Un moniteur d'objets pour un systeme reparti a objets

Celine Valot

INRIA

UNITE DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tel. (1) 39 63 55 11

Rapports de Recherche

N° 985

Programme 3

UN MONITEUR D'OBJETS POUR UN SYSTEME REPARTI A OBJETS

Céline VALOT

Mars 1989

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: (1) 39 63 55 11

Rapports de Recherche

N° 985

Programme 3

UN MONITEUR D'OBJETS POUR UN SYSTEME REPARTI A OBJETS

Céline VALOT

Mars 1989

Un moniteur d'objets pour un système réparti à objets

An object monitor for a distributed object-oriented system

Céline Valot
INRIA, B.P. 105, 78153 Le Chesnay Cédex, France
tel.: +33 (1) 39-63-52-08
telex: 697 033 F e-mail: celine@sor.inria.fr

Résumé

SOS est un système d'exploitation réparti à objets devant permettre l'exécution d'applications réparties de natures différentes. Le système SOS est basé sur l'approche objet et le principe du mandataire permettant l'encapsulation des entités et la construction d'interfaces bien définies.

Le moniteur d'objets est un outil chargé d'observer l'existence et l'évolution des objets à l'intérieur d'un espace d'adressage. L'objectif était de faciliter la tâche de mise au point des programmes SOS. La réalisation de cet outil a permis de mettre en évidence certains aspects problématiques du système SOS. Ce travail est financé en partie par le contrat CEE Esprit 367 "SOMIW".

Abstract

SOS is a general-purpose, object-oriented distributed operating system allowing the execution of distributed applications of different kinds. The SOS operating system is object-oriented and based on the Proxy Principle. Entities are encapsulated and accessible by well-defined interfaces.

The object monitor presented here is in charge of observing the objects life and evolution within address spaces. The goal was to easie debugging of SOS programs. The construction of this tool has highlighted some problems of the SOS operating system. This work is financed in part by the CEC contract Esprit 367 "SOMIW".

Chapitre 1

Introduction

Le projet que nous présentons ici est un moniteur d'objets pour le système réparti à objets, SOS, développé à l'INRIA par l'équipe SOR (Systèmes à Objets Répartis). Le système SOS est un système d'exploitation qui fait partie du projet Esprit 367 "SOMIW", basé sur une approche objet permettant l'unification, la structuration du système en entités bien encapsulées, accessibles par une interface bien déterminée.

1.1 Le système SOS

Le projet SOR a pour but d'expérimenter et de valider l'approche objet dans la conception et réalisation d'applications réparties. L'objectif de SOS n'est pas de cacher la répartition mais d'offrir des interfaces structurées, faisant abstraction du type de données traitées. L'approche objet et le *principe du mandataire* [Sha87b] caractérisent ce système d'exploitation réparti. Une ressource est un groupe d'objets coopérants pour réaliser un service. Un *mandataire* est un objet faisant l'interface entre un client et une ressource. Le mandataire agit comme un représentant de la ressource pour le client.

Cette approche encourage la structuration des applications en objets répartis, par l'intermédiaire de la notion de "groupe d'objets", et leur encapsulation derrière l'interface offerte par les mandataires. Le mandataire constitue un protecteur de la ressource toute entière.

La version de SOS étudiée ici est la version 3 livrée en Juin 1988 aux partenaires du projet SOMIW. Ce prototype fonctionne au-dessus d'UNIX 4.2BSD sur stations Sun. Il est écrit en C++. SOS sera expliqué plus en détail au chapitre 2.

1.2 Les motivations du moniteur d'objets

L'équipe SOR a rapidement éprouvé le besoin de disposer d'outils de mise au point adaptés à SOS.

En effet, certains événements comme la migration d'objets dans laquelle intervient l'*édition de liens dynamique* [Gau87a], le système de multi-tâches [Str82], et les *invocations inter-contextes* [Sha87a] ont pour effet de rendre l'environnement d'exécution très instable. Il n'est pas rare qu'au cours de plusieurs exécutions du même programme SOS dans les mêmes conditions, les résultats obtenus varient d'une fois sur l'autre. Ceci ne facilite pas la tâche de mise au point d'un programme SOS et les outils de mise au point standard d'UNIX, comme dbx, se sont révélés peu adaptés à l'environnement SOS.

Pour pallier à ce problème, plusieurs outils d'aide à la mise au point ont été réalisés, parmi lesquels :

- Un système de traces concernant différents points sensibles d'une exécution, traces qui sont reportées dans un fichier.
- Un système de "replay" pour rejouer une exécution donnée, en réinjectant dans le système les données obtenues à partir des traces.
- L'adaptation de l'outil de mise au point GDB du projet GNU, à l'édition de liens dynamique et au système de tâches.
- Enfin, un "moniteur d'objets", que nous appelerons MNT, pour examiner l'état du système à un moment donné.

1.3 Les objectifs du moniteur d'objets (MNT)

Le MNT est donc chargé de surveiller les objets, en ce qui concerne leur création, leur migration, leur destruction, les relations existantes entre les différents objets, ainsi que les contextes d'exécution. C'est dans ce sens que le terme "moniteur" sera employé dans la suite de ce document.

Dans un système réparti, le manque d'informations sur l'état global du système rend cette tâche difficile [Tan85]. Le modèle retenu pour la réalisation du MNT est l'utilitaire UNIX ps qui rapporte des informations sur l'état des processus actifs.

Etant donné que les informations sur les objets et les contextes sont détenues par le système SOS, le moniteur devra bénéficier de facilités d'accès à ces données.

Un problème se pose au niveau de la mise à jour des informations. En effet, entre le moment *t0* où le moniteur est lancé et le moment *t1* où il est en mesure de fournir les informations demandées, l'état du système a changé. Cependant, il semble plus important de fournir les informations les plus récentes, même si elles ne sont pas forcément à jour.

Une contrainte de réalisation est l'intégration du moniteur dans un système existant, en effectuant le minimum de modifications sur les différents services composant le système SOS.

1.4 Plan du document

Le chapitre 2 donne quelques informations sur SOS. Le chapitre 3 est consacré à la présentation de l'implémentation de cet outil ainsi qu'aux difficultés rencontrées lors de cette réalisation. Cette réalisation a permis de mettre en lumière certains problèmes de la Version 3 du système SOS. Le chapitre 4 est consacré à l'analyse de ces problèmes.

Chapitre 2

Le système SOS

Le système SOS est un système réparti à objets. L'approche SOS est caractérisée par :

- Existence de structures réparties, les domaines, et contrôles inter-domaines;
- Efficacité et encapsulation dans la notion de mandataire;
- Transparence mise en œuvre dans les mandataires;
- Cohérence et dépendances entre les objets par le biais des groupes.

2.1 L'approche objet

Un objet est une double entité, où sont combinées une représentation interne contenant la description des données, et une représentation externe contenant les méthodes permettant de manipuler ces données [Hai87]. Un objet SOS est une instance d'une classe C++. Les méthodes d'une classe permettent aux instances de cette classe de répondre aux invocations effectuées. Un objet commence son existence lors de son instanciation, à l'appel d'une méthode particulière appelée "constructeur". Il est détruit par l'appel à une autre méthode, le "destructeur".

Dans SOS, l'objet est l'unité d'allocation et de communication. Il existe un certain nombre de types d'objets prédéfinis, mais l'utilisateur est libre de créer autant de nouveaux types qu'il le désire. Tous ces types ont le même statut.

Les caractéristiques de l'approche objet sont :

- Evolution et réutilisabilité;
- Les types de données abstraits;
- Encapsulation des données accessibles par une interface déterminée;
- La généricité qui permet de définir un objet sans définir son type, et de fournir le type au moment de l'instanciation;
- La hiérarchie des types est un mécanisme d'héritage qui permet à un objet de partager les caractéristiques d'un deuxième objet;
- L'extensibilité permise grâce aux développements séparés, sans modifications sur le système existant.

2.2 La notion de mandataire

Un *mandataire* est un objet faisant l'interface entre deux domaines. Une ressource est en général composée de deux objets principaux : un

serveur et un *fournisseur de mandataires*, appartenant à un même *groupe*.

Le principe du mandataire stipule *qu'un client ne peut accéder à une ressource, que par l'intermédiaire d'un objet délégué, le mandataire.*

Cet objet est local au client et représente le groupe d'objets formant la ressource. Cette approche encourage la structuration des applications en objets répartis et leur encapsulation derrière l'interface offerte par les mandataires [Sha86b].

La figure 2.1 illustre cette notion. Un contexte et un groupe sont les deux types de domaines de base.

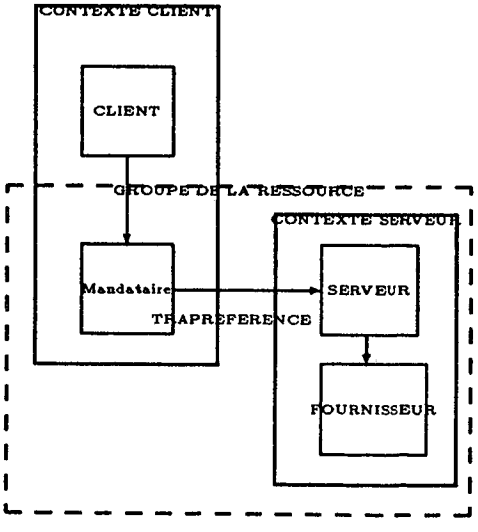

Figure 2.1 : Le principe du mandataire

2.2.1 Notion de contexte, d'accointance, d'OID

Un objet SOS est instance d'une classe C++, qui dérive de la classe de base `sosObject`.

Les objets du système SOS sont instanciés ou "mappés" dans des espaces de mémoire virtuelle, dénommés *context*.[1] Ces objets sont appelés *"accointances"* de ce contexte. Chaque accointance possède un identificateur unique dans tout l'univers SOS. Cet identificateur se dénomme *OID concret*.

La connaissance d'un OID ne donne aucun droit sur l'objet qu'il identifie. L'obtention de cet OID permet d'effectuer une requête d'importation d'un mandataire de l'objet identifié, appelant sa méthode d'exportation, `giveProxy`. Cette méthode vérifie les droits d'accès sur la ressource ainsi désignée et peut refuser l'exportation du mandataire si ces droits sont insuffisants.

2.2.2 La communication dans SOS

Deux objets situés dans le même contexte communiquent par appels de procédures locaux.

Rappelons qu'un client pour accéder à une ressource située hors de son contexte doit importer un objet mandataire représentant la ressource. Le mandataire est exporté dans le contexte client par le *fournisseur* de mandataires. Le client invoque le mandataire qui peut répondre localement à la requête ou la rediriger sur son serveur. Une condition est nécessaire pour qu'ils communiquent : l'existence d'un canal de communication entre le mandataire et le serveur, appelé *TrapReference*. Ce canal ne peut être alloué que s'ils appartiennent à un même groupe. La communication entre ces deux objets situés dans des contextes différents se fait par *invocations inter-contexte*, à l'aide de la primitive `crossInvoke` implémentée par le noyau. A la réception de cette invocation distante, la procédure de `stub` sera appelée, qui décodera l'appel et engagera une action appropriée.

[1] Le mot "contexte" sera employé dans le reste de notre exposé et dénotera cet espace de mémoire.

Le schéma 2.2 représente ces différentes formes de communication.

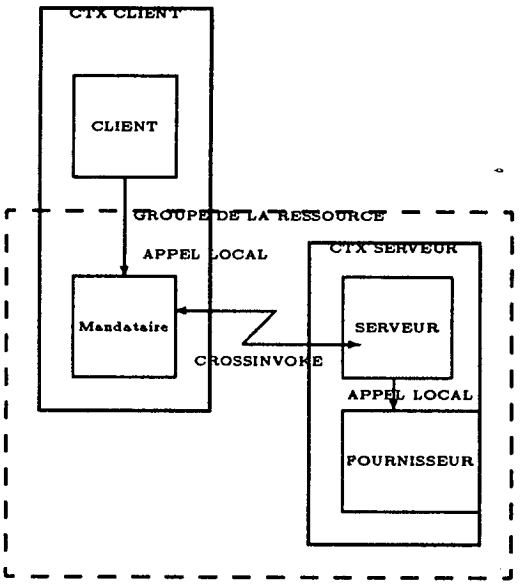

Figure 2.2 : Les différentes formes de communication

2.2.3 La notion de groupe

Une accointance peut faire partie d'un *groupe*, auquel cas elle possèdera un *OID de groupe*. Un *groupe* est simplement un ensemble d'objets ayant un OID de groupe en commun. Une accointance peut appartenir à plusieurs groupes.

2.3 Le noyau et les services systèmes

Sos est un système "ouvert" [Lam79] basé sur un noyau minimal, complété par des services systèmes externes implémentant des fonctionnalités systèmes.

Le *noyau* est chargé de la création et destruction des contextes, de la gestion des tâches et des communications inter-contextes.

Le *Service d'Accointances* est le service gestionnaire des objets SOS. Chaque contexte possède un mandataire de ce service, installé à la création du contexte. Ce service se charge de la création des objets, de leur destruction, leur recherche, leur migration, leur appartenance à un groupe, ainsi que de la gestion des *TrapReferences*.

Le *Service de Nommage* a pour principale tâche d'établir une correspondance entre un nom symbolique et un nom interne (l'objet qu'il désigne).

Le *Service de Stockage* assure la conservation permanente des objets SOS, c'est-à-dire la possibilité de les stocker sur disque et de les retouver après la terminaison d'un contexte.

Le *Service de Communication* se charge de la communication inter-sites. Il implémente une bibliothèque *d'objets protocole* utilisables par les applications. Les mécanismes d'appel de procédure distant, le "Remote Procedure Call", et de diffusion (multicast) sont disponibles [Mak88].

Chapitre 3

Le moniteur d'objets (MNT)

3.1 Le fonctionnement du moniteur d'objets

Trois principaux types d'objets, le *fournisseur de mandataires*, le *serveur* et le *mandataire*, formant un *groupe*, réalisent la ressource MNT. Un quatrième, l'utilisateur de la ressource, ou *client*, est un objet quelconque, qui accède aux fonctionnalités du MNT en invoquant un *mandataire* du MNT qu'il aura importé dans son contexte. Il y a un fournisseur MNT par systèmes, un serveur MNT par machines et un mandataire par contexte.

Un client du MNT peut demander, par exemple, la liste des objets instanciés dans son propre contexte, ou encore la liste des contextes actifs sur son site.

Les connaissances du MNT sur l'état du système ont posé des problèmes. Pour les résoudre en effectuant le minimum de transformations sur le système SOS, nous avons opté pour une solution qui engendre la contrainte suivante :

Pour qu'un contexte, ainsi que son contenu, soient connus du moniteur d'objets, ce contexte doit impérativement importer un mandataire de ce service.

La figure 3.1 illustre cette contrainte : le contexte A et le contexte C ayant importé un mandataire du MNT sont donc connus du serveur du MNT. Par contre, le contexte B n'ayant pas sollicité l'importation d'un mandataire du MNT est inconnu.

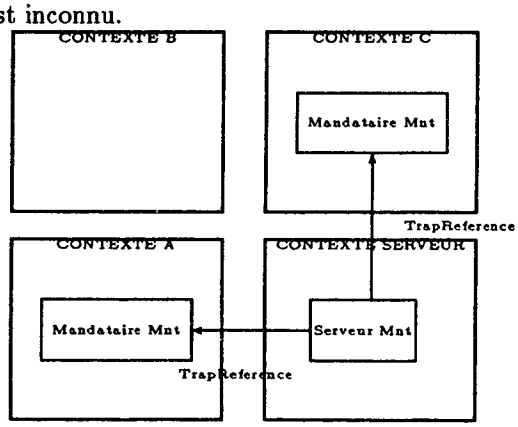

Figure 3.1 : **Les connaissances du Moniteur**

Les objets qui composent la ressource moniteur sont tous des objets SOS, c'est-à-dire connus par le système.

Le rôle du mandataire

Il accède à la table des *descripteurs d'accointances*, contenant toutes les informations disponibles sur les objets, maintenue par le Service

d'Accointances dans chaque contexte. Cependant cette vision est limitée au seul contexte où il se situe.

Sa présence au sein d'un contexte conditionne l'existence de ce contexte aux yeux du MNT. Il est le seul objet de la ressource MNT auquel le client peut accéder.

C'est un *objet migrateur* [Gau87b]; il peut donc être copié ou transporté d'un contexte à un autre.

Le mandataire pris individuellement est limité par son manque d'informations globales, mais c'est l'interaction des différents mandataires qui permet au MNT de connaître l'état global du système.

Le mandataire possède deux possibilités pour répondre aux requêtes d'un client. Si la requête concerne son contexte, le mandataire lui répond directement. Si elle concerne un autre contexte, le mandataire s'adresse au serveur qui redirigera la requête sur le mandataire situé dans le contexte adéquat. La réponse retournera au client par le biais du serveur.

Il faut noter que l'implémentation du mandataire ne fait pas l'objet de deux codes différents pour satisfaire ces rôles.

Le rôle du serveur

Un serveur, dans le cadre du système SOS, est un objet qui redéfinit la procédure de **stub**, permettant de répondre aux requêtes des mandataires.

Le serveur du moniteur ne possède pas d'informations sur les objets mais connaît les contextes lancés sur une machine. Ceci par le biais des *TrapReferences* (cf 2.2.2) qui le lient aux différents mandataires.

Il doit recevoir les requêtes d'un mandataire quand celui-ci ne peut répondre localement à un client.

Il doit aussi déléguer les requêtes reçues, aux mandataires concernés et, au retour, renvoyer les réponses au mandataire du contexte client.

Pour pouvoir s'adresser au mandataire adéquat, il doit gérer le mouvement des mandataires, matérialisé par leur création et destruction. Une destruction est provoquée par la terminaison d'un contexte, que cela soit volontaire ou accidentel.

Remarquons que le rôle du serveur est assez limité dans la réalisation de la ressource MNT.

Le rôle du fournisseur de mandataires

Le fournisseur de mandataires est un objet qui redéfinit la méthode **giveProxy** permettant de répondre aux requêtes d'importation de mandataires d'un client.

Cette méthode **giveProxy** consiste à créer localement le mandataire candidat à la migration et à le préparer à migrer.

L'importation d'un mandataire (avant et après) est illustrée par les figures 3.2 et 3.3.

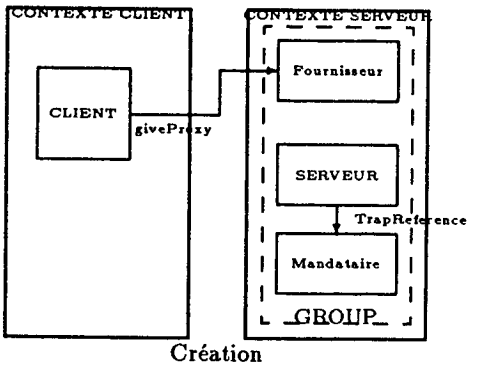

Figure 3.2 : **Avant la migration du mandataire**

Après importation

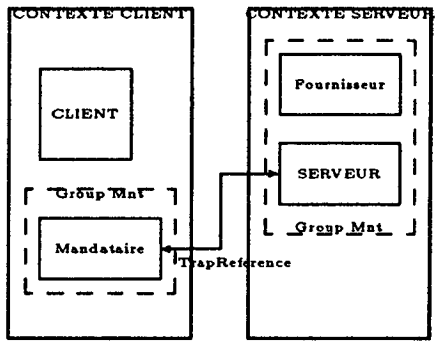

Figure 3.3 : **Après la migration du mandataire**

Le client

Le client est un objet quelconque qui sollicite l'importation d'un mandataire du moniteur et s'adresse à lui par appels de procédures locaux.

Pour conclure, la contrainte que constitue l'importation obligatoire d'un mandataire pour être connu du MNT est une nouveauté dans l'utilisation de SOS. En effet, jusqu'alors, on importait un mandataire dans de but d'utiliser une fonctionnalité d'un service. Cette contrainte entraîne une limitation des fonctionnalités du MNT.

3.2 L'implémentation du moniteur d'objets

3.2.1 Les structures de données

Nous distinguerons les structures de données du Service d'Accointances, et celles propres au MNT.

Les données du Service d'Accointances

Les structures de données matérialisées par, d'une part, les *OIDs* (identificateurs uniques), et les *références* et, d'autre part, les *descripteurs d'accointances* sont à la base de la gestion des objets effectuée par le Service d'Accointances.

A chaque objet instancié dans un contexte correspond un *descripteur d'accointance*, et à chaque contexte une table de *descripteurs d'accointances*. En voici une description simplifiée, en C++ :

```
/* classe des descripteurs d'objets */

class AD{
    friend class mntProxy ;
    friend class mntServer ;

    int valid ;
      /* Concrete OID */
    OID concreteOID ;
      /* Group OID*/
    OID groupOID[MAX_OIDS-1] ;
     // opaque field
    opaqueBytes opaqueField ;
      // reference to code
    class ref codeObject ;
        /* Trap Reference */
    trapTable* trapRef ;
      // direct segment
    segmentDesc directSegment ;
    vtab codeTable ;
       // address method table
    vtab methodTable ;

} ;
```

C'est à la table de descripteurs d'accointances que le mandataire du MNT va accéder dans le but d'obtenir la liste des objets d'un contexte.

```
/* acquaintance descriptors table */
class TAD{
    friend class acquaintanceService ;
    friend class requestDesc ;
    friend class mntProxy ;
    struct AD table[TAB] ;
    AD* getAddress(short) ;

} ;

extern class TAD descTable ;
```

Pour obtenir la liste des contextes, le serveur du MNT doit accéder à ses **trapReferences** désignant tous ses mandataires.

Les données du MNT

Les structures de données utilisées par le MNT sont des listes chaînées évitant l'allocation de tableaux de taille statique. Nous avons deux types de listes différentes : la première pour les OIDs de contextes, la seconde pour les descripteurs d'accointances.

Nous allons maintenant décrire la structure des messages d'invocation et de retour utilisés par les objets du MNT ainsi que les codes qui leur sont significatifs. Un message d'invocation est une classe C++ notée **mntInvoke** qui dérive de la classe de base **invokeMessage**; deux champs ont été rajoutés : l'OID du mandataire destinataire ainsi que la référence du demandeur. Plusieurs constructeurs sont offerts selon le type de l'invocation requise. Le message de retour est matérialisé par la classe **mntReturn** qui dérive de la classe de base **returnMessage**.

Plusieurs codes d'invocation et de retour entre le serveur et le mandataire ont été définis parmi lesquels :

- Codes d'invocations :

  mntAllCtx : demande la liste des contextes;

  mntAll : demande la liste de tous les objets dans tous les contextes.

- Codes de retour :

  mntOK : tout s'est bien passé;

  mntNotKnown : contexte inconnu pour le serveur;

  mntFailed : défaillance du moniteur;

  badOpCode : mauvais code d'invocation.

Voici la structure des messages employés par le MNT :

```
/* code mntInvoke et mntReturn */

struct mntInvoke : public invokeMessage {
    OID who ;
    ref from ;

    mntInvoke(int op, OID x)
     { opCode = op ; who = x ;
       from = nullRef ;}

    mntInvoke(int op, ref r)
     {opCode = op ; from = r ;
      who = nullOID ;}

    mntInvoke(int op)
     {opCode = op ; who = nullOID ;
      from = nullRef ;}
} ;

struct mntReturn : public returnMessage {
    int size ;  // nbr d'objets recus

} ;
```

3.2.2 L'interaction des principaux objets

Nous illustrons ici l'interaction des objets du MNT par un exemple : obtenir la liste des contextes actifs.

C'est une requête mettant en jeu le serveur et le mandataire du MNT. Du coté du mandataire, voici le code de la méthode **getAllCtxs()**, méthode publique accessible par le client :

```
OID* mntProxy::getAllCtxs()
raises(noServer) raises(failed)
raises(unknown)
{
// preparation des messages
  mntInvoke args(mntAllCtx) ;
  mntReturn* result ;
  segmentDesc* buf[2] ;

// allocation segment
  buf[0] = new segmentDesc(0,
           sgAllocate|sgCopyTo) ;
  buf[1] = 0 ;
// l'invocation inter-contextes
  begin
     result = (mntReturn *) crossInvoke
              ((invokeMessage*)&args, buf) ;
  except
// traitement exceptions
       when(badTrapReference)
            raise(failed) ;
       when(crossInvokeException)
            raise(noServer) ;
       when(others)
            raise(unknown) ;
  end
     // decodage du retour
  if(result->retCode != mntOK) {
     raise(failed) ;
  }
  else return (OID *) buf[0]->getAddr() ;
}
```

La première étape de cette méthode consiste en la préparation de l'invocation inter-contextes, le mandataire ne sachant pas répondre à cette requête. Il crée donc une instance du message **mntInvoke**, affecte l'**opCode** avec une valeur prédéfinie en utilisant le constructeur approprié de la classe **mntInvoke** :

```
mntInvoke(int op) {opCode = op ;
      who = nullOID ; from = nullRef ;}
```

Ensuite, il doit préparer un tableau de segments pour recevoir la réponse du serveur constituée par la liste des contextes. Nous n'avons besoin que d'un seul segment, instancié avec les permissions **sgAllocate** et **sgCopyTo** donnant le droit d'allouer et de copier ce segment. Le tableau de segments doit toujours être terminé par un pointeur nul.

Remarquons ensuite l'appel de la primitive **crossInvoke** entouré par un bloc **begin ... except ... end**. C'est le mécanisme des exceptions permettant de traiter des conditions anormales d'exécution [Abr87]. Cette primitive peut lever trois exceptions :

- **badTrapReference** : l'objet appelé n'a pas été trouvé;
- **crossInvokeException** : terminaison anormale du contexte durant l'invocation;
- **others** : toute autre exception levée par l'objet appelé.

Le traitement de l'exception est laissé à l'appréciation du programmeur.

Au retour de l'invocation inter-contextes, on entreprend la phase de décodage du champ **retCode** de la classe **mntReturn**.

Du coté du serveur, à la réception de l'invocation du mandataire, la méthode de **stub** est appelée :

```
void mntServer::stub(invokeMessage* request,
                     returnMessage* reply,
                     segmentDesc** segs)
{
    mntInvoke* req = (mntInvoke *) request ;
    mntReturn* rep = (mntReturn *) reply ;
// decodage de l'invocation
    switch(req->opCode) {

    case mntAllCtx:

// traitement requete
      OID tmp[NCONTEXT] ;
      int nbr = getAllCtxs(tmp) ;

// copie resultat dans segment recu
      segmentDesc sg(0, sgCopyFrom) ;
      sg.assign(tmp, (OIDSIZE * (nbr+1))) ;
      segs[0]->allocate((OIDSIZE * (nbr+1))) ;
      if(segs[0] != 0)
              sg.copyTo (segs[0]) ;

// codage du retour
      rep->retCode = mntOK ;
      rep->size = nbr ;
      break ;

    case mntAll:
        .
        .
      break ;
    default:
      break ;
 }
}
```

Cette méthode prend trois arguments : une instance d'un **invokeMessage**, d'un **returnMessage** et un tableau de segments. Le serveur va décoder le champ **opCode** du message d'invocation pour déterminer l'action à entreprendre; il va appeler la méthode **getAllCtxs()** qui va remplir sa liste des contextes. Ensuite il alloue le segment envoyé par le mandataire et y range le resultat obtenu. L'affectation du champ **retCode** constitue le message de retour, renvoyé au sortir de cette méthode **stub**.

Comment le serveur obtient-il la liste des contextes?

```
int mntServer::getAllCtxs(OID* tab)
{
    mntInvoke what(mntCtxs) ;
    OID* p = tab ;
// parcours table des trapRefs
    for(int i=0 ;
        i < (MAX_TREFS * MAX_TABLES) ;i++) {
        ref r ;
        OID o ;
        begin
            getAD()->trapRef
            ->getTrapRef(i, r) ;
        except
            when(notValid)
                break ;

        end
// verification existence
        begin
            mntReturn* reponse =
             (mntReturn *) crossInvoke(&what,
                        0, i) ;
        except
            when(badTrapReference)
                continue ;
            when(crossInvokeException)
                continue ;
            when(others)
                continue ;
        end
// insertion dans la liste
        r.getCID(&o) ;
        *p = o ;
        p++ ;
    }
    return p-tab ;
}
```

La liste des contextes correspond aux mandataires qui ont été exportés vers les différents contextes. Le serveur y accède par le biais de ses trapReferences qui le lient aux mandataires. Dans un premier temps, nous avions imaginé qu'il suffisait d'obtenir la liste de ces trapReferences. Cependant, il est tout à fait possible que durant le parcours de cette liste, un contexte se termine anormalement. Mais nous pensions que l'invalidation de la `trapReference` correspondante nous permettrait de connaître cette terminaison. Cette invalidation est faite seulement après l'échec d'une invocation inter-contextes vers le contexte terminé. C'est pourquoi nous lançons des invocations vers tous les mandataires désignés.[1] Ainsi, pour tout contexte terminé, le serveur doit engager une invocation pour connaître l'invalidation. Dans le cas d'une invocation qui s'est déroulée sans exceptions, le serveur insère dans sa liste l'OID du contexte correspondant.

Cette méthode est illustrée par le schéma 3.4. Cette solution est lourde et couteûse mais inévitable dans la version 3 de SOS.

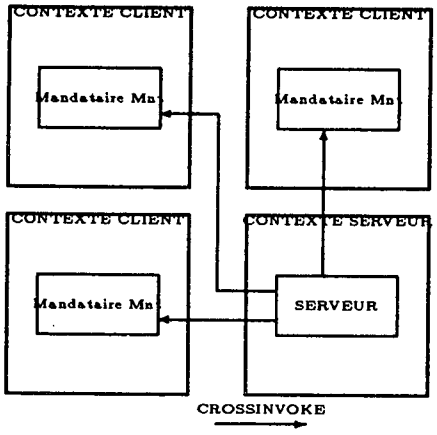

Figure 3.4 : **La recherche des contextes**

[1] Ceci nécessite une procédure de "stub" chez le mandataire qui se contente d'affecter le champ "retCode" avec une valeur appropriée.

3.3 Réfléxions sur le moniteur d'objets

Nous allons maintenant expliquer pourquoi le MNT ne remplit pas le contrat précisé par les objectifs (cf 1.3). Nous décrirons la façon dont nous avions imaginé sa réalisation et les problèmes rencontrés à cette occasion.

3.3.1 Différences par rapport aux objectifs

Le peu de collaboration entre le noyau, le Service d'Accointances et MNT, ainsi que le manque de privilèges de ce dernier, l'ont pénalisé dans sa capacité à répondre aux attentes des utilisateurs.

On peut constater une lourdeur de MNT, un mandataire devant être importé pour être connu de cette ressource sans pour autant en utiliser les fonctionnalités. Ceci renvoie aux utilisateurs la responsabilité des connaissances du MNT. Or, nous pensons qu'un contexte ou processus "surveillé" devrait avoir un rôle passif dans cette tâche.

L'interprétation des résultats donnés par le MNT n'est pas aisée, car il fournit une suite de chiffres (identificateurs uniques, références, Trap-Références). Ceci est dû à des problèmes de désignation dans le système SOS. Un utilisateur souhaiterait plutôt obtenir ces informations sous forme aisément interprétable.

De plus, MNT ne connaît pas les relations existantes entre *groupes* et objets, ni entre *mandataires* et *serveurs* pour la raison suivante : il aurait fallu procéder à des comparaisons d'OIDs, ce qui nous est apparu peu viable à réaliser, de par le coût en terme de ressources. Ce problème ne semble pas avoir de solution à court terme.

Il manque aussi au MNT la vision des *contextes prédéfinis*. Ces contextes sont lancés avant le serveur du MNT et ne peuvent aisément importer

un mandataire de cette ressource. Ce problème exige une redéfinition du statut du MNT et la résolution des problèmes de "boot" des services systèmes.

Le serveur MNT n'est pas au courant de la disparition volontaire ou accidentelle d'un de ses mandataires. Ceci entraîne qu'il est obligé d'effectuer une invocation inter-contextes pour vérifier si le contexte est toujours présent. Nous avons vu que cette méthode de vérification est lourde et couteûse.

3.3.2 Les différentes étapes de la réalisation

A l'origine, le MNT devait être constitué de trois types d'objets principaux (cf 3.1) avec cependant quelques différences concernant le mandataire. La seule tâche du mandataire aurait été de répondre au client. Pour l'accès aux données système de chaque contexte, plusieurs solutions étaient envisageables :

1. soit, des invocations inter-contextes entre la ressource moniteur et les différents mandataires du Service d'Accointances;
2. soit, modifier le code des mandataires du Service d'Accointances pour faire en sorte qu'ils informent le MNT des changements survenus dans leur contexte;
3. soit, mettre les tables de descripteurs d'accointance en mémoire partagée entre les mandataires et le serveur du Service d'Accointances, en donnant un droit d'accès au MNT;
4. soit, posséder, comme le Service d'Accointances, un mandataire du MNT dans chaque contexte.

Le premier choix supposait une modification du code des mandataires du Service d'Accointances, pour que ceux-ci puissent répondre au mandataire du MNT ainsi qu' un mécanisme d'invocations *intra-contexte*. Il y avait cependant un problème de désignation. L'ensemble des mandataires du Service d'Accointances forment un *groupe* et sont tous considérés comme équivalents. Mais, la notion de *groupe* n'offre pas à l'utilisateur la possibilité de sélectionner un objet en particulier, ou bien de s'adresser à tous les membres du groupe. MNT aurait souhaité profiter de la notion de groupe pour s'adresser aux mandataires du Service d'Accointances individuellement ou collectivement. Par ailleurs, il aurait fallu que le mandataire du MNT possède les *TrapReferences* adéquates.

Le deuxième choix entraînait trop de modifications.

La troisième solution, l'utilisation de la mémoire partagée, nous avait semblé la meilleure car elle aurait évité la duplication des informations entre le Service d'Accointances et MNT. Cette solution a été abandonnée au vu des problèmes et des limitations des mécanismes de mémoire partagée du système UNIX 4.2BSD.

La quatrième solution a donc été adoptée. Elle consiste à instancier un mandataire de MNT dans les contextes sur lesquels des informations sont demandées.

Les connaissances du mandataire du MNT ont donc été localisées au contexte[2] ; celui-ci est l'unité de localisation et de gestion des objets. Cependant, nous voulions établir une distinction entre un objet chargé d'observer, que nous appelerons "mandataire privé", et un autre dédié au client, appelé "mandataire client".

Le "mandataire client" aurait été importé uniquement en cas de besoin, sur requête du client.

Par contre, l'objet appelé "mandataire privé" chargé de récolter les informations aurait été *exporté* à l'intérieur des contextes sur lesquels des

[2]Le serveur du Service d'Accointances n'a aucune connaissance globale des objets qu'il gère.

informations étaient sollicitées. Il y aurait eu une nette distinction entre, d'une part, un "mandataire client" représentant la ressource aux yeux du client et d'autre part, des objets internes à la ressource. Le serveur aurait servi à faire la liaison entre ces "mandataires privés" et le "mandataire client". Nous attirons l'attention du lecteur sur le fait qu'une *importation* est à l'initiative du client et qu'une *exportation* est à l'initiative du serveur.

Deux problèmes principaux se sont posés.

- L'accès aux informations relatives aux contextes;
- L'absence d'un mécanisme d'exportation.

En effet, le noyau détient la liste des contextes fonctionnant sur le site, information inaccessible par MNT, sauf en liant celui-ci au noyau, ce qui nous est apparu peu élégant.

Par ailleurs, le mécanisme d'exportation n'était pas encore implémenté par le Service d'Accointances. A ce jour, pour simuler une *exportation*, on envoie à un objet une requête d'exportation qui provoque une action d'importation de sa part.

La deuxième étape a vu l'abandon de l'idée d'*exportation*, et l'unification du "mandataire client" et du "mandataire privé" en un seul mandataire. Nous avons alors voulu instancier dans chaque contexte un mandataire de MNT avant l'appel à la fonction "`main()`" de chaque programme. Ceci aurait évité d'avoir à importer un mandataire de MNT pour être connu de cette ressource. Sa première tâche aurait été d'envoyer au serveur l'OID de son propre contexte puis de répondre aux requêtes des clients. Un problème important s'est posé pour mettre en œuvre ce schéma : en effet, pour que le mandataire instancié au lancement d'un contexte puisse communiquer avec le serveur, il fallait qu'il possède une *TrapReference* adéquate vers ce serveur, ceci n'étant possible que si l'objet serveur existe. Des problèmes de "boot" se posaient donc pour les contextes préexistants à celui du serveur MNT.

Chapitre 4

Quelques réfléxions sur le système SOS

4.1 Les problèmes de nommage des objets

Le nommage des objets au sein du système SOS est à la charge du Service de Noms. Celui-ci établit la correspondance entre un nom symbolique donné par l'utilisateur et la référence d'un objet. Cependant, le Service de Noms ne connaît que ce qu'on lui a demandé d'enregistrer explicitement. Il serait pourtant intéressant que tous les objets SOS soient nommés de façon plus systématique.

Cependant, nous aurions souhaité pouvoir *retrouver un nom symbolique à partir d'une référence*. Autrement dit, savoir si un objet a été précédemment enregistré auprès du Service de Noms.

4.2 La notion de groupe

La notion de *groupe* désigne un ensemble d'objets SOS ayant le même *OID de groupe* qui coopèrent pour réaliser la répartition de la ressource [Sha86a]. Le groupe est extensible à travers sites, car préservé au cours des migrations. Le groupe procure une protection aux objets SOS car deux objets ne faisant pas partie du même groupe ne peuvent communiquer entre eux.

Il existe des défaillances à plusieurs égards :

D'abord, nous aurions aimé pouvoir considérer que les objets appartenant au même groupe soient différents pour pouvoir s'adresser à l'un d'eux en particulier.

Ensuite, nous aurions souhaité que la notion de groupe offre, par exemple, une possibilité de diffusion sur tous les objets du groupe.

Pour conclure, la notion de groupe n'offre pas les fonctionnalités que l'on pourrait en attendre. Elle est un moyen de protéger les objets mais ne donne pas à l'utilisateur la possibilité d'en tirer parti. Nous pensons qu'il devrait y avoir une plus grande adéquation entre les moyens de communication et les mécanismes de protection.

4.3 La communication dans SOS

Dans le système SOS, la communication est possible entre deux objets de contextes différents seulement s'il existe une *TrapReference* entre eux; ceci n'est permis que s'ils sont du même groupe.

Cependant, la méthode `setTrapRef` qui établit une *TrapReference* entre deux objets n'est utilisable que si les objets en cause sont situés dans le même contexte. L'appartenance à un même groupe est vérifiée, en prévision d'une éventuelle migration de ces objets, leur donnant ainsi la possibilité de communiquer.

Mais, il semble justifié de vouloir utiliser la méthode `setTrapRef` pour des objets situés dans des contextes différents. Mais, pour des raisons de protection, cette méthode n'est pas publique. Nous pensons, cependant, que la vérification qui est faite au niveau du groupe est suffisante.

4.4 La gestion des TrapReferences

Dans cette section, nous allons parler de la gestion des `trapReferences` par le Service d'Accointances ainsi que de l'accès offert à l'utilisateur.

Nous avons constaté qu'un objet n'a pas accès à ses propres `trapReferences`. En effet, la méthode du Service d'Accointances permettant cet accès, `ref* getTrapRef(sosObject* ,int)` n'est pas publique.

Ainsi, le serveur MNT doit parcourir la liste de ses `trapReferences` pour choisir le mandataire pouvant répondre à une requête. Ceci n'est faisable qu'en utilisant le mécanisme de "friend" du langage C++, fournissant un accès privilégié à cette méthode. Faciliter l'accès aux `trapReferences` comporte néanmoins un risque : des modifications erronées de la part d'un utilisateur pourraient entraîner l'incohérence de ces données.

Le deuxième problème constaté réside dans la mise à jour de ces `trapReferences` faite par le Service d'Accointances. Une `trapReference` doit être invalidée quand l'objet (ou le contexte) sur lequel elle pointe est détruit volontairement ou accidentellement. Mais cette invalidation n'est effectuée qu'au retour d'une invocation inter-contextes qui a échoué, l'objet ayant été détruit.

Nous avons vu que cela a posé des problèmes pour le MNT entraînant au moins une vérification pour tous les contextes; ce n'est qu'après cette vérification que le serveur pourra tirer parti de l'information (`trapReference` invalidée) et éviter ainsi de nombreuses invocations inter-contextes. Il faut souligner que le Service d'Accointances est au courant de toute destruction d'un objet SOS car le destructeur de la classe `sosObject` est automatiquement appelé. Néanmoins, il est difficilement envisageable qu'à chaque destruction d'objets SOS, le Service d'Accointances recherche tous les objets ayant une `trapReference` positionnée sur cet objet pour l'invalider.

4.5 Le concept du mandataire

Le mécanisme du mandataire est une notion qui a ses avantages et inconvénients. Parmi les avantages [Sha86b] : le mandataire cache la complexité de la ressource; il possède un rôle de capacité car peut implémenter des contrôles d'accès et donc agit comme protecteur de la ressource; il est aussi un moyen de communication sûr et, étant programmable, d'une grande souplesse.

Cependant, nous trouvons que ce mécanisme possède certains inconvénients :

- La conception et l'utilisation d'un mandataire sont relativement complexes;
- Son utilisation est systématique;
- La nécessité d'une importation est assez lourde.

La conception d'un serveur et d'un mandataire pour une application quelconque n'est pas une tâche aisée car la division des tâches entre ces

deux objets n'est pas facile à déterminer. En effet, quelles sont les fonctionnalités que l'on doit mettre dans un mandataire et, quelles sont celles que l'on doit laisser au serveur? Par ailleurs, la mise au point du mandataire est difficile, celui-ci étant un objet migrateur.

L'accès à une ressource passe systématiquement par l'importation d'un mandataire. Nous pensons néanmoins que l'utilisation d'un mandataire peut pénaliser certaines applications, pour des raisons de performances.

En effet, le coût de la migration d'un objet n'est pas négligeable.

En résumé, l'utilisation d'un mandataire se justifie pour les utilisateurs naïfs du système SOS mais peut gêner les utilisateurs intermédiaires, dans la réalisation de certaines applications.

4.6 Le manque de certains mécanismes

Nous allons argumenter en faveur de certains mécanismes qui ne sont pas implémentés dans SOS, dont le besoin s'est fait sentir pour la réalisation de MNT.

4.6.1 Les dépendances

Les dépendances [Hab88] serviraient à coordonner les changements d'états au sein d'un groupe d'objets, quand l'état de l'un d'eux influe sur l'état et/ou l'activité des autres membres du groupe. Si l'on considère deux objets **A** et **B** interdépendants, **A** veut être averti des changements d'état de **B**, pour pouvoir mettre à jour son propre état.

Si l'on considère qu'un serveur et ses mandataires sont des objets interdépendants, l'utilité d'un tel mécanisme ne fait pas de doute. L'interaction dans un groupe serait facilitée. Cela rendrait aussi plus utile la notion de groupe.

4.6.2 Exportation

Un mécanisme d' exportation nous apparaît un point important à développer. Une importation est toujours faite à l'initiative du client, qui ne peut donc se comporter en objet passif. Avec l'exportation, MNT aurait pu exporter un mandataire dans le contexte à observer, sans que ce dernier participe au déroulement de cette tâche.

De plus, certaines applications pourraient mieux fonctionner en effectuant certaines choses de manière beaucoup plus automatique. Par exemple, on pourrait très bien imaginer que le Service de Noms exporte un mandataire dans chaque contexte automatiquement [Hab88].

4.6.3 Un compilateur de procédures d'interface

Le compilateur de procédures d'interface ou "*stubs*" est un outil devant faciliter la programmation des invocations inter-contextes [Gou88]. Une invocation inter-contextes nécessite, de la part du programmeur, l'écriture de procédures de codage et décodage des messages d'invocation et de retour. C'est une tâche répétitive qui entraîne une réécriture de code conséquente. Le compilateur générerait les procédures d'interface nécessaires à la communication des objets comme dans [Jon85].

4.6.4 La protection

Dans le système SOS, il y a deux mécanismes de protection : le *groupe*, et la distinction entre parties privée et publique d'une classe C++.

Nous avons déjà exposé précédemment que les limitations du *groupe* ne permettaient pas à l'uti-

lisateur de tirer parti de ce mécanisme de protection. En C++, le mécanisme de "friend" [Str85] est utilisé pour contourner la protection privé/public.

Il est nécessaire de déterminer quels sont les droits d'accès donnés à, d'une part, l'utilisateur final de SOS et, d'autre part, ceux donnés au programmeur pour permettre l'implémentation de mécanismes de protection adéquats. Cette distinction faciliterait l'intégration de nouveaux services au système existant.

4.6.5 Lancement d'un service à la demande

Un des principes majeurs du système SOS stipule que toute utilisation d'une ressource passe par l'importation d'un mandataire. Cependant, la requête d'importation se solde par un échec quand le service n'est pas lancé.

L'on peut imaginer qu'en cas d'échec d'une requête d'importation dû à l'absence du service requis, un mécanisme de lancement du service serait mis en œuvre, de manière automatique.

4.6.6 Un mécanisme de redirection des invocations inter-contextes

Pour MNT, nous aurions souhaité disposer d'un "mécanisme de redirection des invocations inter-contextes" : un mandataire effectue une requête distante; il passe donc par le serveur MNT qui redirige la requête sur le mandataire adéquat, et reçoit la réponse. Nous aurions aimé que ce mandataire puisse répondre *directement* au mandataire demandeur, sans passer par le serveur. Ceci correspond à un mécanisme de *"forward"*.

Ce que nous pouvons faire actuellement, et ce que nous aurions souhaité pouvoir faire, est représenté par les schémas 4.1 et 4.2.

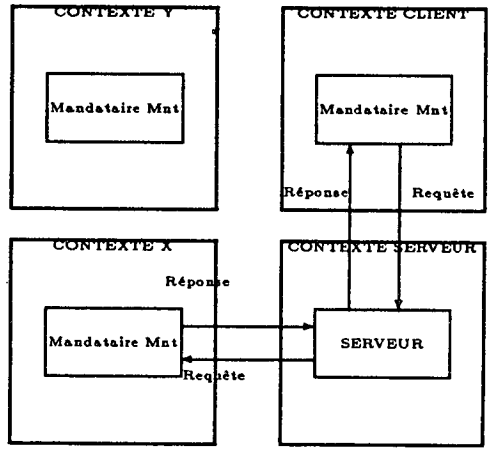

Figure 4.1 : **Réponses du mandataire à un autre mandataire**

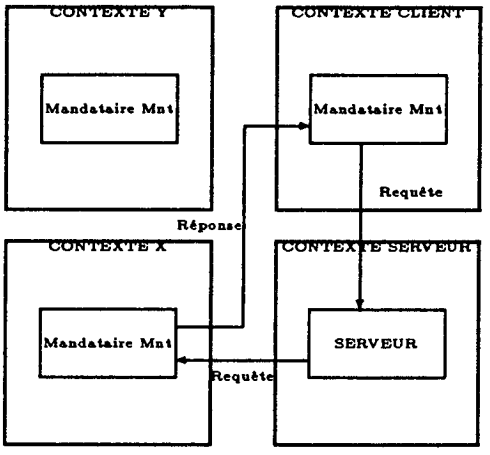

Figure 4.2 : **Le mécanisme de redirection des invocations inter-contextes**

Par ailleurs, un mécanisme de communication asynchrone serait nécessaire dans le cas où un objet ayant effectué une invocation ne souhaite pas recevoir de réponses.

Chapitre 5

Conclusion

SOS est un système d'exploitation réparti, structuré en une collection de sous-systèmes, chacun d'eux étant constitué d'un groupe d'objets communiquant à travers un réseau. L'interface entre un client et de tels sous-systèmes est réalisée par un objet local, le *mandataire*. Tous les accès du client au sous-système passent par cet objet, fournissant par là, un mécanisme de communication sûre. Le principe du mandataire ressemble un peu à celui des *portes* de Chorus [Zim84], à celui de *stubs* [Bir84], ou encore au mécanisme des *capacités* de Amœba [Mul86]. Contrairement à ces derniers, le mandataire est typé et programmable. Le mandataire est plus complexe mais plus flexible et fournit à l'utilisateur une vision uniforme de toutes les ressources, qu'elles soient locales, distantes, ou réparties.

Nous avons décrit le fonctionnement du moniteur d'objets du système SOS ainsi que les problèmes rencontrés durant la phase de réalisation. Cette réalisation a permis la mise en évidence de plusieurs aspects problématiques du système SOS :

- Son utilisation : la frontière entre l'utilisateur naïf et le développeur n'est pas bien déterminée.
- Les dépendances, les exportations, le lancement d'un service à la demande sont des mécanismes qui font défaut.
- L'absence de distinction entre les différentes qualités d'utilisateurs ne facilite pas l'intégration avec les services existants.

Les travaux futurs envisagés consistent principalement en l'adaptation de l'outil de mise au point GDB du projet GNU à l'édition de liens dynamique et au système de tâches.

Bibliographie

[Abr87] Vadim Abrossimov. Exception handling in C++ programs. Dans *Un recueil de papiers sur le système d'exploitation réparti à objets SOS, Rapport Technique INRIA no. 84*, mai 1987.

[Bir84] A. D. Birrell et B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Programming Languages and Systems*, 2(1), février 1984.

[Gau87a] Philippe Gautron et Marc Shapiro. Two extensions to C++: a dynamic link editor, and inner data. Dans *Proc. C++ Workshop*, USENIX, Santa Fe NM (USA), novembre 1987.

[Gau87b] Philippe Gautron, Marc Shapiro, et Sabine Habert. On the use of the dynamic link editor. Dans *Un recueil de papiers sur le système d'exploitation réparti à objets SOS, Rapport Technique INRIA no. 84*, mai 1987.

[Gou88] Yvon Gourhant et Philippe Gautron. *Génération automatique du mécanisme d'importation de proxies.* Note technique no. SOR–24, Projet SOR, INRIA, septembre 1988.

[Hab88] Sabine Habert. *Propositions pour les dépendances.* Note technique no. SOR–14, Projet SOR, INRIA, juin 1988.

[Hai87] Daniel C. Haibert et Patrick D. O'Brien. Using types and inheritance in object-oriented programming. *IEEE Software*, 4(5):71–79, septembre 1987.

[Jon85] M. B. Jones, R. F. Rashid, et M. R. Thomson. Matchmaker: and interface specification language for distributed processing. Dans *Proc. 12th Annual ACM Symposium on Principles of Programming Languages*, pages 225–235, ACM, New Orleans LA (USA), janvier 1985.

[Lam79] B. Lampson et R. F. Sproull. An open operating system for a single-user machine. Dans *Proc. 7th Symp. on O.S. Principles*, pages 98–105, 1979.

[Mak88] Mesaac Makpangou et Marc Shapiro. The SOS object-oriented communication service. Dans *Proc. 9th Int. Conf. on Computer Communication*, Tel Aviv (Israel), October–November 1988.

[Mul86] S. J. Mullender et A. S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29(4):77–100, mars 1986.

[Sha86a] Marc Shapiro. SOS: a distributed object-oriented operating system. Dans *2nd ACM SIGOPS European Workshop, on "Making Distributed Systems Work"*, Amsterdam (the Netherlands), septembre 1986. (Position paper).

[Sha86b] Marc Shapiro. Structure and encapsulation in distributed systems: the Proxy Principle. pages 198–204, IEEE, mai 1986.

[Sha87a] Marc Shapiro, Vadim Abrossimov, Philippe Gautron, Sabine Habert, et Mesaac Mounchili Makpangou. *Un recueil de papiers sur le système d'exploitation réparti à objets SOS.* Rapport Technique no. 84, mai 1987.

[Sha87b] Marc Shapiro, Vadim Abrossimov, Philippe Gautron, Sabine Habert, et Mesaac Mounchili Makpangou. SOS : un sys-

tème d'exploitation réparti basé sur les objets. *Technique et Science Informatiques*, 6(2):166–169, 1987.

[Str82] Bjarne Stroustrup. *A set of C classes for co-routine style programming.* Rapport no. CSRT 90, ATT, Murray Hill NJ (USA), 1982.

[Str85] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, 1985.

[Tan85] Andrew S. Tanenbaum et Robert van Renesse. Distributed operating systems. *Computing Surveys*, 17(4):419–470, décembre 1985.

[Zim84] Hubert Zimmermann, Marc Guillemont, Gérard Morisset, et Jean-Serge Banino. *Chorus: a Communication and Processing Architecture for Distributed Systems.* Rapport de Recherche no. 328, septembre 1984.

Table des matières

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique