les propriétés données sont uniquement vraies dans le cas des applications à sessions indépendantes, elles ne le sont pas dans le cas contraire. Les auteurs de l'algorithme ont étendu ces propriétés aux systèmes à sessions dites *conséquentes*, qui constituent une forme particulière des sessions dépendantes. Une session A est *conséquente* d'une session B si la terminaison de B dépend de celle de A. Pour généraliser les propriétés démontrées aux sessions conséquentes, la définition de l'état passif a été généralisée. Un nœud dans l'état *passif généralisé* doit accepter les demandes d'exécution des sessions et démarrer les sessions conséquentes, mais il n'est pas en train d'exécuter une session non-conséquente qu'il a démarrée, et il ne démarrera aucune session non-conséquente. Les propositions qui ont été démontrées restent valides en changeant l'état passif par l'état passif généralisé. Cette généralisation a quand même un coût non négligeable, puisqu'il s'agit d'agrandir considérablement l'ensemble passif. L'*ensemble passif agrandi* (Enlarged Passive Set) défini pour traiter le cas des sessions conséquentes comporte en plus des nœuds de PS(Q), tous les nœuds qui peuvent démarrer des sessions qui pourront avoir des sessions conséquentes sur Q. Vu que la relation de *conséquence* est transitive, l'ensemble EPS peut alors être très grand.

Les *performances* de cet algorithme dépendent largement du nombre de nœuds qui doivent être mis dans l'état dormant (qui revient à suspendre tous les nœuds qui risquent de communiquer avec eux). Ce nombre est caractérisé par le cardinal de l'ensemble passif CPS, qui est assez grand dans le cas des sessions indépendantes, et qui croit exponentiellement avec le nombre de connexions dans le cas des sessions conséquentes. Ce problème de performances est assez délicat et réduit considérablement l'intérêt de l'algorithme. L'algorithme fonctionnerait bien dans un système de type client-serveur où un serveur ne peut pas devenir client pendant qu'il exécute une demande de service. C'est uniquement dans de tels cas que l'ensemble de nœuds à mettre dans l'état passif est facile à calculer, car il suffit d'analyser les connexions entre les clients et les serveurs.

Malgré ces limitations, cet algorithme possède un avantage non négligeable. Le fait que les incohérences soient totalement évitées dispense le système de fournir des mécanismes de recouvrement et de traitement des erreurs dûes aux changements. En plus, l'algorithme est facile à mettre en œuvre sur plusieurs environnements puisqu'il ne nécessite aucun support particulier du système d'exploitation sous-jacent.

5.2 Le système POLYLITH

Le système POLYLITH [HP93, PH91, HWP93] est un environnement qui offre aux utilisateurs la possibilité de construire des applications distribuées pour les systèmes hétérogènes. Le système est construit à base de modules (processus) qui communiquent exclusivement par messages (pas de mémoire partagée entre les modules). Les messages sont envoyés au moyen d'un bus de messages. Le bus permet d'assurer que **toutes** les communications entre les processus peuvent être contrôlées par un agent externe. Les modules ne peuvent communiquer que par des canaux

privés de communication. Le système s'intéresse aux changements structurels, il permet le remplacement des modules pendant leur exécution.

Les concepteurs de POLYLITH partent du principe que la participation d'un module au processus de reconfiguration peut s'avérer nécessaire. Cette approche est contraire à celle de Conic qui considère que les changements de configuration doivent être indépendants des algorithmes, des protocoles et de l'état d'exécution des applications. La participation des modules peut être demandée sous trois formes :

- Un module peut devoir fournir son état d'exécution. Cet état sera utilisé pour initialiser un module de remplacement après la reconfiguration.

- On peut avoir besoin d'initialiser un module quand il est créé dynamiquement. Pendant le remplacement, le nouveau module doit être initialisé à partir de l'état de l'ancien module.

- Au lieu de permettre la reconfiguration à n'importe quel moment, un module peut avoir besoin de retarder la reconfiguration en attendant que le module atteigne un point de reconfiguration où la cohérence de l'application à laquelle il participe peut être préservée.

5.2.1 Les types abstraits des processus

L'approche adoptée par POLYLITH pour la reconfiguration des processus est basée sur la représentation de ces processus sous forme de types abstraits. Chaque module définit un type abstrait, et chaque processus qui s'exécute est décrit par une instance de ce type. Les structures d'exécution d'un processus déterminent la valeur de cette instance. L'état peut être extrait pendant l'exécution au moyen d'une fonction spéciale qui fournit une représentation de cet état. Les structures qui constituent l'état d'un processus de POLYLITH sont les suivantes :

- Le compteur ordinal du programme

- Le segment de données qui contient les données statiques du programme.

- La pile d'exécution, qui contient l'ensemble des variables locales et temporaires, ainsi que les adresses de retours des fonctions appelées, et les copies de sauvegarde des registres utilisés.

- Les données dynamiques allouées dans le tas.

- Les descripteurs de fichiers, et les informations concernant l'état du processus relativement au noyau.

Cette liste de structures suppose un langage à portée lexicale statique et des modules à un seul flot d'exécution. Le schéma de préparation d'un module est décrit dans la figure 4.4.

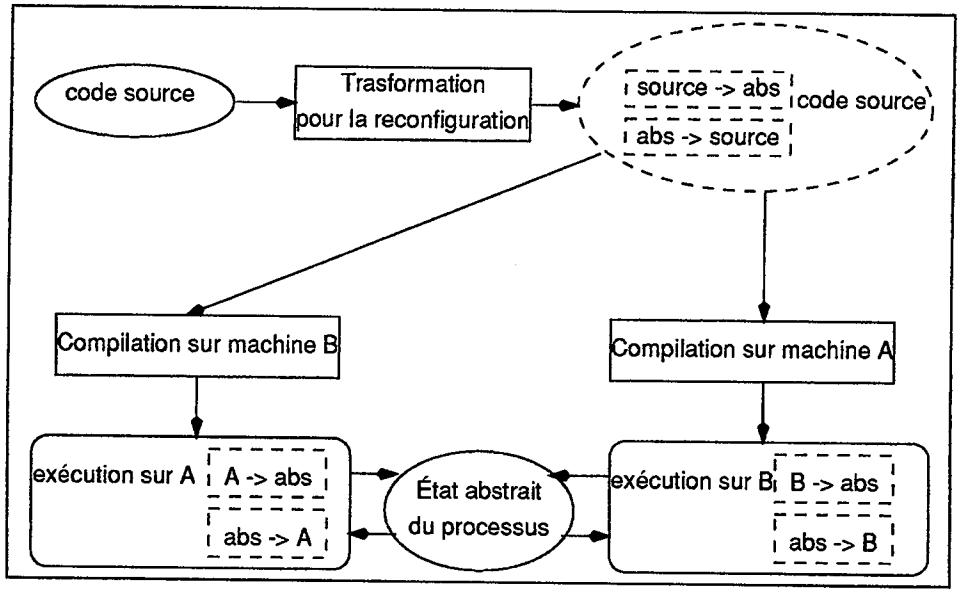

FIG. 4.4 – *Récupération et restauration de l'état d'un processus dans POLYLITH*

5.2.2 L'algorithme de remplacement

L'algorithme de remplacement est assez simple. Il se décompose en quatre étapes :

1. Attendre que le module à remplacer atteigne un état configurable.

2. Bloquer les messages sur tous les ports de communication du module en question. Ce blocage est rendu possible grâce au bus de message.

3. Copier l'état du module dans une copie de sauvegarde.

4. Créer un nouveau module de remplacement et l'initialiser depuis la copie de sauvegarde.

Il est clair que cet algorithme possède un coût moins élevé que celui de Conic, car il ne nécessite pas la suspension d'autres modules que le module concerné. Mais il introduit de nouveaux problèmes qui ne sont pas moins importants :

- Le gestionnaire de la configuration a besoin d'accéder à l'état du processus afin de pouvoir le copier. Le programme doit donc définir certains points où il est possible de copier cet état. Il s'agit de ces fameux point de reconfiguration. Ceci supprime la transparence pour le programmeur.

- Le système est incapable de copier toutes les informations constituant l'état du module. En particulier, les descripteurs de fichiers et les informations spécifiques au noyau (état du processus, temps d'exécution, .etc) ne sont pas copiés parce qu'ils ne sont pas transférables entre systèmes différents. C'est donc une limitation de l'approche elle même, plutôt que de la mise en œuvre.

- Les changements sont autorisés uniquement pendant que le système est dans un état reconfigurable. Mais cet état reconfigurable n'est pas facilement accessible car il nécessite l'arrêt des communications sur les ports du module concerné, ce qui peut se traduire par une longue attente.

- L'algorithme nécessite un mécanisme spécial qui est celui d'un bus de messages. Ce mécanisme n'est pas forcément disponible dans les systèmes d'exploitation, et sa réalisation peut être coûteuse.

- Le système permet un seul type de changement : le remplacement de modules. Le remplacement n'est pas un changement de base dans le sens qu'il existe d'autres formes de changements structurels que l'on ne peut pas obtenir à partir du remplacement. En particulier, l'ajout et le retrait ne peuvent être construits à partir du remplacement, alors qu'on peut construire le remplacement comme un retrait suivi par un ajout.

- Pour pouvoir copier l'état d'un processus, les concepteurs font des restrictions assez sévères sur le système d'exploitation de base. En particulier, les modules doivent être à un seul flot d'exécution, ce qui est difficilement acceptable pour un vrai système réparti.

- Le système n'offre pas de support aux communications synchrones, ce qui ne permet pas de l'utiliser pour des applications de type client-serveur. La raison est la suivante : supposons qu'un client A envoie un message au serveur B puis attende la réponse (schéma classique d'un RPC), il est possible qu'entre temps A ait été remplacé par une autre version du client qui ne reconnait pas le RPC qui était utilisé par l'ancien module, d'où un problème d'incohérence.

La majorité de ces problèmes sont dûs à l'approche consistant à copier l'état et non pas à ce système particulier.

5.3 Le système DUS

Les deux algorithmes que nous venons de décrire constituent une approche *pessimiste* de la gestion dynamique. Le troisième algorithme de changement est celui utilisé dans le système DUS (Dynamic Updating System [Hig94, HH96]) adopte une approche plutôt *optimiste*. L'idée de base est de laisser le système évoluer sans contraintes, en espérant qu'il n'y aura pas d'incohérences. Si le système détecte des incohérences dûes aux changements, il tente de les corriger grâce à un mécanisme transactionnel. Les étapes de l'algorithme décrits dans la figure 4.5 sont les suivantes :

1. Quand un processus doit être remplacé, une nouvelle version du processus est créée et commence son exécution. L'ancienne version du processus n'est pas arrêtée et continue son exécution en même temps que la nouvelle version. Les différentes version du même processus constituent alors *un groupe de processus.*

2. Les messages sont envoyés à tous les membres du groupe. Le protocole utilisé garantit que tous les processus reçoivent les messages dans le même ordre. Ceci

garantit une trace d'exécution identique pour les différentes versions du même processus.

3. L'ancienne version du processus détermine un point de sauvegarde approprié et suspend son exécution.

4. Si on observe des incohérences, la nouvelle version du processus est avortée. Une procédure d'erreur permet à l'ancienne version de recommencer son exécution à partir du point de sauvegarde.

5. Si toutes les nouvelles versions des processus sont créées sans provoquer d'incohérences, toutes les anciennes versions sont arrêtées et l'opération de changement se termine avec succès.

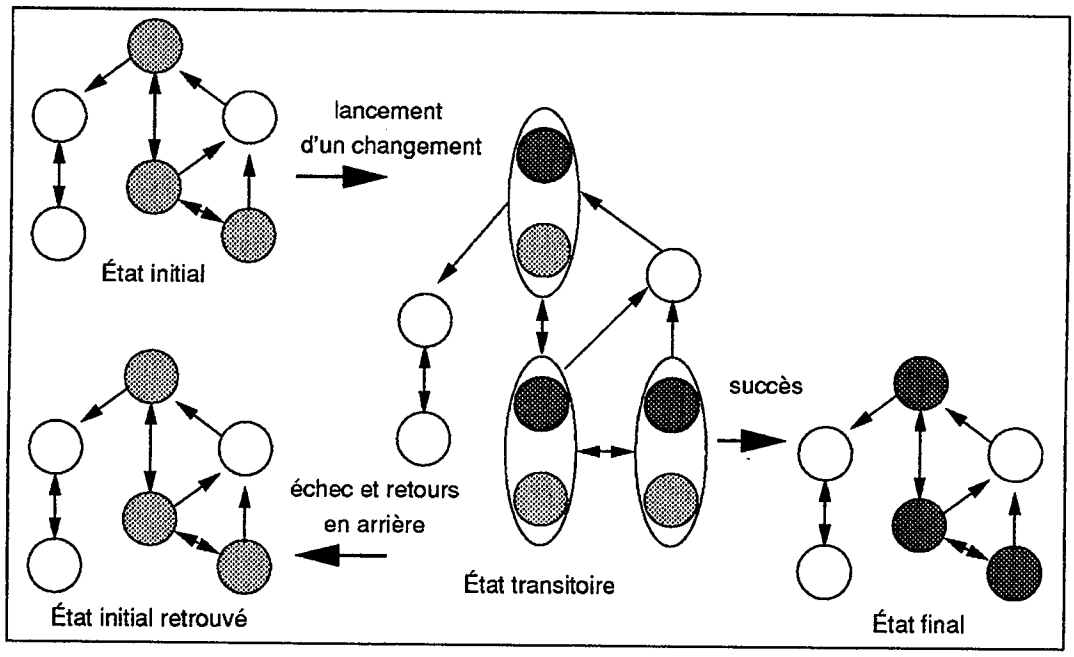

FIG. 4.5 – *Approche optimiste de la configuration dynamique*

Cet algorithme a le mérite de sortir de l'idée classique de Conic qui a été systématiquement reprise par les concepteurs des systèmes configurables, à savoir qu'il était nécessaire ET suffisant de geler le fonctionnement d'une certaine partie du système pour pouvoir procéder aux changements. Aucune suspension n'est nécessaire, ce qui signifie que l'inconvénient majeur de Conic disparaît ici. De même pour l'exactitude de l'algorithme, qui n'est pas réservée à une classe particulière d'applications. Ceci dit, cet algorithme n'est pas forcément le mieux adapté à nos besoins à cause des problèmes suivants :

- L'algorithme nécessite des fonctions spéciales qui ne sont pas disponibles dans la plupart des systèmes d'exploitation. En particulier, l'algorithme a besoin d'un protocole de communication inter-groupes fiable et ordonné, et d'un mécanisme de retour en arrière.

- Le schéma de duplication active utilisé pour fournir un ensemble de versions différentes du même processus ne peut pas être utilisé par toutes les applications. Pour pouvoir utiliser de tels schémas de duplication correctement, il faut aussi dupliquer les données auxquelles ces processus ont accès.

- L'algorithme ne traite pas le cas de l'effet domino [Had88], où pour pouvoir effectuer le retour arrière d'un procesus A on est obligé d'arrêter un processus B qui a lu des valeurs qui ont été écrites par A depuis son dernier point de sauvegarde. Ce processus doit donc effectuer un retour en arrière à son tour, et le même problème se pose de manière récursive. On peut finalement se trouver avec un grand nombre de processus avortés.

5.4 Conclusion

Nous avons présenté dans ce chapitre les aspects qui caractérisent un système à configuration dynamique et que nous rappelons ci-dessous :

- La catégorie de changements. Il s'agit du type de changements possibles. On a identifié quatre types de changements : les changements d'interface, les changements structurels, les changements de réalisation et les changements géométriques.

- Le grain de changement. Le grain est l'unité que le système modifie ou remplace, le grain peut être petit(procédure) moyen (objet ou classe d'objets) et gros (serveur contenant plusieurs objets de plusieurs classes).

- Les contraintes que doit respecter le système de gestion de changements. Il s'agit de préserver la cohérence des applications qui s'exécutent, de limiter la dégradation des performances, de minimiser l'intervention humaine qui est souvent source d'erreurs, de garder la transparence vis-à-vis du code des programmes pour encourager les programmeurs à utiliser notre système, de réduire les mécanismes de base nécessaires à l'algorithme de changement, d'utiliser des spécifications de changement de haut niveau, de valider les changements avant de les effectuer pour s'assurer qu'ils ne violent pas la cohérence du système.

Pour illustrer ces aspects, nous avons présenté les différentes approches d'un bon nombre de systèmes : Argus, Conic, Data Type Replacement (DTR), Polylith, et le Dynamic Updating System (DUS). Nous avons aussi donné une présentation détaillée des algorithmes de changement utilisés dans Conic, Polylith et DUS. Les deux premiers systèmes, Conic et Polylith adoptent une approche pessimiste, et le troisième adopte une approche optimiste. Le tableau suivant résume les points essentiels de notre comparaison. Notons toutefois que les lignes concernant les performances des systèmes sont basées sur une évaluation *subjective* des algorithmes et de leur complexité théorique. Pour la plupart des systèmes étudiés, aucun résultat expérimental n'est donné dans les références, ce qui rend difficile l'évaluation de leurs performances, ainsi que la comparaison de nos résultats avec les leurs.

Systèmes	**Argus**	**Conic**	**DTR**	**Polylith**	**DUS**
Créateur	MIT	Imperial College	Robert Fabry	Univ. Maryland	H. Higaki
Type du système modifié	Client-serveur	Modules communicants	Types abstraits	Processus communicant	Processus communicants
Mécanismes spéciaux nécessaires	env. d'exéc. Argus	env. d'exéc. Conic	OS basé sur les capacités	bus de messages	comms. inter-groupes et roll-back
Langages spéciaux	Argus	Conic	tout lang. procédural	C + macros Polylith	inconnu
Distribution	Oui	oui	non	oui	oui
Comms. Distribuées	RPCs	messages	-	messages	messages
Grain	Serveur	module (instance d'une classe)	type abstrait (classe)	processus	processus
Degré d'intervention humaine	inconnu	réduit	inconnu	réduit	réduit
Ralentissement	moyenne	très grande	réduit	grande	réduite
Temps d'exécution d'un changement	court à moyen	grand	pas claire	moyen	réduit
Changements possibles	Réalisation d'un serveur	ajout-retrait d'un module	changement de type	Remplace. et migration de processus	remplacement de processus

5

Propositions pour la gestion des changements

Ce chapitre est consacré à la présentation de la solution que nous proposons pour la gestion dynamique de la configuration. Pour décrire cette solution, nous suivons le schéma adopté dans le chapitre précédent. Nous commençons donc par présenter nos choix pour le type et le grain de changement. Ensuite, nous donnons une description du modèle de notre système, et des extensions introduites spécifiquement pour permettre la reconfiguration du système. Nous décrivons alors notre algorithme de gestion de changement. Finalement, nous décrivons sur un exemple le fonctionnement de l'algorithme.

1 Introduction

Le chapitre précédent nous a permis de voir différents algorithmes de changement et d'évaluer les solutions possibles au problème du maintien de la cohérence d'un système reconfigurable. La comparaison de ces algorithmes et l'analyse de nos propres besoins permettent de dégager les principes sur lesquels notre algorithme est basé. Il s'agit des points suivants :

- Il est possible de corriger les incohérences en utilisant des techniques classiques telles que les points de reprise et les mécanismes de recouvrement. Mais c'est une solution coûteuse car elle se traduit par une perte considérable de performance au cas où il y a beaucoup de recouvrements. On doit craindre aussi l'effet domino qui peut se produire.

- Les algorithmes qui cherchent à éviter totalement les incohérences souffrent de deux problèmes : d'abord ils ne couvrent qu'un petit ensemble d'applications distribuées ; deuxièmement ils ont un très grand coût en termes de performances car le nombre de processus à suspendre est très grand.

- Il n'est pas déraisonnable de demander aux programmeurs un minimum de coopération. Ils doivent respecter certaines *consignes de programmation* qui ne doivent pas avoir de conséquences importantes sur leur style de programmation. Ces consignes permettent de développer des applications qui sont capables

de fonctionner dans un environnement qui évolue, sans pour autant être obligés de tester à chaque appel d'un objet distribué si la ressource sous-jacente est toujours là . Les programmes qui ne respectent pas ces consignes doivent quand même être capable de fonctionner, mais sans aucune garantie concernant leur comportement en cas de reconfiguration du système.

Nous allons dans la suite de ce chapitre proposer un algorithme dont l'idée intuitive déduite des points précédents est la suivante :

1. Nous utilisons les demandes de verrous utilisées initialement pour la protection des accès concurrents comme des points de contrôle permettant de vérifier la disponibilité des objets.

2. Nous cherchons à réduire au maximum le nombre des incohérences qui peuvent se produire. Nous proposons donc une classe d'applications pour lesquelles il est possible d'éviter les incohérences. Cette classe est définie par une politique de verrouillage.

3. Pour les applications qui ne respectent pas un schéma particulier de verrouillage, il y a un risque d'incohérence qui apparaît quand une activité demande l'allocation d'un verrou sur un objet déconnecté. L'application provoque alors une exception pour demander au système d'essayer de trouver un objet de remplacement pour permettre à cette activité de continuer son exécution.

4. Dans le cas où le système n'arrive pas à trouver un remplacement, l'activité ayant provoqué cette exception effectue un retour en arrière pour tenter de reprendre son exécution.

2 Catégorie et grain de changements

Les catégories de changement que nous avons identifiées dans le chapitre précédent sont les changements d'interface, les changements de réalisation, les changements structurels, et les changements géométriques. Il n'est pas nécessaire d'offrir un support à tous ces types de changement. Pour le moment, les changements d'interfaces qui consistent à modifier les signatures des méthodes sont trop compliqués pour nous car ils nécessitent la recompilation des applications qui communiquent avec l'objet modifié. En revanche, l'autre partie des changements d'interface concernant l'introduction de nouvelles méthodes peut être partiellement permise grâce aux mécanismes d'héritage et de conformité d'interface fournis par OC++.

Plusieurs modèles considèrent que les changements structurels sont les plus importants car la plupart des autres changements peuvent être construits à partir d'eux. En effet, les changements de réalisation peuvent être considérés comme un cas particulier des changements structurels, où l'objet avec l'ancienne réalisation est retiré et celui avec la nouvelle réalisation est ajouté. Le même argument peut s'appliquer aux changements géométriques ; les objets correspondant aux entités logicielles à réorganiser sont retirés et puis réintroduits avec des paramètres d'initialisation différents qui reflètent la nouvelle correspondance.

Nous nous intéressons à la reconfiguration des systèmes à des fins d'administration. Les évolutions dans ce domaine sont plutôt à grain moyen car elles concernent les ressources physiques. Le grain qui nous convient correspond alors à un objet administré. On voit mal l'intérêt d'avoir un grain plus petit (au niveau des méthodes). Ce choix est conforme aux critères concernant le niveau des spécifications de changement, puisqu'il nous permet de spécifier les changements en utilisant des méthodes faisant partie de l'interface de contrôle des objets administrés. Ainsi, pour retirer un objet du système, on doit appeler la méthode **remove** sur cet objet.

3 Modèle du système

Nous donnons ici les définitions de base qui nous sont nécessaires dans la suite. Certaines de ces définitions ont déjà été introduites dans les chapitres précédents, et sont données à titre de rappel.

Objet administré : il s'agit d'une représentation de haut niveau d'une ressource du système, l'objet est identifié par un identificateur unique. A chaque objet est associée une interface qui constitue l'unique moyen pour les autres objets de communiquer avec la ressource sous-jacente. Les objets sont décomposés en deux grandes catégories : les objets actifs qui représentent les flots d'exécution et les objets passifs qui représentent les entités physiques.

Système : un système est défini comme un ensemble d'objets administrés qui représentent l'ensemble de ses ressources. L'ensemble d'objets administrés dans un système est appelé la base de données administratives ou MIB.

Verrous : les verrous sont utilisés pour synchroniser l'accès aux objets administrés et préserver ainsi la cohérence de leurs états persistants. Avant d'appeler une méthode sur un objet, l'activité appelante doit acquérir le verrou nécessaire. Il existe deux types de verrous : exclusif et partagé.

Connexion : on dit qu'un objet est *connecté* au système s'il accepte les appels sur ses méthodes.

Déconnexion : on dit qu'un objet est *déconnecté* du système s'il n'accepte pas d'appels sur ses méthodes.

4 Contexte d'un objet administré

Les notions de connexion et déconnexion au système ont été introduites pour représenter l'état de configuration des ressources qui sont présentes, mais momentanément inaccessibles. De telles ressources n'ont pas été retirées du système, mais elles ne peuvent par être traitées comme si elles étaient disponibles.

Notre but est de pouvoir changer la configuration du système à travers des changements structurels, comme l'ajout et le retrait des éléments. Ces changements posent un problème concernant les pointeurs vers des objets qui sont déconnectés ou

qui ont été complètement retirés du système. Supposons en effet qu'un objet distribué o encapsule dans son état persistant un pointeur vers un objet distribué x et que x est retiré du système. Lorsque o va essayer d'accéder à x, il y aura un problème et si le programmeur qui réalise o ne prend pas les précautions nécessaires et il y aura des conséquences indésirables. Il ne suffit d'ailleurs pas que le programmeur fasse des tests pour vérifier si l'objet est disponible ou pas, il faut aussi qu'il puisse trouver un remplaçant éventuel et changer le pointeur. Il est clair que tous ces tests constituent un surcroît de travail pour le programmeur ce qui est en contradiction avec la contrainte de transparence dont nous avons parlé au chapitre précédent.

Pour résoudre ce problème, nous avons décidé de définir certaines règles de programmation sans pour autant les imposer. C'est à dire que le programmeur qui ne suit pas ces règles pourra toujours compiler et exécuter son application, mais sans pour autant avoir des garanties sur le comportement de son application au cas où le système serait reconfiguré pendant l'exécution. Ainsi, les propriétés qui vont suivre ne sont pas imposées par le compilateur.

Les objets administrés seront définis *sans contexte*. C'est à dire qu'un objet administré ne contient pas dans son état persistant de pointeurs sur d'autres objets administrés. Cette restriction ne s'impose pas sur les variables locales des méthodes, car ces variables ne font pas partie de l'état persistant. Le contexte qui permet à un objet distribué de communiquer avec les autres objets est défini de façon externe à l'objet sous forme d'une *liste de connexions*.

Chaque objet administré contient un ensemble de listes de connexions associées respectivement aux activités qui exécutent ses méthodes. Une activité a qui appelle une méthode m de l'objet o utilise sa propre liste de connexions pour stocker les références des autres objets qu'elle pourrait appeler en exécutant m sur o. Comme les paramètres d'appel des méthodes sont différents d'une activité à l'autre, l'appel de la même méthode par des activités différentes peut aboutir à des traces d'exécution différentes. C'est pour cette raison que l'on associe une instance de la liste de connexions à chacune des activités appelantes. La figure 5.1 illustre cette structure : a et b sont des activités qui appellent des méthodes de l'objet o_1 et leurs listes de connexions sont respectivement l_1 et l_2. Suite à cet appel de méthode sur o_1, a a appelé des méthodes de o_2 et de o_3 et b a appelé des méthodes de o_2 et de o_4.

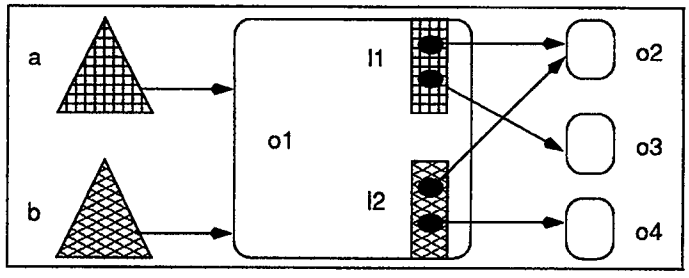

FIG. 5.1 – *Listes de connexions d'un objet administré*

La liste de connexions associée à une activité dans un objet est créée au moment où l'activité obtient son verrou sur l'objet ; elle est détruite une fois que l'activité libère son verrou. Cette stratégie de création et de destruction de listes profite du fait

qu'une activité ne peut pas appeler de méthodes sur un objet si elle n'a pas acquis les verrous nécessaires. Initialement, toutes les entrées dans la liste sont vides. Les méthodes **GetConnection** et **SetConnection** permettent l'accès aux éléments de cette liste. Nous ne pouvons pas modifier le contenu d'une entrée dans cette liste tant que l'activité à laquelle la liste est associée détient un verrou sur l'objet référencé. La figure 5.2 montre un exemple de création et de destruction d'une liste de connexions. Une activité a détient un verrou sur un objet $o1$ et une liste de connexions l_1 lui est associée. Pendant l'exécution d'une méthode sur o_1, a appelle la méthode getLock(2, SHARED), (on demande d'obtenir un verrou partagé sur l'objet référencé par la deuxième entrée de la liste de connexion). L'objet référencé par la deuxième entrée est o_3, dans lequel on crée une instance de la liste de connexions que l'on associe à l'activité a.

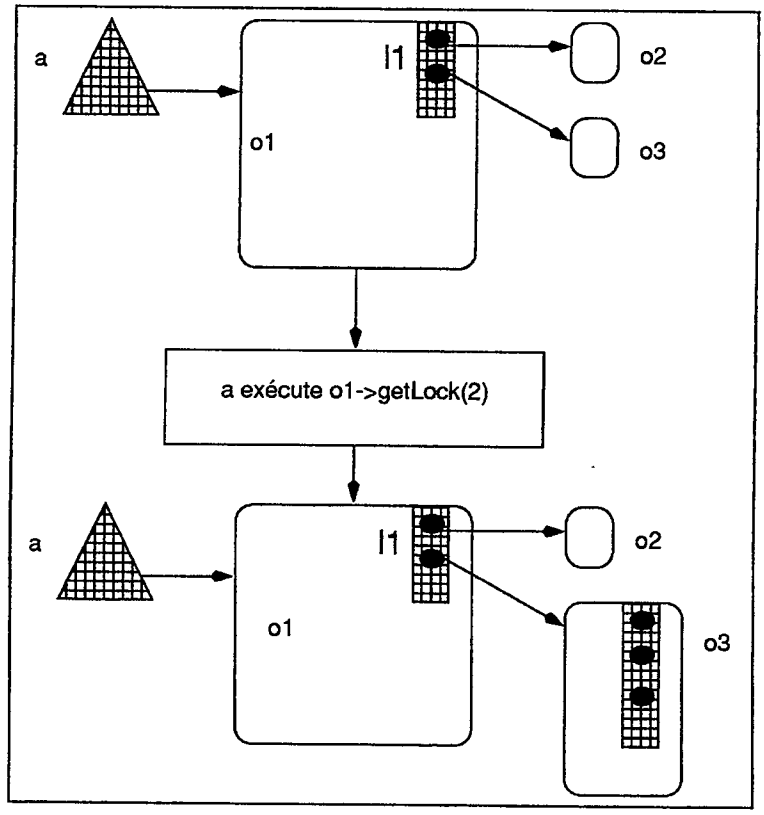

FIG. 5.2 – *Création et destruction de la liste de connexions*

5 Algorithme de changement

Maintenant que le modèle du système et le contexte des objets administrés ont été clairement définis, on peut décrire l'algorithme utilisé pour réaliser les changements.

Le problème le plus prioritaire dans notre cas est celui de la préservation de la cohérence des applications qui s'exécutent sur le système. Pour atteindre ce but, nous cherchons à éviter les incohérences. Nous avons dit à l'introduction de ce chapitre

que la solution idéale serait d'éviter les incohérences au maximum, sachant qu'il est impossible d'aboutir à un succès total.

On note $\mathcal{A}$ l'ensemble des activités du système, $\mathcal{P}$ l'ensemble des objets passifs, et $\mathcal{O}$ l'ensemble de tous les objets. Pour définir plus formellement la cohérence d'une application, nous introduisons les notions suivantes :

- Un objet o_1 *appelle directement* les méthodes d'un objet o_2 pour le compte d'une activité a si et seulement si le pointeur distribué de o_2 appartient à la liste de connexion de o_1 associée à a. On note cette relation comme $o_1\ DI_a\ o_2$. La relation $\mathcal{I}_a$ est définie comme la fermeture transitive de DI_a. Autrement dit, $o_1\ \mathcal{I}_a\ o_2$ signifie que si l'activité a appelle une méthode sur o_1, il se peut qu'il en résulte l'appel d'une méthode sur o_2. Le contraire est aussi vrai, une activité a appelant une méthode sur o_1 ne peut pas appeler une méthode sur o_2 en conséquence de cet appel si la relation n'est pas vérifiée.

- On définit l'ensemble des *objets appelables* depuis un objet o par une activité a comme l'ensemble des objets qui pourront éventuellement être appelés par une activité a si elle appelle une méthode sur l'objet o. La définition formelle est $Inv_a(o) = \{x \in \mathcal{O} | o\mathcal{I}_a x\}$.

- En partant de la définition précédente, on définit l'ensemble des objets appelables par une activité a comme l'ensemble des objets que cette activité pourrait appeler. Si l'on prend en considération qu'une activité a aussi sa propre liste de connexions comme tous les objets administrés, avec la seule différence qu'il n'existe qu'une seule instance pour cette liste, on peut noter cet ensemble comme $Inv(a)$, au lieu de $Inv_a(a)$.

- On définit les *appelants potentiels* d'un objet o comme l'ensemble des activités qui sont susceptibles, grâce aux listes de connexions qui leur sont associées, d'appeler ses méthodes : $PInv(o) = \{x \in \mathcal{A} | o \in Inv(x)\}$.

- On définit la *coupure* d'une activité a à l'instant t, comme l'ensemble des valeurs des objets de $Inv(a)$ à cet instant, et on la note comme $cut_t(a)$. Les coupures sont utilisées comme des points de sauvegarde pour pouvoir effectuer des retours en arrière.

- La *cohérence* d'une activité est définie comme une propriété invariante $\mathcal{F}$ qui s'applique à un ensemble de valeurs d'objets (typiquement une coupure). Une activité est cohérente à l'instant t *ssi* $\mathcal{F}(cut_t(a)) = TRUE$.

La définition que nous avons donnée de la cohérence d'une activité signifie que la cohérence de l'activité ne peut pas être affectée par un objet qu'elle ne connaît pas et qu'elle ne peut appeler ni directement ni indirectement. L'incohérence due au changement de configuration apparaît quand nous avons un objet $o \in Inv(a)$ susceptible d'être appelé par a qui est dans un état déconnecté et que a essaie d'appeler l'une de ses méthodes. L'idée de base de notre algorithme est d'utiliser les primitives de synchronisation (acquisition et libération de verrous) comme des *points de contrôle* permettant de vérifier la disponibilité d'un objet.

Essayons maintenant de discuter les problèmes d'incohérence qui peuvent avoir lieu. Soit $o \in \mathcal{P}$, on peut diviser l'ensemble des activités appelantes de o en deux classes en fonction de leur relation avec cet objet : celles qui détiennent des verrous sur o dénommées $Lk(o)$, et celles qui n'en ont pas. Il est clair que les activités qui sont les plus sensibles aux changements que peut subir o sont celles qui détiennent des verrous sur cet objet. En effet, ces activités on déjà franchi le point de contrôle et supposent donc que l'objet est disponible ; elles risquent donc de s'exposer aux conséquences d'une déconnexion ou d'un retrait de l'objet. Pour éviter ce problème, on interdit tout changement sur la configuration d'un objet o tant que l'ensemble $Lk(o)$ n'est pas vide. Une fois que cette condition est remplie, nous devons régler le problème des demandes de verrous. Ces demandes seront refusées car l'objet n'est plus dans l'état connecté. Nous devons donc nous assurer que ce refus n'aura pas de conséquence sur la cohérence de l'activité appelante.

5.1 Les états de configuration

Pour mieux décrire l'évolution de l'état de configuration d'un objet administré, nous avons besoin d'introduire un ensemble d'états transitoires qui sont nécessaires pour passer de l'état connecté à l'état déconnecté et inversement. On définit alors les états suivants :

- Un objet est dit dans un état de *purge* s'il refuse d'allouer de nouveaux verrous aux activités qui les demandent, mais il continue à accepter les appels sur ses méthodes en provenance des activités ayant déjà acquis les verrous nécessaires.

- Un objet o est dans un état *potentiellement déconnecté ssi* il ne reçoit pas d'appels sur ses méthodes, à l'exception des demandes de verrous qui seront refusées. Pour atteindre cet état, il faut que l'ensemble $Lk(o)$ soit vide. Cet état est fonctionnellement équivalent à l'état déconnecté et il sera considéré ainsi dans la suite de ce chapitre.

- Un objet o est dans état *isolé* si aucune activité ne peut appeler ses méthodes. Autrement dit, *ssi* l'ensemble $PInv(o)$ est vide. Pour ce faire, il faut que la référence de o n'apparaisse dans aucune des listes de connexions des objets de la MIB.

La figure 5.3 décrit les états de configuration et les transitions entre eux. Les cercles en pointillé représentent des états transitoires qui ne sont pas visibles de l'extérieur. Les transitions écrites en gras représentent des transitions provoquées par les méthodes appartenant à l'interface de contrôle des objets administrés.

Dans la suite de ce chapitre nous allons décrire le protocole de changement et l'algorithme utilisé pour faire passer le système d'un état de configuration à l'autre. Le protocole satisfait aux contraintes indiquées dans le chapitre précédent. En particulier, on cherche à satisfaire la contrainte d'avoir des spécification déclaratives de haut niveau pour les changements. Les changements sont effectués en utilisant les primitives *Add, Remove, Connect et Disconnect.*

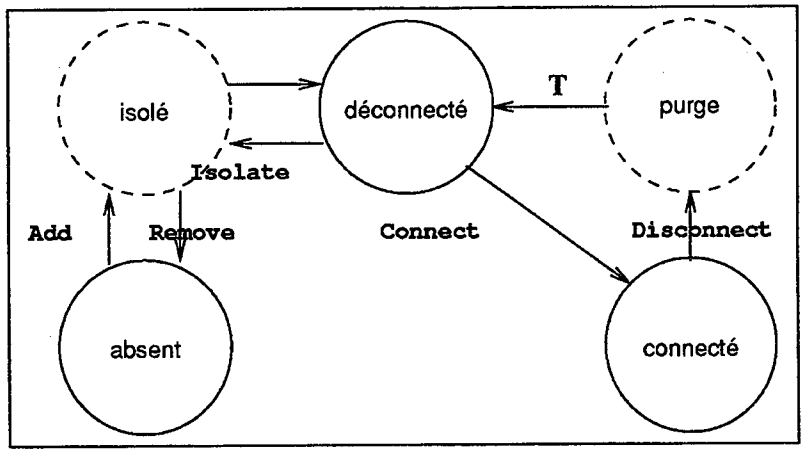

FIG. 5.3 – *Les états de configuration d'un objet administré*

5.2 Règles de changement

Les règles de changement permettent de déduire l'ensemble des actions à réaliser à partir des spécification de changement. Nous allons donc examiner chacune des primitives et présenter les préconditions nécessaires à leur exécution.

5.2.1 Ajout d'un objet

Aucune précondition n'est nécessaire. Au moment de l'ajout d'un objet et son enregistrement dans la MIB, l'objet se trouve dans l'état isolé et il est sans contexte. Aucune communication n'existe donc entre cet objet et les autres objets du système. Il ne peut donc affecter la cohérence du système.

5.2.2 Connexion d'un objet

Aucune précondition n'est nécessaire là non plus. Quand un objet est connecté, il peut y avoir des pointeurs qui référencent cet objet dans les contexte des activités du système. Mais ces pointeurs ne posent pas de problème et il ne peuvent pas être source d'incohérence. La raison est simple : aucune activité ne détient de verrou sur l'objet, il ne peut donc pas figurer dans la coupure des activités et il ne fait pas partie des conditions de cohérence.

5.2.3 Déconnexion d'un objet

Précondition de la déconnexion : aucune activité ne détient un verrou sur l'objet en question. La raison est que le fait de détenir un verrou exprime le besoin de l'activité d'accéder à cet objet. On n'a donc pas le droit d'empêcher l'accès des activités qui détiennent des verrous, car cela pourrait provoquer des incohérences.

5.2.4 Retrait d'un objet

Précondition du retrait : l'objet se trouve dans état isolé. Il n'existe aucun pointeur distribué dans le contexte des activités qui s'exécutent dans le système et qui pointe vers cet objet. Sinon, les activités qui possèdent un pointeur sur un objet retiré risquent d'appeler les méthodes des objets inexistants (pointeur invalide).

5.3 Propriétés de non blocage

Nous nous intéressons au cas où l'on cherche à faire passer un objet de l'état connecté vers l'état déconnecté. Pour ce faire, l'objet doit d'abord passer par l'état de purge pour s'assurer que toutes les activités qui détiennent déjà des verrous pourront utiliser l'objet avant qu'il ne soit déconnecté. Pour que notre algorithme fonctionne correctement, l'état de purge doit être transitoire (sinon on ne peut jamais déconnecter l'objet). Démontrons que c'est bien le cas :

Propriété 1
Dans un système sans interblocage, où le temps d'exécution d'une méthode est fini et où les activités appellent un nombre fini de méthodes, un objet a besoin d'un temps fini pour passer de l'état de purge à l'état déconnecté.

Preuve Soit une activité a qui détient un verrou sur un objet o qui est dans l'état de purge ($a \in Lk(o)$). L'activité a exécute les méthodes $m_1, m_2, ..., m_n$. Toute activité doit libérer les verrous qu'elle détient lors de sa terminaison, ce qui signifie qu'il existe $i \in 1, ..., n$ avec $m_i = o.UnLock$. L'activité a va appeler toutes les méthodes $m_1, m_2, ..., m_{i-1}$ puisque le système est sans blocage. Le temps nécessaire à a pour libérer son verrou et pour quitter l'ensemble $Lk(o)$ est égal à la somme du temps d'exécution de ces méthodes. Étant donné que i est fini et que le temps d'exécution de chacune des méthodes est fini, cette somme est finie.

Soit N le nombre d'activités qui détiennent des verrous sur o au moment où o franchit la transition vers l'état de purge. N ne peut augmenter car o refusera d'allouer de nouveaux verrous. Étant donné que toutes les activités vont finir par libérer leurs verrous au bout d'un temps fini, il est clair que le $Lk(o)$ deviendra vide (ce qui équivaut à dire que o est dans un état déconnecté) au bout d'un temps fini.

5.4 Propriété d'exactitude

Maintenant que nous avons démontré que le temps nécessaire pour passer de l'état connecté vers l'état déconnecté est fini, nous devons voir dans quelle condition ce passage est possible. C'est-à-dire, quelles sont les consignes que les applications doivent suivre pour ne pas avoir des problèmes d'incohérence suite à la déconnexion d'un objet. C'est ce que traduit la propriété suivante :

Propriété 2
Soit a une activité qui détient un verrou sur un objet o qui passe de l'état connecté

à l'état déconnecté ; a préserve sa cohérence si :

- Soit elle ne libère pas son verrou sur o avant la fin de son exécution.
- Soit elle le libère avant sa terminaison, mais n'essaie jamais plus de le redemander.

Preuve

Le premier cas où a garde le verrou jusqu'à sa terminaison est trivial, parce que o continuera à accepter les appels de a (définition de l'état de purge). Dans le deuxième cas, la trace d'exécution de a est composée de séquences d'appels de méthodes $m_1, m_2, ..., m_n$. Si a détient un verrou sur o, et le libère avant sa terminaison, il existe alors un certain $k \in 1, ..., n | m_k = o.UnLock$. La propriété donnée suggère que $\forall i \in k+1, ..., n; m_i \neq o.Lock$, ce qui signifie que à partir du moment où a libère le verrou qu'elle détient sur o, elle n'aura plus besoin d'appeler aucune méthode sur o, et du coup les changements sur la configuration de o n'affectent pas sa cohérence.

Interprétation

Cette propriété définit une classe d'applications qui ne seront pas affectées par les reconfigurations du système. Cette classe est caractérisée par la politique de verrouillage adoptée par l'application. La question qui se pose est de savoir si cette classe d'application est suffisamment large pour rendre l'algorithme intéressant. Les applications qui observent un schéma de verrouillage à deux phases (ex. transactions) vérifient cette propriété. La politique de verrouillage imposée par la propriété est moins restrictive que celle appliquée dans le schéma de verrouillage à deux phases qui force l'application à ne plus demander l'acquisition d'AUCUN verrou une fois qu'elle en a libéré un. Dans notre cas, l'application peut toujours demander l'acquisition de nouveaux verrous, mais ne doit pas chercher à acquérir un verrou qu'elle à déjà libéré. Cette classe d'applications semble assez large ; elle est certainement plus large que celle des applications à interactions indépendantes considérée dans Conic.

5.5 Gérer les refus de verrous

Nous venons de discuter du problème des activités qui détiennent un verrou sur un objet qui passe de l'état actif vers l'état passif. Nous allons maintenant discuter du cas des autres activités qui ne détiennent pas de verrou sur cet objet mais qui cherchent à en acquérir. Ces activités verront leur demande rejetée, et donc risquent de se trouver dans une situation incohérente. On peut alors identifier deux cas qui seront analysés dans la suite :

1. L'activité peut continuer son exécution sans cet objet particulier, ce qui peut être obtenu par plusieurs moyens (remplacement de l'objet par un autre équivalent, l'application a prévu le problème de refus de verrou, etc.). Dans ce cas, l'exécution peut continuer, mais il faut éviter de tomber à nouveau dans ce problème de rejet de verrou qui constitue une source potentielle d'incohérence.
2. L'activité ne peut pas continuer son exécution. On doit donc recommencer l'exécution de l'activité depuis son dernier point de reprise cohérent.

5.5.1 Premier cas, l'activité peut continuer son exécution

Il existe deux cas dans lesquels une activité peut continuer son exécution malgré l'indisponibilité d'un objet qu'elle demande. Le premier consiste à lui fournir un remplacement qu'elle accepte. Le deuxième dépend de la décision de l'activité elle même de continuer son exécution même si un remplacement ne peut être fourni. Dans tous les cas, il faut empêcher l'activité de redemander le verrou sur le même objet ultérieurement pour éviter l'occurrence d'un deuxième rejet.

Quand une activité $a \in \mathcal{A}$ voit sa demande d'acquisition d'un verrou sur un objet o rejetée, l'ensemble des objets accessibles par a ($Inv(a)$) est parcouru pour enlever toute référence à l'objet o. Cet ensemble est construit en effectuant un parcours récursif des listes de connexions de l'activité en question. Les objets administrés héritent de la méthode SubRef(origin,repl) comme partie de leur interface de contrôle. Quand cette méthode est appelée sur un objet o par une activité a, elle garantit qu'après la fin de l'appel, toutes les occurrences de l'objet origin sont remplacés par l'objet repl. La valeur du paramètre repl peut être nil pour exprimer l'indisponibilité d'un objet de remplacement.

L'avantage de cette méthode provient du fait qu'il n'est pas indispensable de retirer TOUTES les références à l'objet déconnecté dans la MIB, parce que l'objet est toujours présent. Mais une fois que l'objet est retiré pour de bon, toute la MIB doit être parcourue pour retirer toutes les références à l'objet. Il faut toutefois noter que la déconnexion d'un objet correspond à une indisponibilité temporaire de la ressource sous-jacente, alors que le retrait d'un objet de la MIB correspond au retrait de cette ressource du système. Si l'on prend l'exemple d'un objet de la classe site, la déconnexion de cet objet correspond à l'arrêt (shutdown) du site, alors que le retrait du site correspond à la déconnexion physique et le retrait du site du réseau. C'est à cause de cette différenciation entre les états «déconnecté» et «isolé» que nous sommes capable de faire l'économie du parcours de la MIB. Ce parcours complet ne doit avoir lieu qu'au cas du retrait de la ressource du système, ce qui est nettement moins fréquent que les arrêts temporaires qui peuvent se produire et rendent la ressource temporairement indisponible.

La nécessité d'enlever les références vers les objets indisponibles dans la MIB nous a obligé à utiliser des objets administrés sans contexte et à introduire le contexte sous forme d'une liste de connexions *externe* à l'objet. Les propriétés d'encapsulation interdisent l'accès à l'état persistant de l'objet, et même si on pouvait contourner ce problème, on serait obligé de pouvoir repérer les variables d'état de l'objet qui représentent des pointeurs distribués pour vérifier systématiquement leur contenu.

La seule contrainte imposée au programmeur avec cette solution est l'obligation d'utiliser la liste de connexions pour pouvoir acquérir les verrous. La méthode de demande de verrou renvoie en résultat un pointeur distribué vers l'objet désiré, ce pointeur peut être alors utilisé sans aucune restriction, tant que l'activité qui a demandé le verrou ne le libère pas. Un tel pointeur est dit *sûr*, puisque on sait que l'objet qu'il référence ne peut passer dans l'état déconnecté tant qu'il est verrouillé.

5.5.2 Cas où l'activité ne peut pas poursuivre son exécution

Dans ce cas, on doit effectuer un traitement d'erreur. L'activité effectue alors un retour en arrière à l'aide d'un point de reprise défini par une coupure cohérente $Cut_{t0}(a)$. Le problème que nous avons concerne les objets qui ont été libérés après l'instant t_0 et qui ont été verrouillés par une autre activité. Pour pouvoir récupérer l'état de a à l'instant t_0, nous seront obligés de forcer un retour vers l'arrière de cette activité à un instant antérieur à t. Le même problème peut se poser à nouveau ce qui pourrait nécessiter un retour de nouvelles activités. Ce problème est connu sous le nom de *l'effet domino* ou les avortements en cascade [Had88]. Un autre problème se pose concernant les objets qui ont été verrouillés après le point de sauvegarde. Pour ces objets, l'activité ne dispose d'aucun moyen de récupérer leur ancien état, ce qui signifie que l'activité ne pourra pas effectuer son retour en arrière proprement. Nous devons éviter les deux cas de figure. D'où la propriété suivante :

Propriété 3

Si une activité a s'assure que sa coupure est cohérente chaque fois qu'elle obtient ou libère un verrou sur un objet, alors l'activité peut restaurer son état à partir de sa dernière coupure sans mettre en cause la cohérence du système.

Preuve

Soit a une activité qui a enregistré sa dernière coupure à l'instant t_0. Soit $m_{t0+1}, ..., m_{t0+n}$ la trace de a à partir de t_0 et jusqu'à l'instant présent. La propriété que nous suggérons garantie que $\forall m_{tk}, t_k \in t_{0+1}, .., t_{0+n}, m_{tk} \neq$ Lock ou Unlock sur un objet. Ce qui implique que $Lk_{t0}(a) = Lk_{t0+n}(a)$. Tout ce dont on a besoin de faire est de restaurer les objets qui appartiennent à $Lk(a)$ à leur état à l'instant t_0.

6 Exemple d'illustration

Pour mieux comprendre le fonctionnement de l'algorithme, nous montrons son application sur un exemple [ORdP96a]. Le même exemple a été donné par les auteurs de Conic et de POLYLITH, il constitue donc l'exemple canonique des algorithmes de gestion dynamique de la configuration. Il s'agit d'une version *évolutive* du problème des philosophes introduit par Dijkstra [Dij72].

Dans le problème classique, les philosophes assis autour d'une table constituent un anneau. Chaque philosophe partage deux fourchettes avec ses deux voisins. Un philosophe peut être en train de réfléchir, de manger ou affamé en attendant de manger. Pour pouvoir manger, le philosophe doit acquérir deux fourchettes, une de son voisin de gauche et une de son voisin de droite. La boucle exécutée par un philosophe est donc la suivante :

```
While (TRUE)
  {
    Refléchir;
    Récupérer la fourchette à droite;
```

```
    Récupérer la fourchette à gauche;
    Manger();
    Relacher les fourchettes();
}
```

Tous les philosophes demandent la fourchette de droite avant celle de gauche, à l'exception d'un seul philosophe qui est gaucher. Cette propriété empêche l'apparition d'interblocages et protège les philosophes de la famine.

Dans la version répartie dynamique, un philosophe est représenté par une activité qui exécute le code de la méthode «run» d'une instance de la classe *Philo* dérivée de la classe module définie dans le chapitre 3. L'ensemble des philosophes peut appartenir à un ou plusieurs domaines, c'est sans importance. Le module *Philo*, comme tous les objets administrés, exporte la méthode **GetLock**(Fork,EXCLUSIVE) qui permet à une activité d'acquérir un verrou sur l'objet fourchette dont elle a besoin. Le premier paramètre représente l'indice de la fourchette dans la liste de connexions (valeurs possibles: RIGHTFORK et LEFTFORK), et le deuxième paramètre représente le type de verrou demandé (en l'occurrence exclusif). Les philosophes sont représentés par des flots d'exécution concurrents et répartis, et les fourchettes sont représentées par des objets administrés qui sont des ressources partagées. Le problème qu'il faut gérer est celui de la redistribution des fourchettes en cas d'un changement de la configuration (ajout, retrait, connexion et déconnexion de fourchettes).

Une activité spéciale contrôle l'exécution des philosophes, il s'agit du *gestionnaire de configuration.* Toutes les actions exécutées par le gestionnaire sont complètement transparentes pour le programme des philosophes. Le gestionnaire peut créer d'autres activités pour pouvoir exécuter plusieurs modifications en même temps.

Nous considérons que chaque philosophe possède une fourchette qu'il partage avec son voisin de droite. Le gestionnaire de la configuration peut demander à un philosophe de s'arrêter d'exécuter sa boucle pour un moment mais sans quitter la table. Un philosophe suspendu ne participe pas à la vie de la communauté et doit donc retirer sa fourchette, ce qui correspond en effet à une déconnexion de la ressource «fourchette». Le gestionnaire peut demander à un philosophe de réintégrer la communauté et donc de remettre sa fourchette à la disposition des autres philosophes, ce qui correspond à une connexion de la ressource «fourchette». Le gestionnaire de configuration peut demander aussi au philosophe de partir définitivement. Quand un philosophe quitte la table, il emmène sa fourchette avec lui, c'est le problème de retrait de ressource. Un nouveau philosophe peut arriver à n'importe quel moment et met sa fourchette à la disposition des autres, c'est le problème d'ajout de ressources. Nous allons voir comment le gestionnaire de configuration gère tous ces problèmes. Dans la suite, la fourchette d'un philosophe x est notée x.fork.

6.1 Déconnexion

Un philosophe auquel on demande de s'arrêter déconnecte sa fourchette. Pour ce faire, il appelle la méthode Disconnect() sur cette fourchette. Quand l'activité qui représente son voisin de droite avec lequel il partageait cette fourchette tente d'acquérir un verrou sur la fourchette la demande est rejetée. Comme nous avons vu

dans la section précédente, il y a deux possibilités :

1. Le gestionnaire peut fournir un remplacement et l'activité a qui demande l'objet l'accepte. Dans ce cas, l'ensemble $Inv(a)$ est parcouru d'une façon récursive pour substituer les références du nouvel objet à celles de l'objet déconnecté.

2. Pas de remplacement, l'activité doit donc effectuer un retour en arrière et restaurer les anciennes valeurs des objets qui constituent le dernier point de sauvegarde.

En ce qui concerne notre exemple, le philosophe x, dont les voisins de gauche et de droite sont respectivement u et v, est prié de suspendre son exécution et de déconnecter sa fourchette. Quand le philosophe v appelle la méthode **GetLock**(LEFTFORK, EXCLUSIVE) pour demander un verrou sur sa fourchette de gauche, la demande est rejetée car la fourchette est déconnectée. Le refus de verrou provoque alors une exception (exception C++). L'exception est traitée par un gestionnaire d'exceptions réalisée sous forme de la méthode **OnLockRejection()**. Cette méthode effectue un traitement par défaut et envoie une notification au gestionnaire de configuration. La notification est envoyée sous forme d'un événement qui contient les informations d'identification de la fourchette et l'activité qui a provoqué cette exception. Le gestionnaire de configuration répond en exécutant le code suivant :

```
v.SetConnection(LEFTFORK,u.GetConnection(RIGHTFORK));
```

Il donne donc à v la fourchette de u pour qu'il l'utilise à la place de celle de x. C'est donc le cas où l'on possède un remplaçant acceptable. Les programmeurs peuvent modifier le traitement de l'exception en surchargeant la méthode OnLockRejection.

On peut déjà faire deux remarques :

- Le coût de déconnexion en lui-même est minimal. Aucune activité n'est suspendue pendant la déconnexion, et les seules activités perturbées par la déconnexion sont celles qui demandent à se servir de la ressource déconnectée. La solution préconisée par Conic nécessitait la suspension de quatre activités (u,v et leurs voisins respectifs) pour obtenir le même résultat.

- L'option consistant à surcharger la méthode OnLockRejection qui permet de modifier le comportement par défaut des applications permet aux programmeurs de réclamer plus de contrôle sur l'exécution de leurs programmes. Le programmeur a donc la liberté totale de fournir son propre gestionnaire de configuration s'il veut que son programme ait un comportement différent de celui préconisé par le gestionnaire par défaut. Ce dernier applique des règles génériques qui ne sont pas optimisées pour chacun des programmes.

6.2 Retrait

Le retrait d'un objet est définitif, à l'inverse de la déconnexion qui elle est temporaire. Pour réaliser le retrait d'un objet o, le gestionnaire de la configuration exécute les opérations suivantes :

1. Déconnecter l'objet o.

2. Pour **toutes** les activités a du système, parcourir récursivement l'ensemble $Inv(a)$ pour enlever toutes les références vers o dans les listes de connexions.

3. Une fois l'étape précédente terminée, l'objet est dans un état de configuration isolé et peut être retiré sans problèmes.

La deuxième étape peut sembler assez coûteuse, mais le retrait total et définitif d'une ressource ne constitue pas une opération fréquente. Il s'agit de la même différence entre l'arrêt d'une machine qui va redémarrer après et la déconnexion physique de la machine pour l'enlever définitivement du réseau. De plus, le parcours ne nécessite l'arrêt d'aucune activité puisque l'objet est déjà dans un état déconnecté et qu'il n'est utilisé par personne. On peut donc exécuter ce genre d'opérations sous forme de ramasse miettes avec une priorité basse. Finalement, seuls les objets qui sont verrouillés seront examinés pendant la phase 2 du parcours car les objets non verrouillés n'ont pas d'instances de listes de connexions associées (les objets administrés sont par définition sans contexte).

Quand le gestionnaire de configuration demande au philosophe x de quitter la table, le philosophe déconnecte sa fourchette, puis la retire du pool de fourchettes puisqu'il s'agit d'un départ définitif. Le gestionnaire de configuration exécute alors les opérations suivantes:

```
x.fork.Disconnect();
x.fork.Remove();
pour (p appartenant à {activités représentant des philosophes}) faire
  si l'une des fourchettes de p $=$ x.fork alors
    changer la référence dans la liste de connexions par NUL.
```

6.3 Ajout

L'ajout est une opération très simple, le gestionnaire de configuration se contente d'appeler la méthode Add de la ressource (fourchette) ajoutée. La raison est que l'état de configuration de la nouvelle ressource vaut *isolé* au départ et que l'objet doit être connecté pour être pris en considération par les activités du système.

6.4 Connexion

Quand un objet est connecté, le gestionnaire de la configuration peut avoir besoin d'effectuer une redistribution des ressources pour permettre aux activités de tenir compte de la disponibilité de la nouvelle ressource.

Dans notre exemple, le gestionnaire décide de connecter le philosophe x au milieu de deux autres philosophes, u sur sa gauche et v sur sa droite. Un changement doit être effectué pour permettre à v de partager la fourchette de x au lieu de celle de v, et x partage la fourchette de u à la place de v. Le gestionnaire de la configuration doit donc exécuter les opérations suivantes:

```
x.fork.Connect();
x.SetConnection(LEFTFORK,u.fork);
// x partage la fourchette de u positionné à sa gauche
```

```
x.SetConnection(RIGHTFORK,x.fork);
// x tient sa fourchette dans sa main droite
v.SetConnection(LEFTFORK,x.fork);
//v partage la fourchette de x positionné à sa gauche
// et non plus celle de u.
```

7 Evaluation

L'algorithme cherche à trouver un compromis entre quatre points:

- réduire le ralentissement du système,
- réduire les mécanismes nécessaires pour effectuer les changements,
- permettre à une classe assez large d'applications de fonctionner sur le système sans être affecté par les changements,
- et finalement offrir aux programmeurs une transparence maximale des changements pour les encourager à utiliser notre système.

7.1 Evaluation des Performances de l'algorithme

A priori, notre solution au problème de la gestion dynamique de la configuration devrait mieux satisfaire aux trois critères de performance que nous avons définis dans le chapitre précédent que les algorithmes concurrents.

- Il ne devrait pas y avoir de ralentissement considérable même dans le cas le plus coûteux, celui de la déconnexion d'un objet. Ceci devrait résulter du fait qu'aucune suspension d'activité n'est nécessaire pour effectuer le changement.
- Un changement devrait être réalisé plus rapidement que dans les autres algorithmes car dans l'algorithme que nous proposons, le gestionnaire de configuration n'attend pas la suspension des activités avant d'exécuter la modification demandée.
- Le mécanisme de verrouillage n'introduit pas de coût supplémentaire relativement au mécanisme d'accès aux objets utilisé à l'origine pour protéger les accès concurrents. La modification apportée au mécanisme de verrouillage consiste à ajouter le code permettant de gérer la création, la destruction, et la modification des listes de connexions, dont le coût est pratiquement négligeable.

Pour vérifier cette analyse a priori. Nous avons utilisé le prototype développé au dessus de la plateforme OODE pour écrire un programme qui met en œuvre l'exemple canonique des philosophes montré dans la section précédente. Nous avons utilisé ce prototype pour étudier l'évolution du temps d'exécution d'une famille de philosophes en fonction de la taille de la famille (nombre de philosophes) et en fonction du nombre de changements qui ont lieu pendant l'exécution.

L'exemple a été testé sur une cellule de deux stations de travail de type Bull DPX-20 équipées d'un processeur RS/6000 et qui fonctionnent sous le système AIX version 3.2. Chacune des stations est équipée de 64 Megaoctets de RAM et une des deux stations exporte une partition partagée par NFS qui a été utilisée pour stocker les objets persistants de OODE. Les résultats donnés dans la suite ont été obtenus comme une moyenne d'un nombre suffisant de mesures (chaque série de mesures a été effectuée au moins trois fois) avec une précision de 5% pour le résultat final.

Nous avons commencé par calibrer notre système de mesure pour obtenir les bons paramètres d'exécution. Les paramètres que l'on peut faire varier sont le nombre de philosophes et le temps d'utilisation des fourchettes pour manger qui est déterminé d'une façon aléatoire pour augmenter la concurrence entre les activités. Nous avons choisi d'utiliser un temps de repos du même ordre de grandeur que le temps de réservation pour simplifier le problème. Pour éliminer les effets dus aux changements, nous avons mesuré le temps d'exécution de la la version statique du programme. Nous avons alors lancé deux séries de mesures du temps d'exécution du programme: la première pour une famille de philosophes ayant un temps d'utilisation aléatoire entre 100-1000 ms, et la deuxième pour une famille ayant un temps d'utilisation aléatoire entre 10-100 ms. Le nombre d'itérations effectuées par chaque philosophe a été fixé à 200. Les résultats obtenus nous permettent de faire deux constatations:

- Le temps d'exécution dans le cas d'un temps de réservation assez grand (100ms-1000ms) ne varie pas énormément comme on peut le constater dans la figure 5.4. C'est dû au fait que les activités deviennent très vite pratiquement synchrones l'une avec l'autre, avec la moitié qui possède les deux fourchettes et l'autre moitié qui se repose. Ce comportement très régulier qui se traduit par un temps d'exécution qui ne varie pas en augmentant le nombre d'activités ne nous convient pas. C'est avec un temps d'utilisation nettement plus petit (10-100 ms) que nous avons trouvé un comportement linéaire comme le montre la figure 5.5. Le système ne permet pas de suspendre les activités pour une durée inférieure à 10 ms, nous donc avons décidé d'utiliser cette valeur pour effectuer nos mesures.

- Après avoir opté pour un temps de réservation entre 10 et 100 ms, nous avons obtenu une courbe linéaire du temps d'exécution en fonction du nombre de philosophes. Cette courbe monte soudain après 50 philosophes, c'est donc la limite du nombre d'activités concurrentes que le système peut supporter sans avoir de répercussions sur les performances. Cet effet sur les performances du système est du au fait que les sémaphores utilisés pour réaliser l'exclusion mutuelle utilisent l'attente active (c'est un problème de réalisation de OODE). L'activité teste si le sémaphore est passant et s'il ne l'est pas elle exécute un appel à l'ordonnanceur (scheduler) du système abandonnant le processeur, mais il n'y a pas de blocage et l'activité continue à être active et on assiste à un grand nombre de commutations de contexte. C'est pour cela que le système semble atteindre la limite de saturation avec un nombre de philosophes supérieur à 50. L'intervalle utile que nous avons choisi est donc entre 5 et 50 philosophes. Les temps d'exécution sont mesurés en secondes.

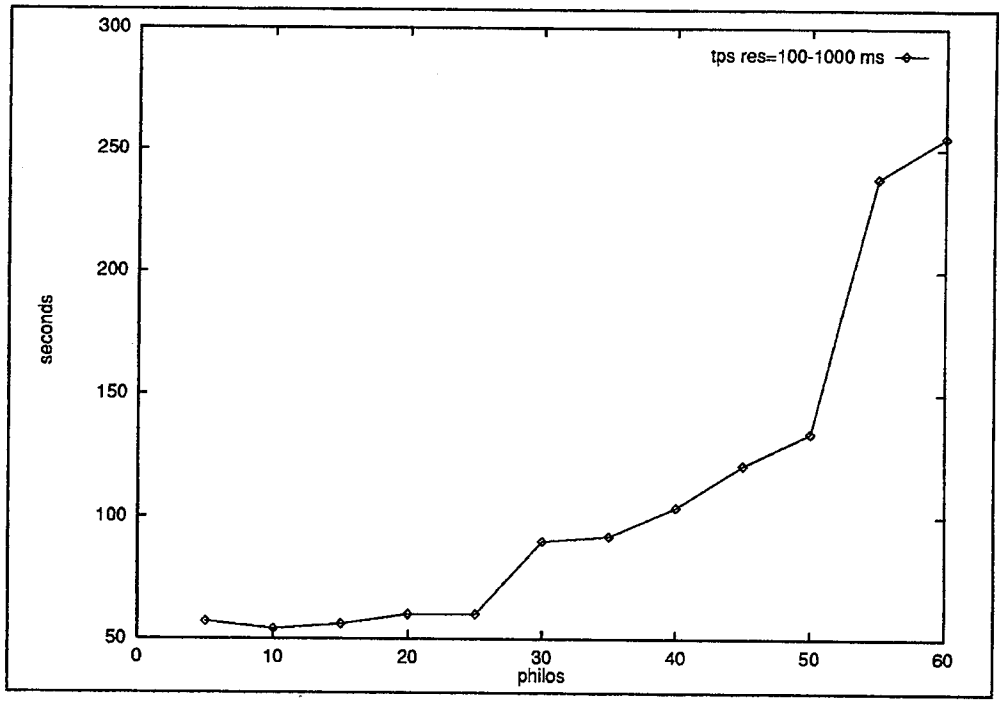

FIG. 5.4 – *Temps d'exécution de la version statique avec tps de réservation=100-1000ms*

Nous avons ensuite mesuré le temps d'exécution de la version dynamique du programme. Dans cette version, le gestionnaire de la configuration envoie un nombre k de changement à chacun des philosophes pendant l'exécution du programme. Nous avons constaté un temps d'exécution stable pour k allant de un jusqu'à cinq. La figure 5.6 montre comment le temps d'exécution de la version dynamique reste très proche de la version statique. Elle montre aussi le temps d'exécution dans le cas où k est égal à 10, on voit que le temps d'exécution n'augmente pas considérablement.

La dernière série de mesures a été effectuée en mettant le système dans des conditions extrêmes, c'est à dire que le gestionnaire de configuration exécute des changements tant que le programme n'a pas terminé son exécution. Le résultat est montré dans la figure 5.7. On voit clairement que le comportement reste linéaire, avec une pente légèrement plus importante. Ce qui permet de conclure que même dans le cas d'un système dont la configuration ne cesse pas de changer, les performances restent acceptable. L'algorithme utilisé par Conic dont les résultats sont donnés dans la figure 5.8 montre un temps d'exécution nettement plus important, et surtout un comportement exponentiel de l'algorithme. Les performances de notre algorithme sont donc nettement meilleures que celles de Conic.

Nous n'avons pas pu mesurer le temps d'exécution d'un changement car ce temps semble être tellement petit que l'outil de mesure utilisé n'est pas assez précis pour effectuer de telles mesures.

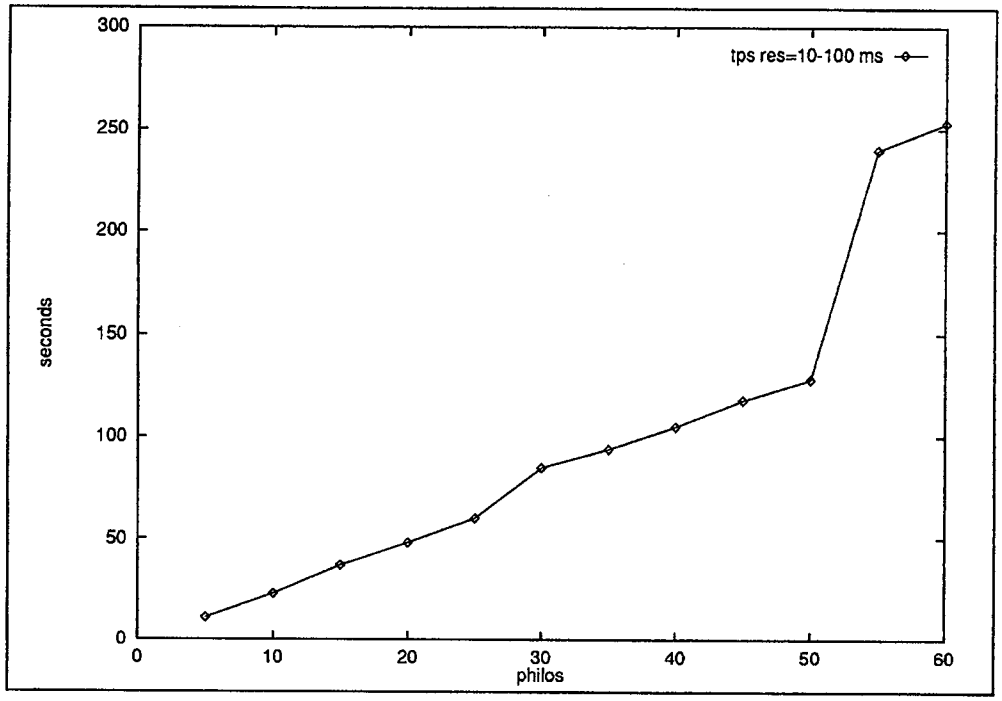

FIG. 5.5 – *Temps d'exécution de la version statique avec tps de réservation=10-100ms*

7.2 Evaluation des mécanismes nécessaires

L'algorithme proposé n'a pas besoin de mécanisme spécial à l'exception du mécanisme de retour en arrière. Ce mécanisme n'est pas fourni par la plateforme OODE, ce qui nous a conduit à le simuler en effectuant manuellement les sauvegardes et les restaurations des copies d'objets lors de l'acquisition et de la libération des verrous.

7.3 Evaluation de la classe d'applications acceptées

Nous avons montré une classe d'applications définie par une politique d'allocation et de libération de verrous. Cette classe contient clairement la classe des applications à interactions indépendantes définie par Conic. Une autre classe est celle des applications qui respectent un schéma de verrouillage à deux phases. Il est difficile d'estimer l'importance de la classe définie dans l'absence d'une véritable classification des applications réparties selon leur politique de verrouillage.

7.4 Evaluation du niveau de transparence fournie

La transparence est importante puisqu'elle permet aux programmeurs d'écrire les programmes sans se poser de questions concernant l'évolution du système pendant leur exécution. La seule vraie contrainte que nous imposons est d'utiliser la liste de connexions pour obtenir la référence de l'objet distribué. La liste de connexions sera utilisée une seule fois, lors de l'acquisition du verrou sur l'objet. Une fois le verrou acquis, l'entrée dans la liste est protégée et ne peut plus être modifiée avant

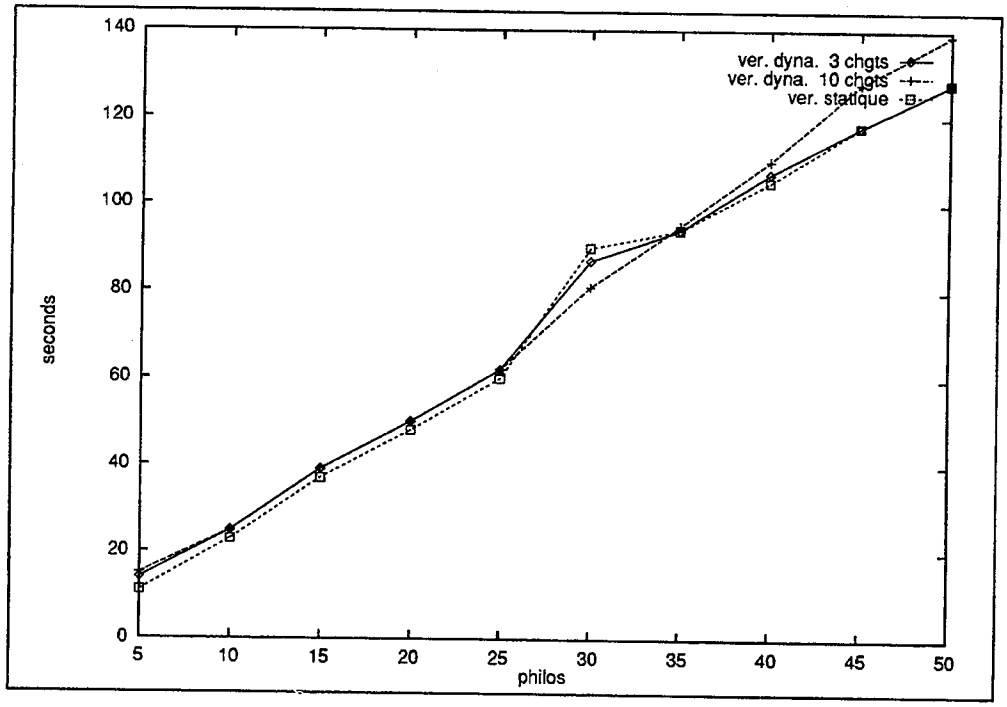

FIG. 5.6 – *Comparaison des tps d'exécution des versions statique et dynamique*

la libération du verrou. La référence peut alors être utilisée sans indirection sous forme de variable locale. L'exemple suivant montre cette possibilité :

```
void MyClass::MyMethod()
{
ManagedObject *aux;
aux = GetLock(5,SHARED);
aux->method1;
aux->method2;
aux->unLock();
}
```

8 Conclusion

Dans ce chapitre, nous avons présenté nos propositions pour un algorithme de gestion des changements. L'algorithme permet de réaliser des changements structurels qui permettent la construction de la plupart des autres catégories de changement. Le grain de changement choisi est celui de l'objet. L'idée de base de l'algorithme est de chercher à éviter les incohérences autant que possible, et dans les cas où les incohérences sont inévitable, d'effectuer un retour en arrière pour les corriger. L'algorithme cherche à établir un compromis difficile entre la réduction du ralentissement du système, la réduction du nombre de mécanismes nécessaires, la transparence vis-à-vis des programmeurs, et le support pour une classe d'applications assez large.

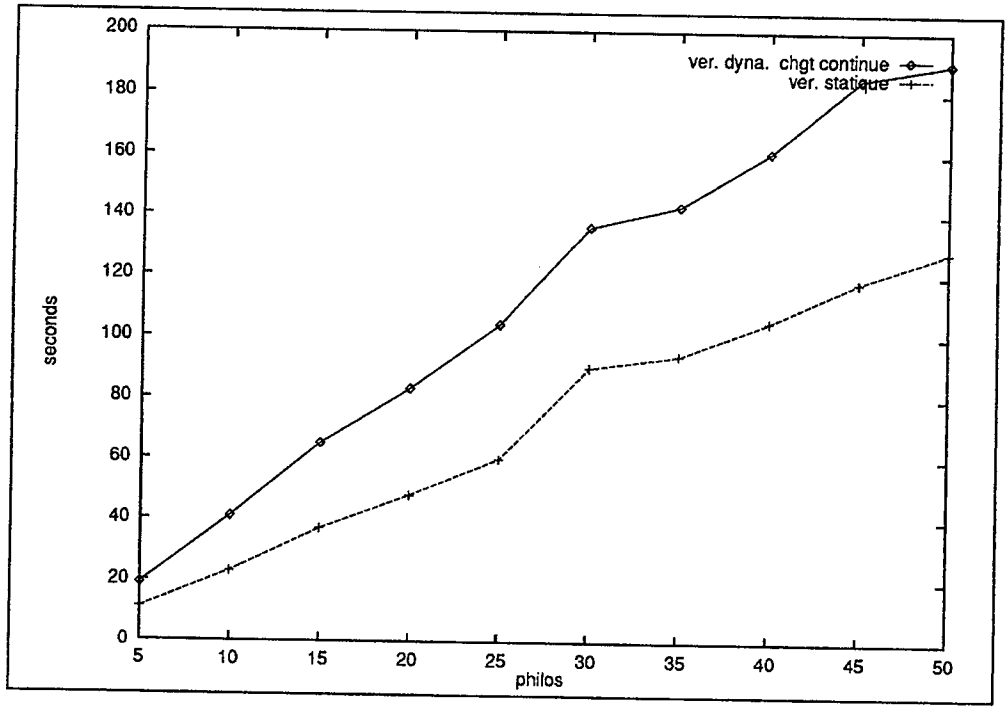

FIG. 5.7 – *Comparaison du temps d'exécution entre la version statique et la version dynamique*

Les mesures que nous avons montrées montrent la validité de notre approche, et une comparaison avec les autres systèmes de gestion de changement semble favorable dans tous les points que nous avons cités.

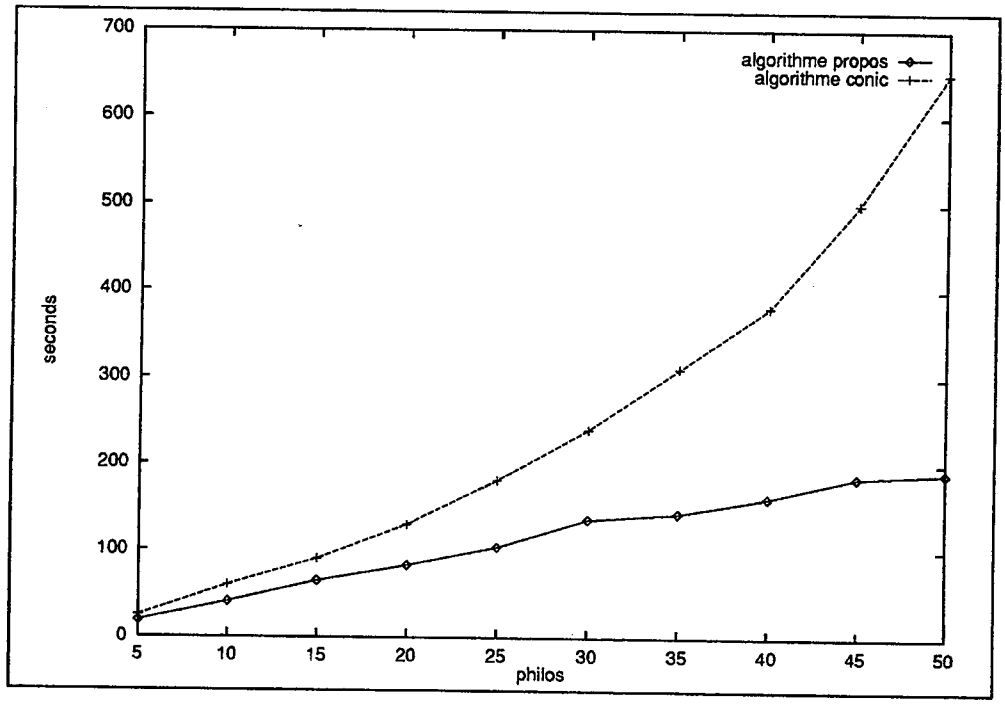

FIG. 5.8 – *Comparaison du temps d'exécution entre la version statique et la version dynamique de Conic*

6

Conclusion et perspectives

A l'origine de ce travail, nous avons répertorié l'ensemble des fonctions d'administration dans un système réparti, telles qu'elles sont définies dans les normes actuelles. Nous avons cherché à définir ce que pourrait être un noyau minimal d'administration réalisant la définition et la gestion de la configuration. Nous avons enrichi ce noyau d'un mécanisme de notification d'événements permettant ainsi la réalisation des fonctions de l'audit. Notre contribution se décompose en deux parties. D'une part, nous définissons un moyen de spécifier les ressources dont dispose le système. D'autre part nous définissons et mettons en œuvre des mécanismes permettant de faire évoluer dynamiquement la configuration.

Nous rappelons d'abord les éléments essentiels constituant notre contribution dans le domaine de l'administration des systèmes répartis, puis nous dégageons quelques perspectives d'évolution qui devraient faire l'objet des études futures.

1 Conclusions sur la spécification de la configuration

Les objectifs qu'un langage de spécification de la configuration doit remplir peuvent se résumer en plusieurs points. Ainsi que nous l'avons vu au chapitre II un langage de spécification de la configuration d'un système réparti doit permettre de donner une représentation abstraite des ressources constituant le système. La représentation fournie doit permettre de décrire les entités administrées dans leur diversité et leur grand nombre, ainsi que les relations liant ces entités. Le langage doit permettre de spécifier complètement l'interface d'accès aux entités. Le langage doit être simple pour permettre une compréhension facile par des personnes qui ne sont pas forcément expérimentées dans le domaine des langages. Il doit aussi être extensible pour permettre l'introduction de nouveaux types de ressources au sein du système. Un mécanisme permettant la désignation des instances d'entités doit être intégré au langage. Ce mécanisme doit être souple pour répondre aux problèmes de migration et doit permettre d'attacher une connotation sémantique facilement compréhensible au nom de l'objet.

Nous avons vu comment différents systèmes d'administration existants (MOIRA, SNMP, OSI) tentent d'atteindre ces objectifs et quelles sont les faiblesses de chacun

d'eux. Nous avons vu aussi les solutions proposés par les MIL aux problèmes de spécification des systèmes hétérogènes à grand nombre de ressources. Nos propositions faites dans le chapitre III sont basées sur les points suivants :

- L'utilisation d'un langage orienté objet pour représenter les entités du système par des instances de classes d'objets. Ainsi, nous avons conçu et réalisé un ensemble de classes permettant de représenter les ressources que nous considérons pour le moment. L'ajout de nouveaux types d'entités se fait par la définition d'une nouvelle classe. Le mécanisme d'héritage fourni par les langages orientés objets permet un développement rapide des nouvelles classes en réutilisant le code qui a déjà été écrit. L'ensemble des classes à été mis en œuvre grâce à la plateforme d'objets distribués OODE.

- Le langage utilisé, appelé OC++, est dérivée de C++, qui est reconnu comme le langage orienté objet le plus utilisé. Ce langage enrichit C++ par l'introduction des notions de distribution et de persistance des objets. Le système peut alors être vu comme un ensemble d'objets distribués qui communiquent par appels de méthodes. L'ensemble des objets persistants décrivant la configuration du système s'appelle la base de données administratives (MIB).

- Le langage OC++ est conforme aux spécifications de CORBA qui constitue le standard actuel des systèmes à objets, ce qui permet l'interoperabilité de notre MIB avec toute application conforme aux spécifications de CORBA, même si cette application n'a pas été écrite avec OC++.

- La conception et la réalisation d'un préprocesseur et d'un ensemble de classes abstraites qui permettent d'introduire des extensions supplémentaires au langage. Cette technique nous a permis de fournir un mécanisme de spécification de relations qui ne fait pas partie des fonctions fournies par OC++.

- La représentation donnée ne se contente pas de couvrir les aspects administratifs des entités. Elle couvre aussi les aspects fonctionnels en permettant de spécifier toutes les fonctions fournies par l'entité représentée. Ainsi, la représentation fournie par le service de gestion de la configuration pour une ressource quelconque sera la seule représentation utilisée par les applications pour communiquer avec cette ressource. Cette unicité de la représentation permet de poser les questions liées à l'administration d'une ressource en même temps que celles liées à son fonctionnement, ce qui permet alors une meilleure intégration de l'administration au sein du système.

- L'intégration d'un service de désignation permettant d'utiliser un système de désignation à deux niveaux pour les instances d'objets. Une désignation interne est utilisée par les applications pour faire référence aux objets administrés ; elle a l'intérêt d'être unique et de ne pas dépendre de la localisation de l'objet. L'autre désignation est symbolique et basée sur une construction hiérarchique. Le service de désignation permet d'obtenir la référence à un objet à partir de son nom symbolique.

- La représentation fournie pour une ressource permet de spécifier des notifications d'événements qui seront engendrées lors des changements de l'état de cette ressource. Ces événements permettent de réaliser les fonctions d'audit qui seront à leur tour utilisées pour surveiller le comportement des ressources et pour prendre les décisions nécessaires.

Nous pensons avoir atteint l'objectif consistant à définir un noyau minimal d'administration. De plus l'ensemble des mécanismes fournis constituent un noyau d'administration cohérent . Cependant, nous sommes convaincu qu'il existe un certain nombre de fonctions intéressantes à ajouter et dont nous discuterons dans la section consacrée aux perspectives.

2 Conclusion sur la gestion dynamique de la configuration

A notre connaissance, la gestion dynamique de la configuration n'est pas intégrée comme une des fonctions proposées par les systèmes d'administration existants. Les solutions proposées à ce problème l'ont été par la communauté de génie logiciel, qui n'a pas forcément les mêmes contraintes que les nôtres. Le chapitre IV nous a permis de dégager les objectifs que le service de gestion des changements doit remplir et que nous rappelons.

Le système doit posséder un degré acceptable de souplesse en permettant les types de changement qui correspondent aux évolutions auxquelles le système pourrait faire face pendant sa durée de vie qui peut s'étaler sur plusieurs années. Le grain est un paramètre important d'un algorithme de changement dynamique. Trop petit, les opérations élémentaires risquent d'être trop nombreuses, trop grand, on peut être amené à modifier des objets non concernés par les modifications. La cohérence des applications qui s'exécutent sur le système ne doit pas être affectée par les changements qui peuvent avoir lieu pendant leur exécution. L'algorithme de gestion de la configuration doit limiter la dégradation des performances, qu'il s'agisse du ralentissement provoqué par l'exécution d'un changement, ou de celui résultant des mécanismes ajoutés au système pour permettre la gestion dynamique de sa configuration, ou finalement du temps que le système met à effectuer un changement, s'agissant d'un état transitoire et potentiellement instable. La solution proposée doit minimiser l'intervention humaine qui peut être source d'erreurs et de ralentissements. Les changements doivent être transparents pour les applications. Les programmeurs d'applications ne doivent pas avoir à incorporer de tests à leur code pour prévoir d'une façon explicite l'évolution du système en vérifiant la disponibilité des ressources. L'algorithme proposé doit se contenter d'un ensemble de mécanismes de base simples et peu coûteux en terme de réalisation et de performance. Les spécification de changement doivent être déclaratives et exprimées dans un langage de haut niveau. Finalement, les changements doivent être validés a priori pour s'assurer qu'ils ne violent pas les règles de cohérence du système.

L'analyse faite montre qu'aucune des solutions existantes n'est vraiment utilisable dans un système distribué. En particulier, ces solutions présentent un coût

considérable en performances ou en nombre de mécanismes de base nécessaires. D'autre part, seule une classe très restreinte des applications peuvent fonctionner correctement sur les systèmes qui utilisent ces solutions. Nous avons donc proposé dans le chapitre V une solution basée sur les idées suivantes :

- Nous utilisons des objets sans contexte. Les références aux objets appelables depuis un objet sont stockées dans une liste de connexions. La liste est crée dynamiquement et rattachée à une activité au moment où cette activité verrouille l'objet. Le gestionnaire de la configuration peut modifier la liste d'un objet pour retirer les références vers des objets non disponibles de façon transparente à l'objet en question.

- L'acquisition des verrous est utilisée comme un point de contrôle permettant de savoir si un objet est disponible ou pas. Un objet ne peut pas être déconnecté tant qu'il existe au moins une activité qui détient un verrou sur cet objet. Nous démontrons que le temps nécessaire pour libérer tous les verrous est fini. Nous montrons aussi que si une activité ne demande pas à obtenir des verrous sur un objet qu'elle a déjà libéré, alors sa cohérence ne sera pas affectée par la déconnexion de l'objet.

- Les activités qui demandent à acquérir un verrou sur un objet déconnecté verront leur demande refusée et un traitement d'exception permettant de chercher un remplaçant éventuel est déclenché. Au cas où aucun remplacement éventuel n'est obtenu, l'activité exécute un retour arrière vers un point de reprise établi. Il appartient à l'activité d'organiser ses points de reprise comme elle le souhaite.

- La comparaison de notre algorithme avec celui de Conic, qui constitue le système de référence en matière de gestion dynamique, est assez favorable. Le ralentissement apporté par notre algorithme est nettement moins important, ceci résulte du fait qu'aucune suspension n'est nécessaire pour exécuter les changements. Notre algorithme est compatible avec une classe d'applications plus large que celle prévue par Conic. Les contraintes pour le programmeur sont réduites à l'utilisation d'une indirection au moment de l'acquisition d'un verrou. Le seul point discutable de notre algorithme est celui du mécanisme de reprise nécessaire.

Nous estimons que la solution proposée constitue un bon compromis entre cinq critères antagonistes : préserver la cohérence du système, limiter la dégradation des performances, fournir un support pour une classe assez large d'applications, garder un bon niveau de transparence vis-à-vis des programmeurs, et finalement réduire le nombre de mécanismes nécessaires. Les mesures que nous avons effectuées permettent de confirmer les bonnes performances de notre algorithme.

3 Développements futurs

Les choix de base de notre système ne sont pas mis en cause. L'utilisation d'un modèle à base d'objets conforme aux spécifications de CORBA pour représenter

les entités administrées, les extensions introduites au langage pour permettre la représentation des relations entre les différentes entités, et la gestion dynamique de la configuration sont autant de points positifs que nous voulons garder. Mais il est certain que notre contribution est loin d'être complète, car le domaine de l'administration des systèmes répartis est très vaste. Il existe un bon nombre de propositions complémentaires qu'il serait bon de prendre en considération pour tout développement futur de notre système. Le but de cette section est de développer et de discuter des fonctions que nous voudrions intégrer à la version suivante.

3.1 Développements pour la spécification de la configuration

La plateforme OODE a été récemment abandonnée et le projet n'a pas donné suite, au moins sous sa forme actuelle. En particulier, le compilateur du langage OC++ qui a été choisi pour décrire la représentation des objets administrés n'est plus maintenu. Un bon nombre de fonctions du langage C++ n'ont pas été réalisées et ne le seront jamais. Nous nous trouvons donc obligés d'utiliser un autre langage pour la spécification de la configuration. Il est préférable que ce langage soit assez proche de OC++ pour faciliter le portage de notre code.

Le langage orienté objet qui semble porteur d'avenir est Java. Nous allons donc considérer la possibilité de porter notre noyau d'administration sur Java.

3.2 Développements pour la gestion des changements

Nous aouhaitons aborder dans l'avenir deux points concernant le problème de gestion des changements :

Les catégories de changement : nous avons montré que notre système permet de réaliser des changements *structurels*, nous avons aussi montré que d'autres types de changements peuvent être obtenus à partir de ces changements, et en particulier les changements de réalisation des méthodes. Cependant, le système n'est pas tout à fait adapté à ces derniers changements. En effet, il est impossible de changer la réalisation d'une classe tant que des instances de cette classe sont chargées en mémoire d'exécution. Un autre problème qui doit être traité est celui de la conversion des instances de l'ancienne version.

La classe des applications acceptées : il s'agit de rendre le système plus souple pour permettre l'utilisation des fonctions de reconfiguration dynamique par une classe d'applications plus large que la classe actuelle. Ce problème est d'autant plus délicat que l'on ne peut trouver nul part un classement des applications distribuées selon leur comportement. A notre connaissance, aucune étude sur le comportement dynamique des applications n'a été effectuée, ce qui rend difficile, et surtout subjective, l'évaluation de la classe actuelle et de son réalisme.

3.3 Interoperabilité avec CMIP

Nous avons défini notre propre modèle pour la représentation des entités administrées. Le modèle à objets que nous utilisons est conforme aux spécifications de

CORBA. Mais nous devons quand même prendre en considération le fait que la plupart des systèmes d'administration existants sont construits autour de SNMP ou de CMIP. Nous nous intéressons en particulier à CMIP car c'est un protocole basé sur une représentation orientée objet. Malheureusement, les objets administrés de CMIP et de OC++ ne peuvent pas communiquer entre eux pour le moment. Il nous faut donc fournir un mécanisme permettant le passage dans les deux sens : il faut d'abord permettre aux applications écrites en OC++ d'accéder aux objets administrés de l'OSI ; il faut aussi rendre les objets OC++ visibles par les applications administratives de l'OSI. Nous nous intéressons en particulier au problème d'accès aux objets administrés OSI depuis noter système.

La solution la plus simple consiste à fournir les primitives d'accès à CMIS, qui est le moyen standard d'accéder à ces objets. Cette solution a plusieurs inconvénients :

- La solution nécessite l'apprentissage du modèle d'objets OSI, ainsi que celui de GDMO, la notation utilisée pour spécifier les objets administrés de CMIP.
- Les programmeurs doivent apprendre l'utilisation de CMIS avec ses options multiples.

Une solution plus intéressante est proposée dans [GG95] qui consiste à représenter les objets administrés de CMIP par des objets du système propriétaire (le système décrit dans la référence est ANSAware) appelés objets adaptateurs *proxies*. Les actions et les attributs d'un objet administré OSI deviennent des méthodes de l'objet adaptateur. Cette solution permet de masquer partiellement le rôle de l'agent CMIP qui sera contacté par les objets adaptateurs et non pas par les applications du système. Il est malheureusement impossible de masquer totalement le rôle de l'agent puisqu'il existe des opérations du protocole CMIP comme la création des objets et la souscription aux notifications qui sont spécifiques à l'agent. Ces opérations seront fournies par un objet proxy qui représente un agent CMIP accessible depuis notre système.

Références Bibliographiques

[Abd91] R. ABDERRAHMAN. «TOBIAS, un modèle orienté objet pour l'administration des systèmes répartis». Rapport Technique Masi 91-46, Université de Paris 6, septembre 1991.

[All94] W. ALLEN. «Experiences gained from the cmipWorks Project». *CMIP Run !*, 3(2), 1994.

[Ba90] K. BIRMAN et AL. «Fast Causal Multicast». Rapport Technique TR90-1105, Cornell University, avril 1990.

[Ba94] B. BEECHER et AL. «Future Computing Environment Monitoring Team, Final Report». Rapport Technique, University of Michigan, juillet 1994.

[BLM96] F. BOYER, E. LENORMAND et V. MARANGOZOV. «Un modèle d'événements pour le support de la coordination dans un système à objets répartis». Rapport Technique 10-96, IMAG-INRIA, Projet Sirac, février 1996.

[Ca94] J. CAYUELA et AL. «OODE : Une plateforme objet pour les applications coopératives». Dans *AFCET*, 1994.

[CRO75] CROCUS. *Système d'exploitation des ordinateurs.* Dunod informatique, 1975.

[Da92] G. DEAN et AL. «Cooperation and Configuration within Distributed Systems Managment». Dans *International Workshop on Configurable Distributed Systems*, Imperial College, London, 1992.

[Dea93] G. DEAN. «Distributed Systems Management as a Group Activity». Dans *IEEE first International Workshop on Systems Management*, avril 1993.

[Dij72] E.W. DIJKSTRA. *Hierarchical Ordering of Sequential Processes.* Academic Press, 1972.

[ELM+92] F. EXERTIER, H. LEJEUNE, B. MICHEL, M. RIVEILL, et M. SANTANA. «An extended C++ for Oode». Unité Mixte Bull-IMAG, juin 1992.

[Fab76] R. FABRY. «How to design a system in which modules can be changed on the fly». Dans *2nd International Conference on Software Engineering*, pages 470–476. IEEE-CS Press, 1976.

[FE89] L. FELDKUHN et J. ERICKSON. «Event Management as a Common Functional Area of Open Systems Management». Dans *IFIP Symposium on Integrated Network Management*, 1989.

[Fou92] Open Software FOUNDATION. «OSF Distributed Management Envireonment Architecture». White Paper, mai 1992.

[Gen96] G. GENILLOUD. «An Analysis of the OSI Systems Management Architecture from an ODP Perspective». Dans *Second IEEE Workshop on System Management*. IEEE Computer Society Press, juin 1996.

[GG95] G. GENILLOUD et D. GAY. «Accessing OSI Managed Objects from ANSAware». Dans *IFIP-IEEE Workshop on Distributed Systems, Operation and Management*, 1995.

[Had88] V. HADZILACO. «A Theory of Reliability in Database System». *Journal of the ACM*, 35(1) :121–14, janvier 1988.

[HH96] H. HIGAKI et Y. HIRAKAWA. «Group Communication for Upgrading Distributed Programs». Dans *16th International Conference on Distributed Computing Systems*, 1996.

[Hig94] H. HIGAKI. «Group Communications Algorithm for Dynamically Updating in Distributed Systems». Dans *1994 International Conference on Parallel and Distributed Systems*, pages 418–424. IEEE Computer Society Press, décembre 1994.

[HP93] C. R. HOFMEISTER et J. M. PURTILO. «A FRAMEWORK FOR DYNAMIC RECONFIGURATION OF DISTRIBUTED PROGRAMS». Rapport Technique 3119, Computer Sciences Department, University of Maryland, août 1993.

[HWP93] C. HOFMEISTER, E. WHITE et J. PURTILO. «Surgeon : a packager for dynamically reconfigurable distributed applications». *Software Engineering Journal*, pages 95–101, mars 1993.

[ISO92] ISO/IEC. « *Open Systems Interconnection - Structure of Management Information - Part 4 : Guidelines for the Definition of Managed Objects*», 1992. ITU-T Recommendation X.721.

[KA95] A. KERBRAT et S. BEN ATALLAH. «Formal Specification for a Framework for Groupware Development». Dans *8th International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE'95)*, pages 303–310, octobre 1995.

[Kal91] B. S. Jr. KALISKI. « *A Layman's Guide to a Subset of ASN.1, BER, and DER*». RSA Data Security, Inc., Redwood City, CA, juin 1991.

[Ker89] A. KERMARREC. «*La Transparence dans les Systèmes Distributés : l'approche de GOTHIC*». PhD thesis, Université de Rennes, septembre 1989.

[Kle88] S.M. KLERER. «The OSI Management Architecture: an Overview». *IEEE Network*, 2(2) :20–29, mars 1988.

[KM85] J. KRAMER et J. MAGEE. «Dynamic Configuration for Distributed Systems». *IEEE Transactions on Software Engineering*, 11(4) :424–436, avril 1985.

[KM90] J. KRAMER et J. MAGEE. «The Evolving Philosophers Problem: Dynamic Change Management». *IEEE Transactions on Software Engineering*, 16(11) :1293–1306, novembre 1990.

[Lay93] N. LAYAIDA. «Gestion de designation dans Guide». Rapport de DEA, IMAG, 1993.

[Lip91] S. B. LIPPMAN. *C++ Primer*. Addison Wesley, 1991.

[Lis88] B. LISKOV. «Dynamic Module Replacement in a Distributed Programming System». *Comm. ACM*, pages 300–312, mars 1988.

[Ma92] J. MAGEE et AL. «Structuring Parallel and Distributed Programs». Dans *International Workshop on Configurable Distributed Systems*, mars 1992.

[MR90] K. MCCLOGHRIE et M. ROSE. «Management Information Base for Network Management of TCP/IP-based internets». RFC-1156, octobre 1990.

[MSS93] M. MANSOURI-SAMANI et M. SLOMAN. «Monitoring Distributed Systems». Rapport Technique DOC92/32, Imperial College, avril 1993.

[ORdP94] I. OUEICHEK et X. ROUSSET de PINA. «An Object-Oriented Model for Distributed System Management». Dans *International Conference on Parallel and Distributed Systems*, pages 418–424. IEEE Computer Society Press, décembre 1994.

[ORdP96a] I. OUEICHEK et X. ROUSSET de PINA. «Dynamic Configuration Management in The Guide ObjectOriented Distributed System». Dans *International Conference on Configurable Distributed Systems.* IEEE Computer Society Press, mai 1996.

[ORdP96b] I. OUEICHEK et X. ROUSSET de PINA. «Using Persistent Objects for Configuration Management in Distributed Systems». Dans *Second IEEE Workshop on System Management.* IEEE Computer Society Press, juin 1996.

[PH91] J. PURTILO et C. HOFMEISTER. «Dynamic Reconfiguration of Distributed Programs». Dans *11Th International Conference on Distributed Computing Systems*, pages 560–571, 1991.

[Pra95] Aiko PRAS. «*Network Management Architectures*». PhD thesis, University of Twente, Pays bas, 1995.

[Ra88] M.A. ROSENSTEIN et AL. «The Athena Service Management System». Dans *Usenix Conference Proceedings, MIT Project ATHENA*, 1988.

[San95] M. SANTANA. «*OC++ Reference Manual*». Unité Mixte Bull-IMAG, mars 1995.

[SD94] I. SOMMERVILLE et G. DEAN. «A Configuration Language for Modeling Evolving System Architectures». Rapport Technique SE/2/1994, Lancaster University, 1994.

[SF91] M. SEGAL et O. FRIEDER. «On Dynamically Updating a Computer Program: From Concept to Prototype». *Journal of Systems and Software*, 14(2):111–128, février 1991.

[SF93] M. SEGAL et O. FRIEDER. «On-the-fly Program Modification: Systems for Dynamic Updating». *IEEE Software*, mars 1993.

[SFDC90] M. SCHOFFSTALL, M. FEDOR, J. DAVIN, et J. CASE. «A Simple Network Management Protocol (SNMP)». RFC-1157, octobre 1990.

[Sha76] M. SHAW. «Research Directions in abstract data structures». *SIGPLAN Notices*, 11, 1976.

[SO86] K.W. SHIRRIF et J.K OUSTERHOUT. «A Trace-Driven Analysis of Name and Attribute Caching in a Distributed System». Dans *6th International Conference on Distributed Computing Systems*, 1986.

[Sol92] R. M. SOLEY. «*Second edition of the Object Management Architecture Guide, containing updated OMG Object Model*». Object Management Group Inc, 492 Old Connecticut Path, Framingham MA 01701, novembre 1992.

[Sta93] W. STALLINGS. *SNMP, SNMPV2, and CMIP.* Addison-Wesley, avril 1993.

[SWH+95] R. SMITH, F. WRIGHT, T. HASTINGS, S. ZILLES, et J. GYLLENSKOG. «Printer MIB». RFC-1759, mars 1995.

[Tho94] Jim THORNTON. «Prescription: A Language for Describing Software Configurations». Rapport Technique 94-18, University of British Columbia, juin 1994.

[Tho96] J. THORNTON. «Practical Description of Configurations for Distributed Systems Management». Dans *International Conference on Configurable Distributed Systems.* IEEE Computer Society Press, mai 1996.

[TI89] R. THOMSON et I.SOMMERVILLE. «An Approach to the Support of Software evolution». *Computer Journal*, 32(5), 1989.

[Zel93] Mark ZELEK. «MIM Think». *CMIP Run !*, 2(1) :1–4, 1993.

Table des matières

Table des figures

Management kernel for a distributed system with persistant objects

This work studies the issues raised by management of distributed systems. It suggests a set of basic services which can be used to build a complete set of management services. Hence, we define a management kernel which consists of an extended configuration management service that implements configuration specification, change management, and monitoring management.

Configuration specification allows us to provide a high-level description of system resources. This specification is written in an extension of the C++ language. The extension is implemented through a class library and a preprocessor which allows the specification of relations among different entities and the specification of notification to be emitted to signal a change in the state of a resource.

We propose an original protocol for configuration change management. It allows the system to accept structural changes (addition, removal, connection and disconnection). The suggested algorithm allows the system to evolve smoothly from a configuration to anther one, without affecting the consistency of applications running on the system. The change is handled with a very low performance hit.

Keywords : distributed systems, system management, configuration specification, change management, change algorithm.

Noyau d'administration pour un système réparti à objets persistants

Ce travail analyse les problèmes posés par l'administration des systèmes répartis. Il dégage un ensemble de services de base permettant la construction d'un ensemble complet de services d'administration. Il définit donc un noyau d'administration constitué d'un service étendue de gestion de la configuration, remplissant à la fois la spécification de la configuration, la gestion des modifications, et la gestion de l'audit.

La spécification de la configuration permet de fournir une description de haut niveau des ressources du système. Cette spécification est décrite grâce à une extension du langage C++. L'extension est réalisée grâce à une bibliothèque de classes et un préprocesseur permettant la spécification des relations entre les différentes entités et celle des notifications qui pourront être émises pour signaler un changement de l'état d'une ressource.

Nous proposons un protocole original de gestion des modifications de la configuration. Il permet au système d'accepter les changements structurels (retrait, ajout, connexion et déconnexion). L'algorithme proposé permet au système d'évoluer d'une configuration vers une autre sans mettre en cause la cohérence des applications qui s'exécutent sur le système, et ceci avec un coût très bas en terme de ralentissement observé.

Mots clés : systèmes répartis, administration, spécification de la configuration, gestion des modifications, algorithme de modification.

THÈSE

présentée par

Ibaa Oueichek

pour l'obtention du grade de

Docteur de l'Institut National Polytechnique de Grenoble

(Arrêté ministériel du 30 mars 1992)

Spécialité : **Informatique**

Conception et réalisation d'un noyau d'administration pour un système réparti à objets persistants

présentée devant la commission d'examen le 23 octobre 1996

Composition du jury

Président :	Jacques MOSSIERE
Rapporteurs :	Mansour FARAH Rachid GUERRAOUI
Examinateur :	Roland BALTER
Directeur de thèse :	Xavier ROUSSET de PINA