Chapitre II

Conception et réalisation de systèmes répartis

II.1 Introduction

Dans ce chapitre, nous présentons notre contribution dans le domaine de la conception et de la réalisation de systèmes répartis. À l'opposé du chapitre précédent, cette partie concerne un domaine plus expérimental − l'élaboration des systèmes. La méthodologie de recherche dans ce domaine consiste à choisir un problème ou un objectif, concevoir une solution ou une preuve de faisabilité, la réaliser de manière la plus efficace et analyser les résultats. Le succès d'un projet expérimental dépend de plusieurs facteurs. D'abord, il faut que le problème ou l'objectif soit choisi avec soin. Par exemple, le problème doit apporter une idée nouvelle ou permettre de prouver ou d'invalider une hypothèse. Parfois, on fait de la recherche exploratoire − en ne disposant pas d'hypothèses à tester, on espère les trouver au cours du projet. Ce type de projet est utile parce qu'il sert à collecter de l'information qui permet ensuite de formuler une question ou une hypothèse. Le facteur temps pour prototyper une solution est important : s'il faut six mois pour prototyper une solution, il est intéressant de le faire ; s'il faut trois ans, le problème peut être dépassé au moment où le prototype est terminé. Un moyen pour réduire ce temps consiste à réutiliser le travail des projets précédents ou une infrastructure existante. L'étape de la mise en œuvre peut être parfois précédée de l'évaluation des choix de conception par des techniques d'évaluation des performances déjà décrites dans le chapitre précédent. Néanmoins, pour beaucoup de systèmes, la mise en œuvre reste le seul moyen pour tester la validité des objectifs et des solutions employées. Ceci est encore plus valable si l'on pense à un transfert technologique ; il est très difficile de transférer des idées vers l'industrie sans une phase d'expérimentation préalable. Le prototype doit être conçu de telle façon que les tests de l'hypothèse ou de l'idée de départ soient facilités (par exemple par insertion de sondes, collection de statistiques). Une fois un prototype opérationnel, il faut l'utiliser, le mesurer et analyser son fonctionnement. Des outils de mesure et des programmes de tests permettent d'évaluer et d'analyser un prototype.

II.2 Conception et réalisation de systèmes répartis

Le développement de la microinformatique dans les années 80 a provoqué l'apparition des réseaux locaux et des stations de travail qui ont remplacé peu à peu les architectures centralisées. Les stations de travail ont été suffisamment rapides pour donner à l'utilisateur toutes les fonctions nécessaires précédemment supportées par un ordinateur central. De plus, cette solution décentralisée était moins onéreuse et avait l'avantage d'offrir à l'utilisateur le contrôle direct de sa station. L'utilisateur pouvait disposer pour lui seul de la capacité de traitement autrefois partagée avec d'autres utilisateurs. En outre, les capacités de traitement d'un système pouvaient être améliorées de façon incrémentale, par exemple un ajoutant une station. Comme il n'était pas justifié économiquement d'avoir des ressources telles qu'imprimantes, serveurs de disques, passerelles réseaux, connectées à chaque station, la solution passait par le partage. Mais ce partage dépendait des moyens rapides de communication constitués par les réseaux locaux. Les premiers réseaux locaux comme *Ethernet* ou *Token Ring*, ont fourni un canal partagé unique au débit élevé de 10 ou 16 Mbits/s et ont défini une politique de contrôle d'accès au canal. À cause de la vitesse de communication, il est devenu possible d'effectuer certaines opérations à distance dont les performances égalent celles des opérations locales. Par exemple, on pouvait accéder aux fichiers distants situés sur une autre machine, lancer l'exécution d'un programme à distance et communiquer avec d'autres utilisateurs. La communication rapide entre des stations a rendu aussi possible un traitement en parallèle où des parties d'un même programme s'exécutent sur des machines différentes [Tanenbaum 85].

Au début, les systèmes répartis ont été basés sur le modèle de processus communicants : une application répartie est constituée d'un ensemble de processus qui communiquent par passage de messages. L'un des premiers systèmes qui a fourni un support d'exécution pour ce modèle était Unix dans sa version *BSD 4.2*. Ce dernier incluait les protocoles de communication de la famille *TCP/IP* et offrait une interface système à ces protocoles en définissant l'abstraction de *socket* [Sun 88a]. Cette abstraction étendait le concept du fichier pour désigner et manipuler un point d'accès au réseau. Le socket peut être vu comme un descripteur de fichier que l'on peut utiliser dans toutes les opérations de fichiers comme *open*, *read* ou *write* (malgré ce polymorphisme d'opérations, les sockets étaient différents des fichiers et certains appels système ont été définis pour manipuler ces nouvelles entités). Les appels système sur les sockets ont permis de programmer des applications en termes de processus communicants par messages. Néannoins, même si le programmeur avait à sa disposition un mécanisme puissant qui permettait de développer des applications réparties, sa tâche était ingrate − voici ci−dessous le code nécessaire pour envoyer un message à un processus serveur désigné par `nom-de-serveur` qui s'exécute sur une machine distante nommée `nom-sa-machine` (les appels système relatifs aux sockets sont en caractères gras) :

```
int  s;
struct   sockaddr_in   mon_sock=AF_INET}
struct   sockaddr_in   son_sock=AF_INET}
int   long_nom;
struct  hostent  *hp;
struct  servent  *sp;
char    nom-ma-machine[20],
       nom-sa-machine[20];
struct   inaddr_in   adr-ma-machine;
struct   inaddr_in   adr-sa-machine;
char   buf[BUFSIZE];

main()
{...
    s=socket(AF_INET,SOCK_DGRAM,0);
    long_nom=sizeof(nom-ma-machine);
    hp=gethostbyname(nom-ma-machine);
    bcopy(hp->h_addr,(char*)&adr-ma-machine,hp->h_length);
    mon_sock.sin_addr=adr-ma-machine;
    mon_sock.sin_port=0;
    bind(s,&mon_sock,sizeof(my_sock));
    long_nom=sizeof(mon_sock);
    getsockname(s,&mon_sock,&long_nom);
    hp=gethostbyname(nom-sa-machine);
    bcopy(hp->h_addr,(char*)&adr-sa-machine,hp->h_length);
    son_sock.sin_addr=adr-sa-machine;
    sp=getservbyname("nom-de-serveur","udp");
    sendto(s,buf,BUFSIZE,0,&son_sock,sizeof(son_sock));
    ...
}
```

La difficulté de programmation était considérable pour plusieurs raisons :

- plusieurs appels système étaient nécessaires pour établir une connexion,
- le programmeur travaillait au niveau des adresses réseau,
- le destinataire des messages devait être connu explicitement,
- malgré l'existence de différents supports de communication par message, la communication entre des systèmes hétérogènes était difficile − la syntaxe et la sémantique des appels système variaient d'un système à l'autre.

Le projet *Epsilon* a proposé des primitives de haut niveau qui facilitent la programmation des échanges de messages entre processus d'application et rendent les programmes indépendants du support système [Bernard 87], [Bernard 89a].

Les systèmes de la génération suivante comme Chorus [Rozier 88], Mach [Accetta 86] et Amoeba [Mullender 86][Mullender 87], en exploitant l'approche *micro−noyau*, ont essayé de pallier ces difficultés d'utilisation des communications dans Unix. Le principe de l'approche micro−noyau et de réduire le système aux fonctions essentielles, et de faire exécuter les fonctions système restantes par des processus utilisateurs ou des processus système. Cette approche nécessite un support de communications accru − même sur une machine, la communication entre processus utilisateurs et serveurs système se fait à l'aide de messages. Ces systèmes ont offert l'abstraction de *porte* qui est une entité dédiée à la communication. Une porte est une file de messages à qui sont associé des droits d'utilisation. Elle est désignée par un nom symbolique, et permet de dissocier l'interface de service de communication de sa localisation physique. Une porte peut en effet migrer d'un site à un autre. La

programmation de communication à l'aide de portes est plus facile que dans le cas de sockets parce que les détails des reseaux sont cachés.

Après le modèle des processus communicants par message, est apparu l'*appel de procédure à distance* (Remote Procedure Call − RPC) [Nelson 81][Birrell 84]. Ce concept a essayé de libérer le programmeur du fardeau de la programmation explicite de communications en fournissant une abstraction analogue au concept d'appel de procédure bien connu dans la programmation séquentielle. L'appel de procédure à distance utilise un support de communication par message et en offrant une abstraction plus élevée, il facilite la programmation des application réparties. Ce concept a été la base des architectures *client−serveur* où un serveur réalise un service défini par une ou plusieurs procédures qui peuvent être appelées à distance par des processus clients. Des exemples notables d'architectures client−serveurs sont des systèmes de fichiers répartis comme NFS [Sun 89] ou AFS [Howard 88].

L'appel de procédure à distance cache les particularités liées à la communication, mais présente toutefois plusieurs inconvénients liés à la sémantique de l'appel dans un milieu réparti et au mode de programmation spécifique. En outre, son caractère purement séquentiel ne permet pas de tirer profit du parallélisme potentiel d'un système réparti. Les problèmes de l'appel de procédure à distance sont discutés plus loin dans la section II.4.

Au milieu des années 80 sont apparus les premiers systèmes à base d'objets comme Eden [Almes 85], Argus [Liskov 85], Emerald [Black 86]et Clouds [Leblanc 88] qui ont enrichi le concept d'appel de procédure à distance et des architectures client−serveurs en fournissant au programmeur une abstraction de plus haut niveau. Ces systèmes toujours d'actualité, exploitent l'approche objet qui a montré beaucoup d'avantages pour la programmation traditionnelle non répartie, à savoir la structuration, la modularité, l'encapsulation, la réutilisation et l'héritage. Il a donc été naturel d'appliquer cette approche au milieu réparti où outre les difficultés de la programmation traditionnelle, le programmeur est confronté aux problèmes de synchronisation, de communication et de répartition géographique des ressources. L'approche objet permet de résoudre de manière propre le problème de désignation qui n'a pas été résolu par l'appel de procédure à distance, ainsi que les problèmes de localisation et de migration. En fait, l'approche objet permet de cacher la répartition pour rendre la programmation des applications réparties indépendante des problèmes de répartition. En même temps, cette approche ne prive pas le programmeur des possibilités du parallélisme et au contraire, elle en permet une expression rigoureuse.

Le besoin de cacher la répartition ressort souvent dans la littérature [Tanenbaum 85] car ce concept facilite énormément les tâches des programmeurs et des utilisateurs. Néanmoins, dans certains cas, il est nécessaire de pouvoir contrôler cet aspect d'une exécution, par exemple quand on veut exécuter un programme sur une machine donnée. Une solution la plus générale est la séparation des mécanismes et

des politiques : il faut des mécanismes qui permettent de cacher la répartition et en même temps, il faut des moyens qui permettent de spécifier comment les mécanismes sont utilisés. Certains systèmes comme Emerald ont contribué à cette approche [Jul 88].

L'un des problèmes majeurs dans la construction des systèmes répartis est celui des performances. La boutade de Tanenbaum citée dans l'introduction montre combien il est difficile d'obtenir des performances satisfaisantes. Les performances globales d'un système dépendent de l'efficacité des mécanismes de base et de leur usage qui est fait par les applications : il faut que les mécanismes de base soient les plus efficaces possibles et qu'ils correspondent bien aux besoins des applications. Les performances de ces mécanismes dépendent de plusieurs facteurs. D'abord, il y a le coût de communication qui, actuellement, est relativement élevé par rapport au seul coût de transfert sur le réseau. Par exemple, les premières versions de Mach étaient à cet égard décevantes (du au fait que les communications avaient été réalisés par un serveur placé dans l'espace utilisateur) [Duda 92a]. Pour transférer un message, il fallait passer plusieurs fois par les frontières de processus. Pour améliorer les performances, la version récente NORMA est revenue à une structuration traditionnelle où les communications sont réalisées dans le noyau. Le coût des communications dépend aussi du nombre de copies qui sont faites et des opérations d'activation de processus responsables du traitement de message. Pour l'appel de procédure à distance, plusieurs facteurs entrent en jeu. D'une part, la linéarisation des paramètres : les structures de données du processus appelant doivent être mises dans un message et reconstituées dans l'espace du processus exécutant. Parfois, ces paramètres doivent être représentés dans une forme indépendante des machines. À cela s'ajoute aussi le coût de transfert du message. Tous ces facteurs font que le coût de l'appel de procédure à distance reste encore relativement élevé.

Au niveau application, les performances dépendent de l'efficacité des mécanismes systèmes qui supportent la communication, mais aussi des politiques d'utilisation de ces mécanismes. Par exemple, nous avons vu dans le chapitre précédent comment des stratégies différentes de duplication d'objets peuvent avoir un impact sur les performances. La grande difficulté consiste à adapter la politique d'utilisation des mécanismes au comportement des applications. Les systèmes à objets semblent être le mieux disposés à répondre à ce genre de problème en séparant les mécanismes des politiques. Un objet est une entité connue et gérée par le système, donc on peut optimiser les mécanismes pour le supporter au mieux. On peut aussi permettre l'adaptation des politiques de gestion d'objets au comportement des applications. Par exemple, on peut exploiter une localité d'accès au niveau d'un objet − comme un objet encapsule des données avec des méthodes d'accès, il délimite une zone qui doit être traitée comme un tout. Aussi, on peut exploiter une localité d'accès au niveau des ensembles d'objets − il existe des objets qui sont souvent accédés ensemble, donc il peut être avantageux de les stocker, de les charger et de les transférer

comme un ensemble. Le fait qu'un objet a une structure connue permet aussi d'optimiser son transfert entre machines hétérogènes. Les coûts élevés de communication et d'accès à la mémoire secondaire jouent en faveur des transferts ou des chargements *en bloc* des objets, les données initialement requises pouvant être accédées par la suite.

La suite de chapitre décrit notre travail au sein de deux projets, à savoir *Epsilon* et *Guide*. Dans le projet *Epsilon*, dont la partie concernant les mesures de systèmes répartis a été décrite au chapitre précédent, nous nous sommes posé le problème de la conception de primitives de programmation des applications réparties sur des systèmes hétérogènes comme Unix ou *MS/DOS*. Dans le projet *Guide*, nous avons travaillé sur les problèmes d'exécution répartie et de communication, ainsi que sur des protocoles de diffusion fiable, des groupes d'objets et sur l'administration des sites.

II.3 Primitives de communication dans *Epsilon*

Comme décrit dans le chapitre précédent, mais du point de vue des mesures, le projet *Epsilon* était une maquette qui permettait de programmer des applications réparties et de les mesurer. Au début, nous nous sommes intéressés à l'aspect mesures : nous avons retenu une configuration matérielle et système, et nous nous sommes vite rendus compte à quel point la programmation des applications était difficile. La maquette comprenait plusieurs *IBM PC* sous *MS/DOS*, un *SM90* sous Unix System V et un *VAX* sous Unix *4.2BSD* reliés par le réseau *Ethernet*. Pour les faire communiquer nous avons utilisé les protocoles de la famille *TCP/IP*, mais leur usage était différent sur ces trois systèmes d'exploitation. En fait, nous avons eu besoin de designer les différents modules impliqués dans une application répartie, ainsi que de déclencher l'émission et la réception de messages entre ces modules, d'une façon aussi simple que possible. D'où cet aspect système d'*Epsilon* : un ensemble de primitives de communication de haut niveau a été conçu et réalisé afin de faciliter l'écriture d'applications réparties. Ces primitives fournissent au programmeur d'applications une interface indépendante du système d'exploitation et des facilités sous−jaçantes de communication entre processus [Bernard 87], [Bernard 89a]. Un programme participant à une application répartie (un *module*) peut être exécuté sur n'importe quelle machine du système sans autre changement dans le code source que les identificateurs des machines où s'exécutent d'autres modules de l'application. L'exemple ci−dessous montre comment on peut programmer l'envoi d'un message en utilisant les primitives :

```
int  iconn;
char   buf[BUFSIZE];

main()
{...
    s_init(0,  0,  0);
```

```
        s_open("nom_de_machine:nom_d_application",     &iconn);
        s_send(iconn,  buf,  BUFSIZE);
        ...
    }
```

Les motivations pour la conception des primitives étaient la portabilité, la flexibilité et la facilité de programmation. Pour être indépendant des systèmes, nous ne voulions pas apporter de modifications aux systèmes d'exploitation. D'autre part, les primitives devaient être flexibles pour permettre de spécifier différents types de communication entre les modules coopérants (en particulier, l'envoi et la réception asynchrone).

Pour la communication nous avons choisi d'utiliser le protocole *UDP* qui est une interface simple du protocole *IP* permettant d'envoyer et de recevoir des datagrammes. Pour des raisons de portabilité, nous avons exclu l'usage direct de la couche liaison d'*Ethernet* − cette mise en œuvre aurait été dépendante du système. Les protocoles utilisés introduisent un léger surcoût par rapport à la couche *Ethernet*, mais on obtient l'indépendance du système. En outre, sur tous les types de machines, nous disposions de ces protocoles. Au dessus de *UDP*, un simple protocole d'arrêt et attente assure la fiabilité de transmission de message.

Nous avons considéré différents modes de communication : l'envoi bloquant (synchrone), l'envoi non−bloquant (asynchrone), un rendez−vous et l'envoi suivi d'une réponse (mode rappelant le RPC). Nous avons décidé de fournir l'envoi non−bloquant (asynchrone), parce qu'en absence de panne du réseau ou des stations, ce mode est suffisant pour le passage de messages et moins contraignant que les autres modes. Par exemple, on peut réaliser facilement un échange de messages synchrones ou un envoi−réponse du style RPC. De même, la réception est asynchrone − s'il n'y a pas de messages, la primitive de réception retourne immédiatement. Une version bloquante et un mécanisme de déroutement logiciel (*upcalls*) [Clark 85] sont également fournis.

La désignation de processus communicants est un problème important et notre objectif était d'utiliser des noms symboliques au lieu d'adresses réseaux ou d'identificateurs de portes. Pour cela les sites du système et les modules d'application sont désignés par leur nom symbolique. Les noms de modules sont choisis librement par le programmeur et ils sont enregistrés localement sur chaque site auprès d'un serveur de site. Ils ne sont utilisés que dans la primitive `s_open` qui ouvre une connexion et renvoie un identificateur de connexion. La demande d'ouverture de connexion est traitée par le serveur de site donné qui active le module désigné. Ensuite, la communication se déroule entre les modules sur la connexion logique désignée par les identificateurs de connexion.

L'ensemble des primitives de communication est complété par des fonctions écrites par l'utilisateur, afin de fournir un puissant mécanisme de déroutement logiciel permettant de réaliser efficacement une communication asynchrone. Le principe de déroutement logiciel consiste à déclarer une fonction qui, au moment de la

réception d'un message, est appelée de façon asynchrone. De cette manière, la structure du programme est bien dissociée en une partie liée à la réception de messages et une autre partie indépendante de la communication.

La primitive `s_open` permet de spécifier les fonctions utilisateurs qui doivent être appelées quand certains événements se présentent. Par exemple, `s_open(f_reception, f_ouverture, f_fin)` permet de spécifier que les fonctions `f_reception`, `f_ouverture`, `f_fin` doivent être appelées quand respectivement, un message nouveau arrive, quand une demande d'ouverture de connexion arrive, et quand une connexion a été fermée par un module distant. L'exemple du code ci−dessous illustre cette fonctionnalité :

site A

```
main()
{   ...
    s_init(...);
    status=s_open("B:exemple",   &iconn);
    if(status == 0)
        /* connexion acceptée */
    else
        /* connexion refusée */
    ...
    s_send(iconn, buf, BUFSIZE);
    ...
}
```

site B

```
main()
{
    s_init(f_reception,  f_ouverture,  ...);
    s_listen();
    while (1)
        s_yield();
    s_exit();
}

f_ouverture  (nom_site,  iconn)
char  *nom_site;
int  iconn;
{
    if (nom_site == A)
        return (1) /* accepte */
    else
        return (0); /* refuse */
}

f_reception  (iconn)
int  iconn;
{
int  long;
char   message[BUFSIZE];

    long = BUFSIZE;
    s_receive(iconn,  message,  &long);
    /* traite le message */
    ...
}
```

Quand la primitive `s_open` est exécutée, on appelle sur le site *B* la fonction `f_ouverture` qui teste si la demande provient du site *A*. Le programme sur le site *B* attend l'arrivée d'un message en utilisant la primitive `s_yield` qui donne le contrôle au protocole de transport. Ensuite, quand le message envoyé par le site *A* arrive sur le site *B*, la fonction `f_reception` est appelée pour traiter le message.

Les détails des primitives sont présentés dans les articles [Bernard 87][Bernard 89a] et dans la thèse [Bernard 89b] ; nous nous limitons ici à donner nos expériences et une critique. Le programmeur qui utilise les primitives d'*Epsilon* doit être conscient de la répartition et doit la gérer − la répartition est visible. Ceci est compatible avec notre but d'offrir des moyens pour programmer des algorithmes répartis. Le programmeur décide par exemple combien de sites et lesquels seront impliqués dans une exécution. Crystal/Charlotte est le projet qui semble être le plus proche des objectifs d'*Epsilon* : la répartition n'est pas cachée, le parallélisme est visible et son usage est facilité par le système [Artsy 87]. Toutefois, dans *Epsilon*, nos machines et systèmes sont hétérogènes ce qui différencie ces deux projets.

Les primitives fourni par *Epsilon* sont très faciles à utiliser parce qu'elles ne nécessitent pas la connaissance d'appels système particuliers pour la communication interprocessus. Le programmeur conçoit son programme à base d'échanges de messages. En outre, les facilités de déroutements logiciels sont à la fois simples et puissantes. Elles permettent de structurer une application en séparant la partie du code responsable du traitement des messages des autres parties. En outre, les fonctions déclarées comme déroutements logiciels sont exécutées par des flots de contrôle asynchrones. Cette facilité de programmation a été vérifiée en pratique − un algorithme réparti d'exclusion mutuelle (Ricart−Agrawala [Ricart 81]) a été très vite développé et mis au point par un programmeur non expérimenté dans la programmation système. Pour évaluer le surcoût introduit par le mécanisme de déroutements logiciels, nous avons programmé un programme de transfert de fichiers. Ses performances ont été comparables aux performances du *Trival File Transfer Protocol*, un programme de transfert de fichiers au dessus de *UDP*, ceci entre deux *IBM PC* et entre un *IBM PC* et un *VAX*.

II.4 Projet Guide

Le travail de conception et de construction de systèmes répartis commencé avec le projet *Epsilon* s'est ensuite prolongé avec le projet Guide. Le projet Guide a pris une direction nouvelle en adoptant un modèle à objets où toutes les abstractions visibles sont des objets. Un objet est une entité autonome qui encapsule des données et le code qui manipule les données. La notion d'objet est une façon élégante et rigoureuse de constituer une unité de nommage, d'exécution, de répartition et de stockage dans un système réparti. Par rapport à la communication par messages et l'abstraction de RPC, la structuration à base d'objets offre un niveau d'abstraction plus élevé où la communication n'est pas visible. De même, la répartition invisible cache les détails de bas niveau comme la localisation des objets et de l'exécution, ces derniers n'étant pas importants pour la plupart des applications. Cette approche met en jeu un système réparti qui englobe plusieurs machines, mais qui est visible de l'extérieur comme une seule machine virtuelle.

Le projet Guide concerne la conception et la réalisation d'un système d'exploitation réparti pour des postes de travail interconnectés par un réseau local [Balter 91]. Les principaux domaines d'application visés sont le génie logiciel (production, mise au point et maintenance de programmes), la communication (courrier, agenda) et la manipulation de documents (édition, échange, stockage). Guide fait partie du projet ESPRIT Comandos (*Construction and Management of Distributed Office Systems*). Nous présentons ci−dessous les principes de son modèle.

Modèle d'objets

Un choix fondamental de Guide est l'adoption d'un modèle d'objets pour la structuration des applications réparties [Krakowiak 90],[Riveill 92],[Balter 94]. Un objet regroupe un ensemble de données, ou variables d'état, et un ensemble de procédures d'accès (appelées aussi *opérations* ou *méthodes*). L'état d'un objet ne peut être manipulé que par un appel à l'une de ses opérations. La notion de *type* permet de regrouper des objets présentant une interface d'accès commune. Un type est réalisé par une *classe* qui spécifie des opérations et une organisation de données communes à un ensemble d'objets, définis comme des exemplaires (ou *instances*) de la classe. Les objets sont persistants, c'est à dire, leur durée de vie n'est pas liée à l'exécution du programme qui l'a créé. Un objet existe tant qu'un autre objet le réfère. Un objet est désigné par un nom interne typé, indépendant de la localisation et appelé *référence système*. Un programmeur d'application manipule les éléments du modèle d'objet par l'intermédiaire d'un langage appelé Guide qui offre l'héritage, le sous−typage et des types génériques.

Modèle d'exécution

L'objectif du modèle d'exécution de Guide est de compléter le modèle d'objets afin de fournir aux utilisateurs les structures d'exécution nécessaires à la conception d'une application répartie : lancement d'activités, création et appel d'objets locaux ou distants, contrôle du parallélisme, communication et synchronisation entre activités [Balter 88][Decouchant 89a][Decouchant 89e][Decouchant 90]. Le modèle d'exécution de Guide présente au concepteur d'application une machine virtuelle multi−sites et multiprocesseurs dans laquelle le parallélisme est apparent et la répartition est cachée.

L'unité d'exécution dans Guide est appelée *domaine.* Un domaine est composé d'un ensemble d'objets et d'*activités* opérant sur ces objets. L'exécution d'une activité dans un domaine consiste en des appels successifs de méthode sur des objets qui sont liés dynamiquement dans l'espace virtuel du domaine. L'appel de méthode d'un objet, qu'il soit local ou distant, est une opération synchrone. Une activité peut créer de nouvelles activités à l'intérieur d'un domaine. Les activités d'un même domaine s'exécutent en parallèle et partagent les objets du domaine.

L'ensemble des objets qui composent un domaine à un instant donné est appelé *contexte* du domaine. La composition du contexte d'un domaine peut évoluer au cours du temps, grâce à des opérations de liaison et de détachement dynamiques. Pour tout domaine, le système gère une structure de données permettant de retrouver les objets de son contexte ; c'est le *descriptif du domaine*. Un objet appartenant à plusieurs domaines possède une entrée dans le descriptif de chacun d'eux. Fig. 2.1 illustre les concepts du modèle d'exécution.

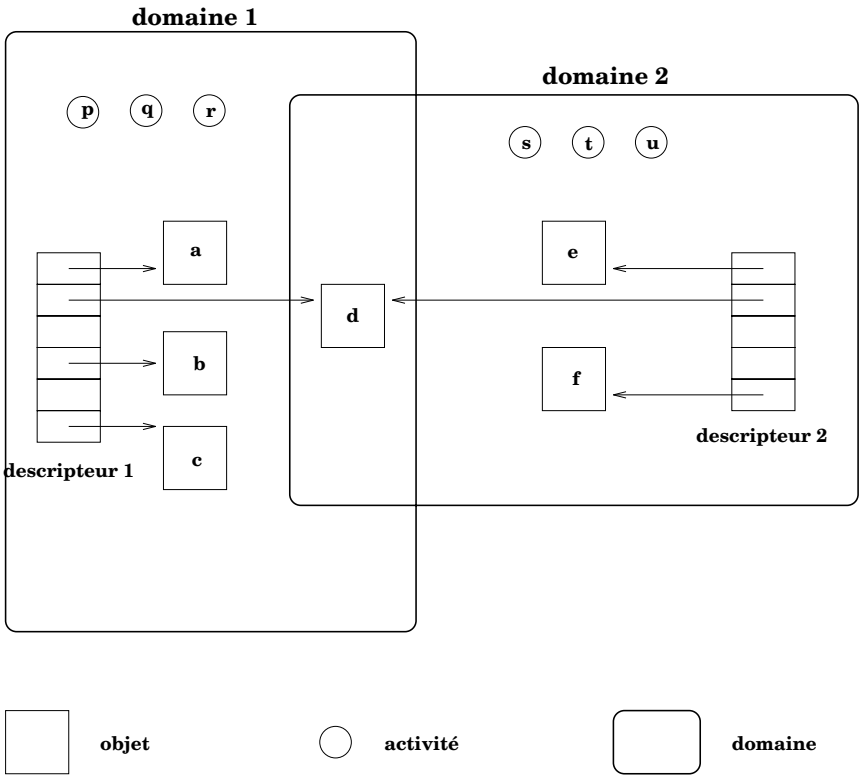

Fig. 2.1 : Concepts du modèle d'exécution

Un domaine peut être créé par un autre domaine (plus précisément, par une activité de ce domaine). L'opération de création spécifie un contexte initial sous la forme d'un objet et d'un point d'entrée sur cet objet. Le domaine est créé avec le contexte initial spécifié; il contient initialement une activité, dite *activité principale*, créée pour exécuter le point d'entrée spécifié. C'est cette activité qui engendrera s'il y a lieu les autres activités du domaine. Une fois créé, un domaine est autonome, mais le domaine qui l'a créé peut le contrôler.

Le langage Guide fournit une primitive d'exécution d'un bloc parallèle de plusieurs activités `COBEGIN–COEND`. Cette primitive bloque l'exécution de l'activité en cours et crée des activités filles pour exécuter chaque opération spécifiée dans le bloc parallèle. La primitive spécifie aussi une condition booléenne portant sur la terminaison ou l'abandon des activités filles. Si la condition est réalisée, l'activité en

cours reprend son exécution. Les blocs parallèles peuvent être imbriqués créant ainsi une arborescence d'activités dont la racine est l'activité principale.

Les objets résident dans une *mémoire permanente d'objets*. Pour réaliser une invocation d'objet, ils sont chargés dans une *mémoire virtuelle d'objets* qui se comporte comme une cache de la mémoire permanente. Les objets sont mono−site, mais il peuvent être déplacés d'un site à un autre. Le modèle d'exécution dissimule la distribution des objets sur différents sites. En principe, n'importe quel site peut être choisi pour exécuter une invocation de méthode sur un objet. En pratique, on tient compte de liaison − si un objet est déjà lié dans un espace virtuel d'une activité, il ne sera pas déplacé sur un autre site. Une meilleure politique serait de choisir le site d'exécution selon les critères de performances et de disponibilité des ressources. C'est le mécanisme de *répartition de charge* qui devrait être responsable de la décision où une méthode doit être exécutée. Pour cela il a besoin d'informations diffusées périodiquement par tous les sites sur la charge instantanée, disponibilité de ressources et autres, et il doit réaliser une stratégie qui a pour but d'optimiser les performances du système, par exemple le temps de réponse ou le débit. Nous avons étudié des mécanismes de répartition de charge, mais malheureusement, ils n'ont pas été réalisés [Lunati 88]. Dans la première implémentation, un objet est toujours chargé sur le site d'exécution, sauf si l'on spécifie explicitement que l'invocation d'objet doit être effectuée sur son site de stockage.

Invocation d'objet

L'*invocation d'objet* est l'opération de base dans le système. L'invocation spécifie la référence de l'objet invoqué, le nom de la méthode et des paramètres. S'il n'est pas encore chargé dans la mémoire virtuelle, il est retrouvé dans la mémoire permanente et chargé sur un site d'exécution. Supposons que l'activité en cours invoque une méthode d'un objet présent sur un autre site. Le système met alors en œuvre un mécanisme qui étend le domaine courant sur ce site. Cette opération est appelée *extension du domaine* et consiste à créer sur le site d'exécution les structures de données nécessaires à la représentation du domaine si elles n'existent pas déjà. Ensuite l'activité appelante s'étend sur le site d'exécution (ce qui sera désigné par le terme *extension d'activité*) : un représentant de l'activité est créé pour exécuter la méthode pour le compte de l'activité appelante. Ce représentant n'est pas détruit à la fin de l'appel et servira les appels ultérieures de l'activité appelante sur ce site.

Modèle de Guide vs. RPC

Nous allons comparer le modèle d'invocation d'objet de Guide avec l'appel de procédure à distance (RPC) selon plusieurs critères.

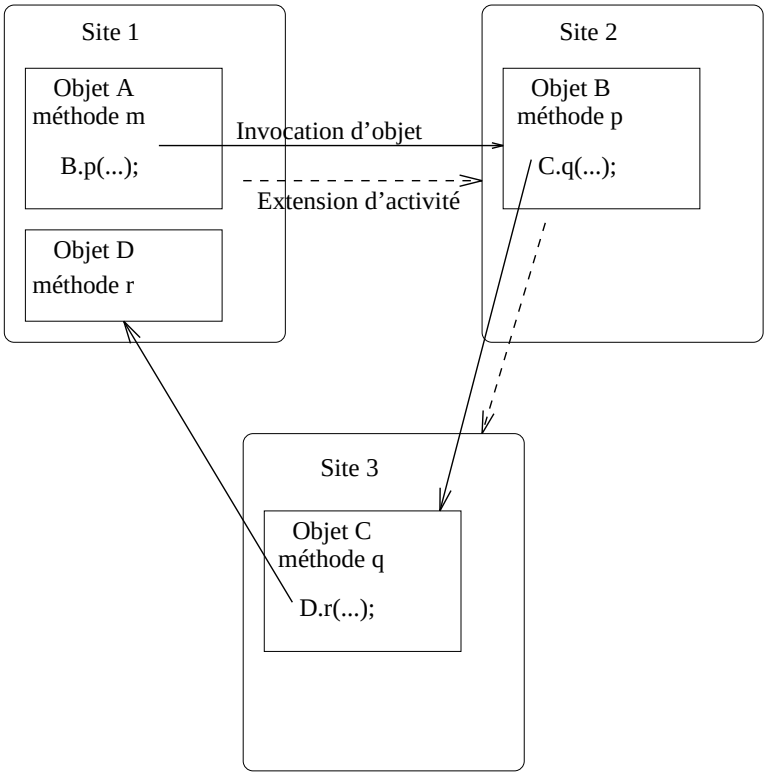

Fig. 2.2 : Invocation d'objet à distance

ouverture

Les concepts d'extension de domaine et d'activité, ainsi que l'invocation d'objet à distance sont des généralisations du concept traditionnel de l'architecture client−serveur et de l'appel de procédure à distance [Wilbur 87]. Le RPC est réalisé dans un univers *ouvert*, celui des sites pouvant être adressés sur le réseau, tandis que l'invocation d'objet concerne un univers *clos*, celui des objets qui peuvent être désignés par des références système. Un client et un serveur qui communiquent par RPC existent sur des systèmes indépendants et autonomes, ce qui nécessite une phase préalable de mise en correspondance (*binding*) par une entité commune et connue des deux. Le RPC est d'habitude fourni au travers de bibliothèques et d'une spécification au niveau d'un langage de programmation et peut être utilisé pour construire des applications réparties sur des systèmes différents. L'invocation d'objet est supporté par l'environnement clos des activités sur des sites Guide (donc, ce mécanisme est placé au niveau du support d'exécution), mais présente un certain nombre d'avantages. En particulier, il offre la portée globale des références objets qui crée un espace global de noms.

invisibilité

Le RPC pose certains problèmes, par exemple d'invisibilité − de manière idéale, un appel local devrait être spécifié et se dérouler de manière identique à un appel distant. Ceci est partiellement vrai pour le RPC. D'habitude, les *talons* (le code responsable du passage de paramètres) sont générés automatiquement à partir d'une description dans un langage de spécification d'interface. Ces définitions doivent être ajoutées aux procédures à exporter. Certains systèmes de RPC imposent des restrictions : par exemple le RPC SUN ne peut avoir qu'un paramètre en entrée et qu'un paramètre en sortie [Sun 88b].

style de programmation

L'usage de RPC impose un style de programmation différent de celui des modèles à objets : les applications sont structurées en termes de serveurs au lieu de l'être en terme de services *et* de données encapsulées [Levy 89]. Donc, dans le cas du RPC, les données à traiter par le serveur doivent être passées en paramètre. Les structures de données complexes doivent être transformées en une suite d'octets, transférées dans un message et reconstruites sur le site d'exécution. Ceci implique des transferts de données qui ne sont pas forcement utiles pour l'exécution. En revanche, l'invocation d'objet est syntaxiquement et sémantiquement identique dans les cas local et distant. Le fait que les données d'un objet sont encapsulées avec le code et le fait de ne passer que des références, impliquent un style de programmation différent où des paramètres sont plus spécifique − ce sont les références des objets à utiliser dans l'invocation. Ceci signifie aussi que seulement les objets nécessaires à l'exécution seront transférés sur le site d'exécution.

passage de paramètres

Un autre problème est lié à la sémantique du passage de paramètres qui dans le cas du RPC est couramment le *passage par valeur*. Dans un milieu réparti où les espaces d'adressage du client et du serveur sont distincts, ce mode de passage implique de copier les valeurs des paramètres à l'appel et au retour. Ce mode de passage peut produire des résultats incorrects si des *alias* de variables sont utilisés (c'est le cas par exemple de la procédure `foo(x:int, y:int)` appelée avec les mêmes paramètres effectifs : `foo(z,z)` ou dans le cas d'un accès concurrent à la variable passée en paramètre [Tanenbaum 89]. La sémantique du passage de paramètres pour l'invocation d'objet est celle du passage par référence qui ne pose aucun problème en milieu réparti.

liaison

Dans les systèmes de RPC, la phase de mise en correspondance entre un client et un serveur (*binding*) est statique et explicite. Donc, il est difficile pour un client de changer de serveurs et pour un serveur de migrer. Avec la liaison dynamique d'objet, ces problèmes sont plus faciles à résoudre − c'est au support d'exécution à assurer les possibilités de migration.

symétrie

Par ailleurs, le RPC associe un client et un serveur de manière asymétrique. Un client demande l'exécution d'une procédure et le serveur l'effectue. Il est difficile d'inverser les rôles, par exemple pour que le serveur appelle une procédure sur le site client. Nous avons vu que l'invocation d'objet est totalement symétrique et il est possible de faire traiter des invocations imbriquées par un seul représentant d'activité (on considère ici les invocations imbriquées, celles dans lesquelles la méthode exécutée à distance invoque à son tour une méthode sur le site d'origine). Comme l'activité appelante est bloquée en attente des résultats elle peut accepter la demande d'exécution, effectuer une invocation d'objet local, envoyer des résultats et continuer à attendre les résultats de son invocation.

Comparaison de Guide avec d'autres systèmes

Plusieurs projets précédents ont choisi d'exploiter une approche objet pour des systèmes répartis. L'un des premiers systèmes était Eden [Almes 85]. Une application dans Eden est construite comme une collection d'objets appelés *Eject* et elle est écrite à l'aide d'un langage dérivé de *Concurrent Euclid* [Black 85]. Les *ejects* sont actifs − ils comprennent un ou plusieurs flots de contrôle qui communiquent par moniteurs. Le nombre de flots de contrôle est limité et sous le contrôle du programmeur. Les *ejects* sont typés, mobiles et l'invocation de méthodes est indépendante de la localisation des *ejects*. Ce modèle d'objet intègre à la fois les aspects d'objets et de support d'exécution au travers d'*eject* qui représente une machine virtuelle multiprocesseurs.

Le projet Argus a mis l'accent sur les aspects de fiabilité d'exécution [Liskov 84][Liskov 85]. Il a étendu le langage CLU pour supporter des transactions atomiques en milieu réparti. Une unité de structuration est le *Guardian* qui encapsule et conceptualise la notion de machine physique. Un *Guardian* contient des objets élémentaire et des processus. La communication entre des *Guardians* se fait par l'appel du gestionnaire de *Guardian* (*handler*). Chaque appel d'un point d'entrée d'un *Guardian* crée une nouvelle activité destinée à traiter la requête correspondante. Il y a donc un nombre variable d'activités s'exécutant en parallèle dans un objet sans que le programmeur puisse exprimer un contrôle sur leur allocation.

Argus et Eden sont des exemples de systèmes supportant *des objets actifs*. Les objets se comportent comme des serveurs multi−tâches où les tâches parallèles partagent un espace commun. L'appel d'une opération d'un objet est considéré comme une requête exécutée par une activité interne à l'objet. Plusieurs requêtes peuvent être servies en parallèle et l'exécution d'une requête peut elle−même créer des activités concurrentes dans l'objet. Ces modèles offrent généralement la possibilité de définir une activité de fond et fournissent des outils pour contrôler le partage des données internes entre les activités concurrentes. Une activité donnée est confinée à un objet. Le flot de contrôle associé à une application est représenté par l'ensemble des activités engendrées par les appels d'opérations (et éventuellement par une tâche de fond).

Le projet Emerald, successeur d'Eden, a introduit un modèle unique d'objets complété par un support pour des types abstraits avec la vérification statique de types [Black 85][Black 87]. Le projet a choisi une solution mixte dans laquelle les objets peuvent être soit *actifs* (dans ce cas là, ils sont le siège d'une activité de fond, une seule par objet), soit *passifs* (dans ce cas, les objets sont utilisables par des activités extérieures). Dans l'approche des objets passifs, on considère deux entités distinctes : les objets et les activités. Une activité est une unité d'exécution indépendante des objets qui lie dynamiquement dans son espace virtuel les objets référencés. Emerald a aussi défini explicitement les notions de localisation et de mobilité des objets − le langage permet d'exprimer et de contrôler la localisation. Emerald ne supportait pas le stockage d'objets individuels dans une mémoire secondaire, la persistance étant réalisée par le stockage de l'image de la mémoire virtuelle sur disque.

L'objectif du projet Clouds a été de développer un environnement réparti qui apparaît comme un ensemble de ressources [Leblanc 88],[Bernabeu 88],[Ahamad 90], [Dasgupta 90], [Dasgupta91]. Le projet a mis l'accent sur la tolérance aux fautes. Le système est basé sur les notions des objets passifs, des invocations d'objets indépendantes de leur localisation et des actions atomiques imbriquées. À la différence de Guide, il n'y a pas de mémoire virtuelle d'objets − chaque objet est couplé dans une mémoire virtuelle séparée qui sert de support d'exécution à un flot de contrôle (*thread*) qui effectue des invocations de méthodes. Ce choix de conception favorise des objets de grande taille.

SOS est un système réparti basé sur l'architecture client−serveur représentée par le principe de *proxy* − l'interface à un service réalisée par un objet partagé est un objet spécial de communication appelé proxy [Shapiro 86]. L'unité d'allocation, de communication et de stockage est l'objet. Le système réalise les mécanismes de base pour la création, l'invocation, la migration et la destruction des objets. Il ne fournit pas de langage spécifique.

Dans Guide, l'approche à objets passifs a été choisie pour deux raisons : d'abord elle paraît plus naturelle dans le contexte d'un système à objets dans lequel une

application est structurée en termes d'appels de services et non de serveurs, un service pouvant être assimilé à l'exécution d'une procédure d'accès à un objet. Il n'y a pas de raison objective d'intégrer chaque service dans un serveur. Donc, un système à objets favorise la programmation en termes client−service plus qu'en termes client−serveur. La seconde raison concerne la granularité des objets. Il est raisonnable d'associer un espace virtuel à un objet lorsque cet objet est de grande taille (c'est le cas des *Guardians* dans Argus, des *Ejects* dans Eden et des objets Clouds). La situation est différente dans Guide car les objets sont de taille variable et peuvent être composés. Associer un espace virtuel à un objet conduirait à un coût prohibitif pour la gestion de petits objets.

Les modèles les plus proches de Guide sont ceux d'Emerald et de Clouds. Néanmoins, à la différence d'Emerald, Guide défini une mémoire permanente d'objets − un support de stockage. Par rapport à Clouds, le modèle de Guide favorise les objets petits (comme la plupart des fichiers dans un système traditionnel) qui peuvent être combinés pour former des objets plus gros. Un aspect original du projet est aussi son modèle d'exécution répartie. Enfin, Guide présente une solution intégrée en fournissant ces concepts au sein d'un modèle homogène ; ce modèle étant mis en œuvre par un environnement complet allant du langage au système et destiné à devenir un produit industriel.

La première phase du projet (Guide−1) a consisté à développer un prototype opérationnel au dessus d'Unix pour montrer la viabilité des principes de conception et leur adéquation aux domaines d'application visés. L'accent a été mis sur l'intégration forte entre le langage et le support d'exécution qui apparaissait comme un *run−time* [Decouchant 88],[Decouchant 89a],[Decouchant 89b],[Decouchant 89d],[Decouchant 89e],[Balter 90],[Decouchant 90],[Decouchant 91],[Balter 91], [Krakowiak 90]. La deuxième phase (Guide−2) a étendu le système au support de plusieurs langages et a introduit plus de séparation et de modularité entre différentes parties du système. Certains aspects négligés dans la première phase (sécurité, protection, tolérance aux fautes, administration). ont été repris et traités [Boyer 91],[Freyssinet 91a],[Freyssinet 91b],[Lacourte 91a],[Lacourte 91b],[Chevalier 92a],[Chevalier 92b],[Chevalier 92c],[Hagimont 92],[Chevalier 93a],[Chevalier 93b],[Hagimont 93],[Chevalier 94],[Hagimont 94] (le lecteur peut aussi se référer à [Riveill 93] et [Rousset 93]).

Au sein du projet Guide nous avons participé aux activités qui sont décrites dans la suite, à savoir: la gestion d'exécution répartie, les mécanismes d'invocation d'objets à distance, les protocoles de communication, la gestion de groupes d'objets et l'administration de sites.

II.4.1 Exécution répartie et communication

Dans cette section, nous allons présenter les aspects relatifs à l'exécution répartie et à la communication dans Guide−1.

Gestion de domaines et des activités

L'implantation du système Guide au dessus d'Unix utilise un processus par site pour représenter chaque activité [Balter 91]. Ces processus sont appelés *représentants d'activité*. Ils sont créés dynamiquement au fur et à mesure que l'activité se déplace de site en site (en effectuant l'extension d'activité). La gestion des domaines et des activités sur un site est réalisé par un processus spécial appelé *serveur Guide*. Les serveurs Guide coopèrent pour réaliser les opérations demandées, par exemple, l'extension de domaine ou d'activité. Les représentants d'une même activité communiquent directement entre eux pour réaliser des invocations d'objet à distance. Un domaine étant une unité d'allocation de ressources, il est représenté comme des tables gérées par un serveur Guide. Les domaines et les activités sont visibles comme des instances des objets systèmes. On peut les contrôler en invoquant des méthodes comme *create*, *start, suspend*, *resume* et *destroy.*

Invocation d'objet

L'exécution répartie est basée sur le mécanisme de l'invocation d'objet à distance. Au moment du premier appel de méthode sur un objet situé sur un site distant, l'extension de domaine et d'activité a lieu : un représentant de l'activité est créé par le serveur Guide distant. Ensuite, le représentant effectue une invocation d'objet local pour le compte de l'activité. Ce schéma permet d'effectuer des invocations imbriquées sur des sites différents et sert de support à toutes les opérations à distance du noyau comme par exemple le chargement d'objets à distance ou le déverrouillage d'un objet chargé en mémoire sur un site distant [Decouchant 90].

Pendant le déroulement d'une exécution distante, le représentant de l'activité sur le site appelant est suspendu jusqu'à la réception des résultats: en effet, l'exécution d'une activité est purement séquentielle et, à un instant donné, seul le dernier contexte appelé est actif. Cela veut dire que tous les représentants d'activité sauf un sont bloqués. Ils attendent ou bien le retour d'invocation de méthode qu'ils ont eux−même effectué, ou bien une demande d'exécution de méthode venant d'un représentant d'activité sur un autre site. Cette propriété permet de n'utiliser qu'un seul processus Unix pour représenter une activité donnée sur un site donné. Soient S_1, S_2, ..., S_n les sites visités par une activité A, et P_1, P_2, ..., P_n les processus Unix représentant cette activité sur ces sites. $P_1, P_2, ..., P_{n-1}$ sont bloqués en attente de retour d'appel et P_n est actif. Si dans le code de la méthode en cours d'exécution par

l'activité *A* on a un appel à une méthode sur un objet de l'un des sites déjà visités S_k, c'est le processus P_k qui sera chargé de l'exécution de la méthode sur l'objet.

L'invocation de méthode sur un objet est réalisée par la primitive `ObjectCall` qui reçoit un bloc de paramètres contenant les références de l'objet et de sa classe, ainsi que les paramètres de l'appel. Si l'objet appelé est distant, l'invocation consiste à transférer le bloc de paramètres de l'invocation au représentant adéquat de l'activité appelante. Celui−ci exécute la méthode localement à l'aide de la primitive `ObjectCall` et ensuite retourne les résultats à l'activité appelante. Pour transmettre des paramètres et recevoir des résultats, un protocole de communication travaillant en mode requête−réponse décrit plus loin est utilisé.

Le passage des paramètres d'une invocation se fait soit par valeur dans le cas des objets élémentaires (comme les entiers ou les chaînes de caractères), soit par référence dans le cas des objets non élémentaires. Les paramètres sont passés dans un bloc indiquant pour chaque paramètre un code (valeur ou référence) et selon ce code, soit la valeur soit la référence. Comme les sites d'exécution peuvent être hétérogènes, il faut prévoir les mécanismes qui permettent de représenter des données d'une façon indépendante des machines utilisées. La solution proposée est de coder chaque paramètre en standard XDR (*External Data Representation*) [Sun 86] avant de les transmettre.

Le bloc de paramètres est expédié dans un message requête transmis au représentant d'activité sur le site distant. L'arrivée du message sur le site distant débloque le processus qui décode les paramètres avant d'effectuer le `ObjectCall` local. Après le `ObjectCall`, le bloc de paramètres est codé et expédié dans un message réponse au représentant d'activité appelante. Finalement, du coté de représentant de l'activité appelante, le bloc de paramètres reçu est décodé.

Communication

Pour réaliser le prototype Guide, nous avons eu besoin de différentes primitives de communication. Premièrement, il faut un protocole de communication fiable en mode requête−réponse : un message est envoyé à une destination et on attend une réponse, pas nécessairement de la même destination. Ce protocole est utilisé par les représentants d'activité pour l'invocation d'objet ce qui nécessite une communication du type requête−réponse. Cette sémantique permet d'optimiser le protocole pour le délai minimal. Deuxièmement, les serveurs Guide ont besoin de transmission fiable de datagrammes, parce qu'ils traitent les opérations sur les domaines et les activités de manière asynchrone. Finalement, le chargement d'objet à distance et la migration nécessitent un protocole optimisé pour le débit.

Pour supporter ces trois modes de communication, nous avons conçu un protocole de communication spécialisé et l'avons réalisé au dessus de *UDP/IP* (le protocole en mode déconnecté sous Unix *4.2BSD*).

L'invocation d'objet à distance aurait pu être réalisée sans faire appel à un protocole spécialisé (en utilisant par exemple le protocole *TCP/IP*, travaillant en mode connecté, disponible sous Unix). Néanmoins, plusieurs raisons nous ont incité à considérer l'utilisation d'un protocole spécialisé. Tout d'abord la nature de la communication requise présente un caractère requête−réponse. Cela veut dire qu'en absence d'erreur de transmission, la réponse à une requête est en même temps la confirmation de la bonne réception de la requête. De même, une nouvelle requête est la confirmation de la bonne réception de la dernière réponse. Donc, dans le cas favorable, seulement deux transferts de messages sont nécessaires pour réaliser une invocation d'objet à distance. Deuxièmement, un protocole spécialisé peut être réalisé au dessus d'un protocole non fiable et performant, par exemple directement au dessus d'*Ethernet*. Le contrôle d'erreur doit en effet être réalisé par la couche supportant l'invocation d'objet (c'est l'argument du contrôle d'erreur de bout en bout [Saltzer 81]). Le temps d'invocation est donc minimisé par rapport à l'utilisation des protocoles en mode connectés. Un autre aspect jouant en faveur de l'utilisation d'un protocole spécialisé est la limitation qui est imposée par Unix sur le nombre de sockets ouverts simultanément.

Notre protocole de communication supporte un service de transport fiable similaire au protocole supportant le RPC proposé par Birrell et Nelson [Birrell 84]. La différence réside dans la façon de récupérer les erreurs de transmission (on a choisi la retransmission sélective en cas d'erreurs ou de pertes de paquets) et dans le fait que le protocole peut supporter des invocations imbriquées sur des sites différents. Le protocole ressemble aussi à un sous−ensemble de fonctionnalités offertes par *VMTP* (*Versatile Message Transfer Protocol*) qui a été conçu spécialement pour la communication inter−noyau dans un système réparti sur un réseau local rapide [Cheriton 86].

Le protocole est responsable de la détection des erreurs de transmission, de la retransmission de paquets perdus et de la suppression des paquets dupliqués. Il utilise des interruptions d'horloge pour gérer des événements temporels et des interruptions d'entrée/sortie pour le traitement de messages en réception. Le schéma d'adressage d'Internet est utilisé. Le protocole est réalisé comme une bibliothèque de procédures qui sont liées aux programmes utilisateurs, actuellement ce programme utilisateur est le noyau de Guide−1.

Performances

Pour évaluer les performances du mécanisme d'invocation d'objet à distance, nous avons mesuré le temps d'exécution d'une invocation locale et distante de méthode vide, et avons comparé ce temps avec celui d'un appel de procédure vide en local et à distance − RPC (les mesures ont été faites sur deux *SUN* 3/60 connectés par l'*Ethernet* à 10 Mbits/s). La table ci−dessus présente les résultats.

Opération	*temps* (*ms*)
appel de procédure local	0.011
invocation d'objet local	0.409
RPC SUN	11.000
invocation d'objet à distance	6.960

On peut constater que l'invocation d'objet est beaucoup plus coûteuse qu'un appel de procédure en local à cause de l'interprétation des invocations pour la liaison dynamique, mais dans le cas distant, l'invocation d'objet est plus performante.

II.4.2 Groupes d'objets

Dans la suite du projet Guide, notre travail a concerné l'aspect communication pour Guide−2. Certains besoins de communication au niveau du système et des applications révélaient l'importance de la communication de groupes, comme la diffusion (*multicast*) [Duda 92a]. Par exemple, la coopération entre applications, la gestion de données dupliquées ou la gestion du parallélisme ont souvent besoin d'acheminer le même message vers plusieurs destinations. Ce type de problème peut être vu comme un problème de gestion de groupes dont les membres doivent effectuer les mêmes opérations. Des protocoles de *diffusion fiable et ordonnée* sont particulièrement intéressants parce que leur propriétés fortes facilitent les tâches de conception et de mise en œuvre des mécanismes de groupes. En outre, beaucoup de travaux récents se sont focalisés sur ces protocoles et plusieurs protocoles performants ont été proposés [Birman 87],[Chang 84],[Kaashoek 91],[Melliar−Smith 90],[Peterson 89]. Un mécanisme de groupes à base d'un protocole de diffusion fiable et ordonnée peut servir de base pour la tolérance aux pannes, la duplication d'objets ou le parallélisme [Bal 89]. Donc, nous nous sommes impliqués dans la conception d'un protocole de diffusion rendu visible au travers d'abstractions qui permettent de l'intégrer au modèle d'objets et d'exécution de Guide.

Protocole de diffusion

Un protocole de diffusion fiable et ordonnée permet d'envoyer le même message simultanément à un ensemble de destinataires. Le protocole garantit les propriétés sémantiques concernant la réception des messages :

atomicité

soit tous les destinataires reçoivent le message, soit aucun,

ordre des réceptions

différents ordres possibles : *FIFO*, causal, total,

résistance aux pannes

les deux propriétés précédentes continuent d'être assurées même en présence d'au plus *K* pannes de sites.

Les méthodes utilisées pour assurer ces propriétés sont souvent coûteuses :

- l'atomicité est réalisée par un protocole à trois phases : la diffusion du message, la réception des acquittements et la diffusion de la validation du message [Birman 87] ;
- l'ordre est en général réalisé à l'aide de l'estampillage par un séquenceur qui valide le message après avoir reçu les acquittements [Chang 84],[Kaashoek 91] ;
- la résistance aux pannes est assurée par le stockage sur au moins *K* sites de chaque message reçu et non acquitté [Chang 84],[Kaashoek 91].

L'ordre de réception peut être différent selon les besoins. Par exemple, on peut assurer un ordre total, causal ou *FIFO* (illustrés dans la Fig. 2.3). L'ordre total est de loin le plus utilisé parce qu'il assure une propriété très forte permettant de réaliser différents mécanismes de duplication. L'ordre causal est un ordre partiel induit par la dépendance entre les événements de réception et d'émission d'un message. L'ordre *FIFO* consiste à assurer que si deux messages sont émis successivement depuis un site, ils sont reçus sur un autre site dans le même ordre (l'ordre des réceptions coïncide avec leur ordre d'émission).

Les principales différences entre les protocoles sont les propriétés assurées et les méthodes pour assurer l'ordre. Dans le protocole de Chang et Maxemchuk [Chang 84], les stations sont organisées en anneau virtuel ; chaque station tient le rôle du séquenceur pendant une durée limitée. Le protocole assure l'atomicité et un ordre total. Dans le protocole de Kaashoek et Tannenbaum [Kaashoek 91], le séquenceur est statique et il garantit aussi l'atomicité et l'ordre total. *ISIS* offre plusieurs protocoles − entre autres *ABCAST* qui garantit l'ordre total en utilisant une procédure à trois phases et *CBCAST* qui garantit l'ordre causal en utilisant des horloges vectorielles [Birman 87]. Une approche différente est prise par le protocole de Melliar−Smith, Moser et Agrawala qui garantit un ordre total avec une probabilité qui tend vers 1 avec le nombre d'envois de messages [Melliar−Smith 90]. L'inconvénient de ce protocole est le délai important que subissent les messages avant d'être délivrés (par exemple, dans le cas d'un réseau de 10 stations, il faut en moyenne recevoir 7 messages avant de pouvoir décider de l'ordre du premier).

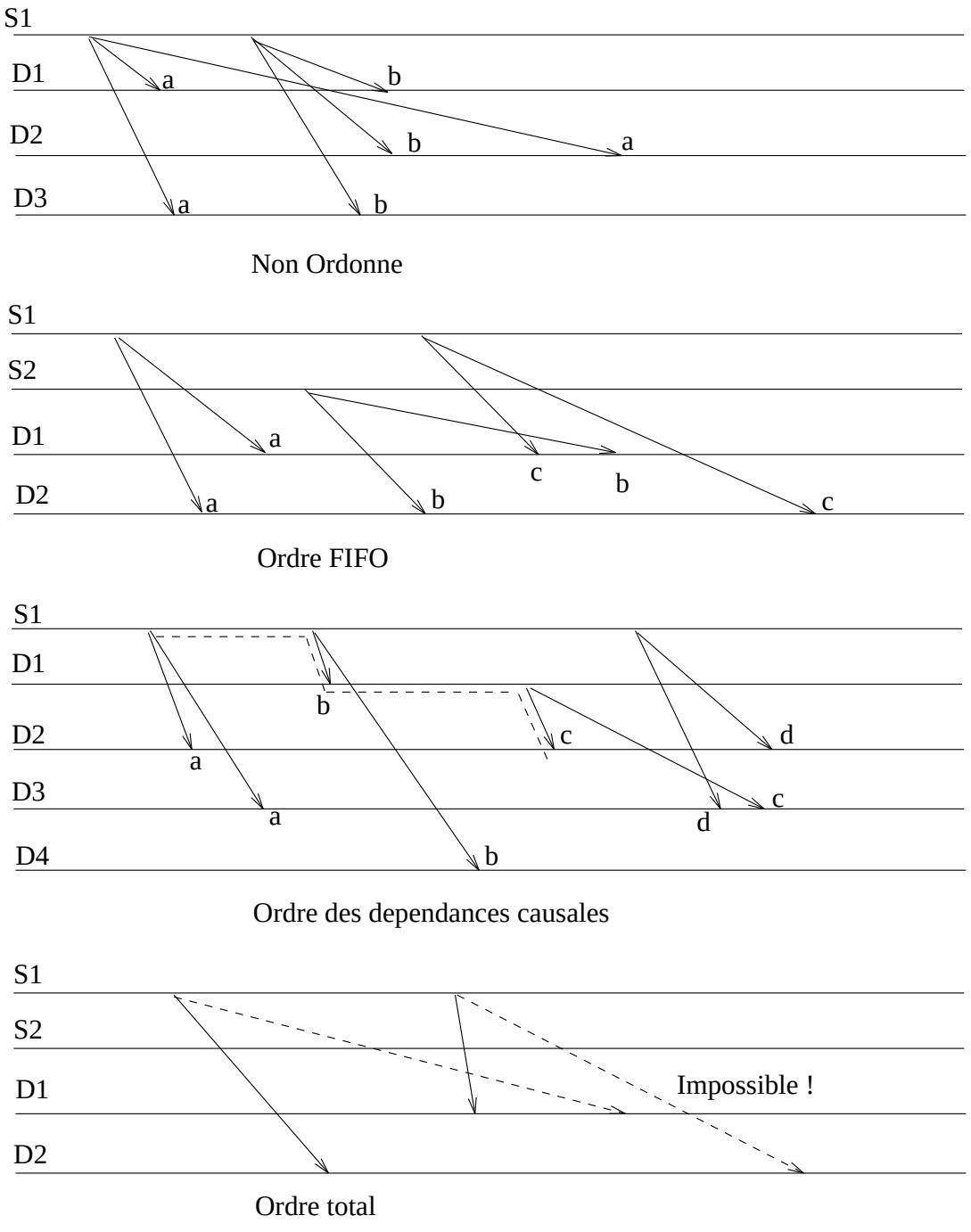

Fig. 2.3 : Différents ordres possibles

D'habitude, on défini un groupe comme un ensemble de processus destinataires de messages coopérant à une même tâche répartie. Un groupe est géré par des

primitives qui permettent la création et la destruction du groupe, ainsi que l'adhésion et le retrait d'un membre. Selon qu'un processus qui n'est pas membre du groupe peut ou non y envoyer un message, le groupe est dit *ouvert* ou *fermé*. Deux autres types de groupes peuvent être distingués − groupes *explicites* ou *opaques*. Les membres d'un groupe explicite connaissent à tout instant les membres du groupe et tous les changements d'appartenance au groupe sont vus dans le même ordre par tous les membres. Ce type de groupe est obligatoire pour les groupes fermés, parce que pour délivrer un message, chaque membre doit savoir qui est membre du groupe. Ce type de groupes est nécessaires pour certaines application, par exemple un calcul parallèle divisé en *n* parties qui sont exécutées par *n* membres du groupe. À l'opposé, les membres d'un groupe opaque ne connaissent pas la composition de celui−ci. Comme ils ne nécessitent pas une procédure répartie d'accord sur les changements d'appartenance, les groupes opaques sont plus faciles à réaliser.

Un protocole de diffusion fiable et ordonnée pour Guide a été conçu et réalisé [Veillard 92a][Veillard 92b]. Notre objectif était de fournir un protocole garantissant l'atomicité, l'ordre total et la résistance aux pannes. En outre, pour des raisons d'efficacité, ce protocole devrait faire usage de la diffusion physique dont disposent certains réseaux locaux comme *Ethernet*. Nos choix de conception les plus importants ont été :

- le protocole est de niveau bas, donc la notion de groupe de processus est traduite par la notion de groupe de sites (sites où sont situés les processus) ; les entités destinataires des messages au niveau du réseau sous−jacent, sont en effet des sites (un gérant du site peut ensuite délivrer un message à leur destinations présents sur le site),
- pour des raisons d'efficacité, le principe du protocole est basé sur l'usage d'un séquenceur (comme dans le protocole de Kaashoek et Tanenbaum),
- les groupes sont ouverts et opaques (on peut toujours construire un service de groupes explicites au dessus des groupes opaques) ; ce choix permet d'avoir une fonctionnalité minimale pour une meilleure efficacité,
- la gestion de groupes est réalisée au−dessous du protocole de diffusion.

Les choix de conception suivent le principes des micro−noyaux − une fonctionnalité minimale est réalisée à un niveau bas et si des fonctionnalités plus évoluées sont nécessaires, elles sont réalisées dans les couches plus hautes. Par exemple, si certaines applications ont besoin de connaître explicitement l'appartenance à un groupe, on peut exploiter les propriétés du protocole de diffusion pour établir cette connaissance (ce qui d'ailleurs est fait pour la gestion de sites décrite plus loin).

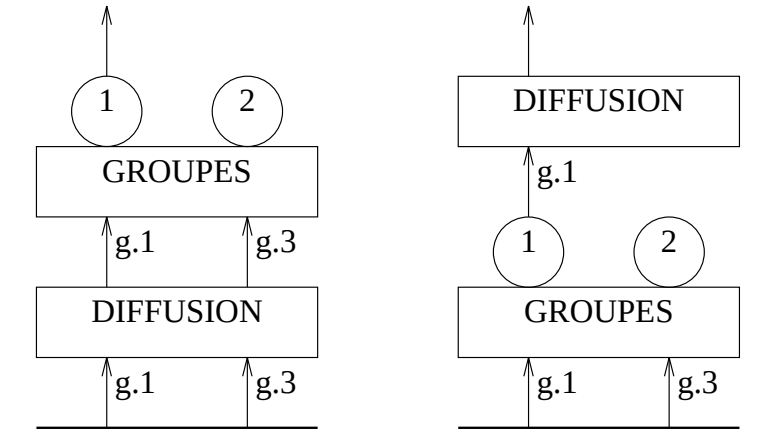

Fig. 2.4 : Structuration des fonctions

Pour des raisons de modularité, nous voulions séparer le protocole et la gestion des groupes. Ces fonctions peuvent être structurées de deux manières présentées dans la Fig. 2.4. Dans le premier cas, tous les groupes utilisent le même séquenceur dans la couche du protocole. Donc, le site qui possède le séquenceur doit supporter la charge générée par tous les autres sites. Dans le deuxième cas qui répartit la charge sur les sites, il y a un séquenceur par groupe. En outre, on évite de traiter certains messages − si le site n'est pas dans un groupe, la couche de groupes peut le rejeter sans aucun traitement par la couche du protocole.

Le protocole proposé est largement inspiré du protocole de Kaashoek et Tanenbaum. Ce dernier a été implémenté dans Amoeba et ses performances se sont avérées bonnes. Le protocole est basé sur l'estampillage de messages par un site séquenceur qui est statique. Ce site peut devenir surchargé et ralentir le système parce que tous les messages passent par ce site. Notre idée a été de répartir cette charge de travail du séquenceur. D'une part, par le placement de la couche de groupe, on obtient la possibilité d'avoir un séquenceur par groupe. D'autre part, on fait circuler le séquenceur dynamiquement et de manière paresseuse au gré du trafic − le site qui envoie des messages devient séquenceur. Ceci peut réduire le nombre de messages à transférer, parce que l'on peut joindre des messages à diffuser aux messages de validation. Dans le cas le plus favorable, on peut obtenir une diffusion physique par diffusion fiable et ordonnée. Les Fig. 2.5 et Fig. 2.6 illustrent le fonctionnement du protocole.

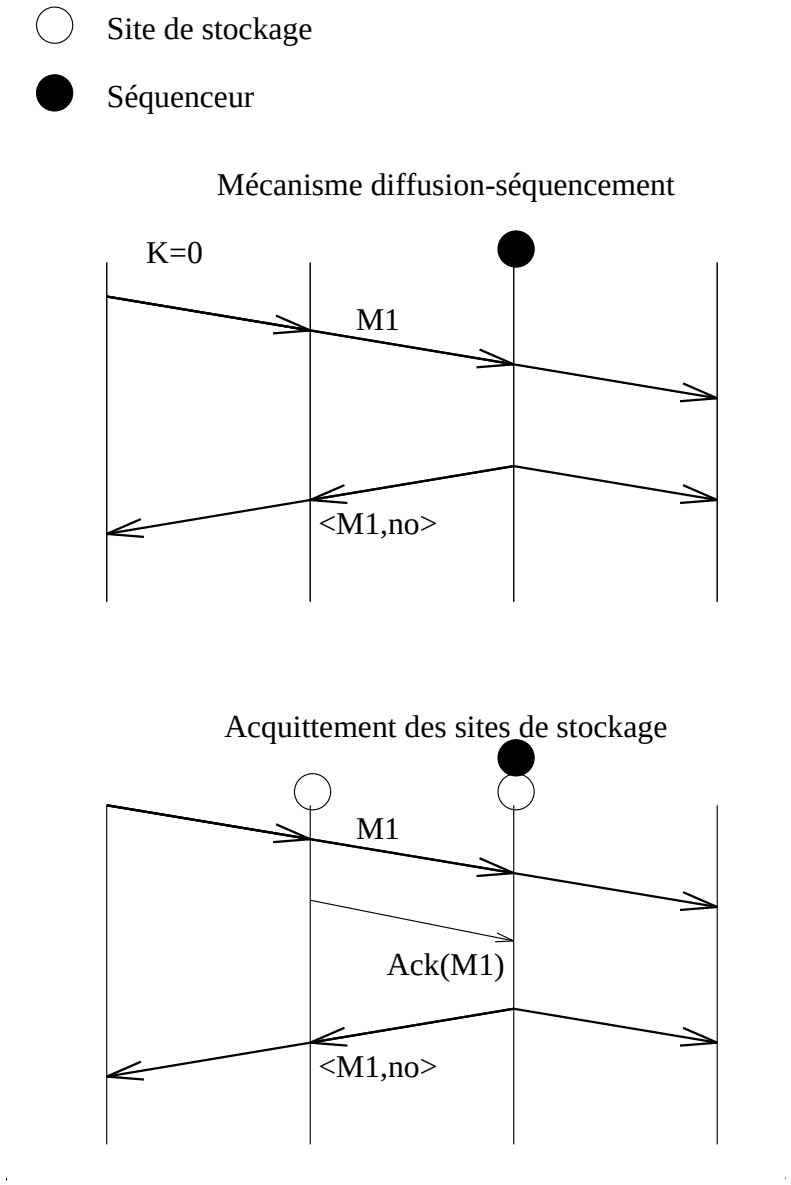

Fig. 2.5 : Principes du protocole

Le protocole fonctionne de façon suivante. À chaque instant, il existe un site séquenceur qui estampille des messages. Le droit d'être séquenceur est lié à un jeton qui peut être passé un autre site. Si le protocole doit résister à *K* pannes, alors *K* sites et le site séquenceur stockent tous les messages non encore acquittés dans un tampon. Si le jeton est passé à un autre site, ce site commence à stocker des messages. Supposons que le jeton est sur le site qui veut envoyer un message. Le site envoie le message par diffusion physique et attend *K* acquittements des sites de stockage.

Après les avoir reçus, il attribue un numéro de séquence et l'envoie par une autre diffusion physique. Si le site a un autre message à envoyer, celui−ci peut être joint au message portant le numéro de séquence. Dans le cas le plus favorable où le site séquenceur a plusieurs messages à transmettre, le coût du protocole est une diffusion physique plus *K* acquittements. Si un autre site a un message a transmettre, il l'envoi en diffusion. Le site séquenceur attribue un numéro de séquence et l'envoie par une diffusion physique en incluant le jeton à destination du site originaire du message. Ce site devient séquenceur pour les messages suivants. De cette façon, le protocole s'adapte au trafic − le jeton va sur un site et y reste tant que le site a des messages à transmettre.

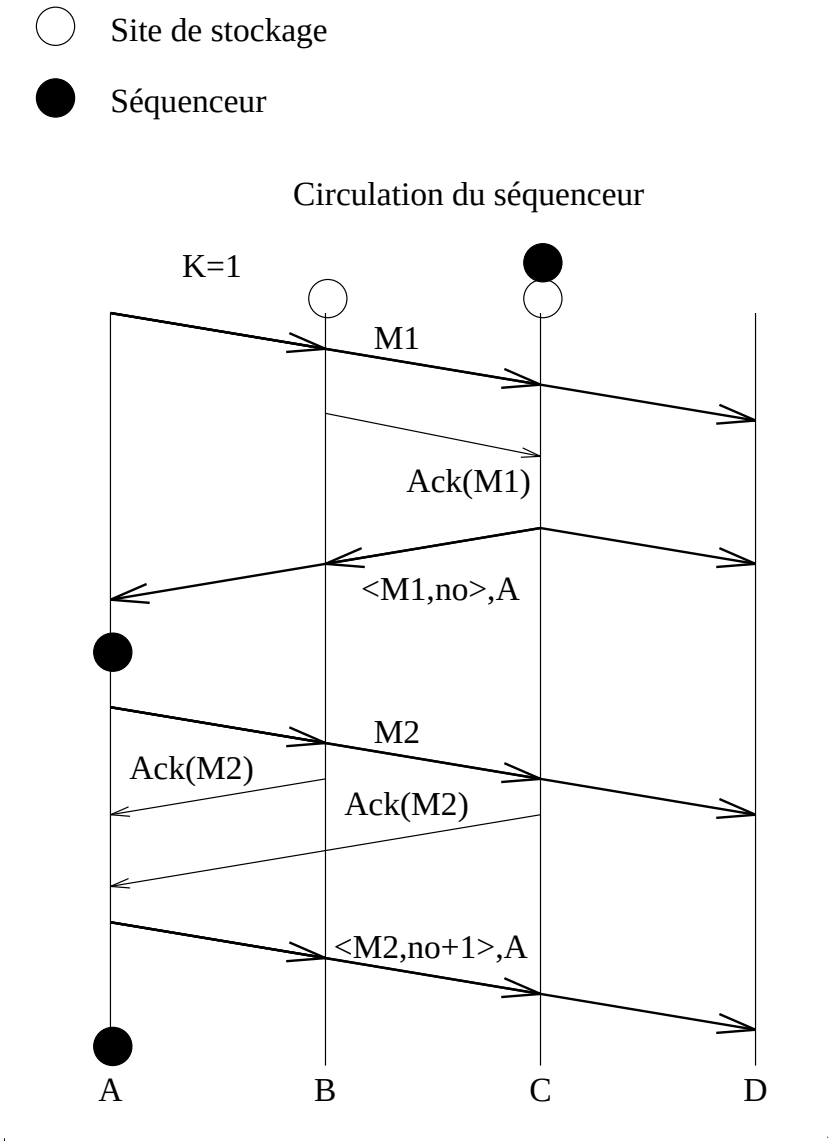

Fig. 2.6 : Circulation du séquenceur

Les pertes de messages et des pannes de sites sont traités comme dans le protocole de Kaashoek et Tanenbaum. Un site qui découvre qu'un message est perdu le demande auprès d'un site de stockage. Si un site séquenceur tombe en panne, une élection a lieu pour déterminer un nouveau site séquenceur.

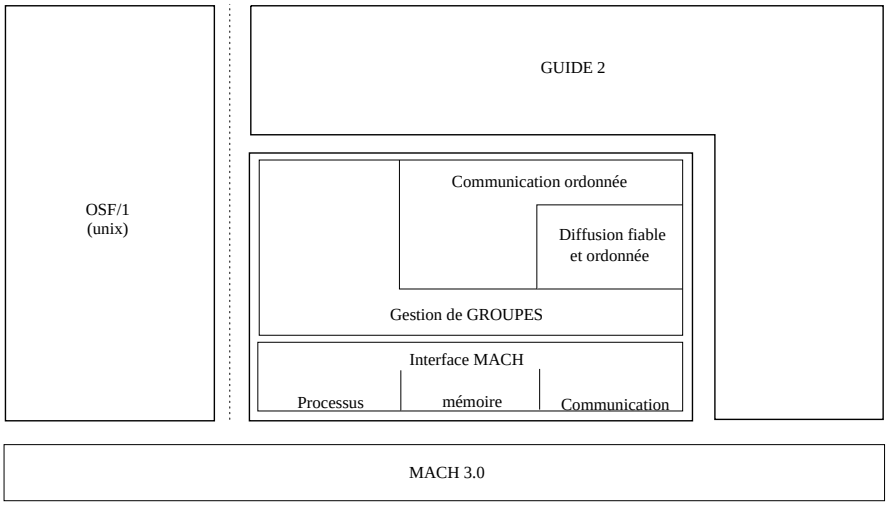

Fig. 2.7 : Architecture de l'implémentation

Le protocole a été réalisé comme une tâche MACH au dessus du contrôleur Ethernet (néanmoins, la circulation du séquenceur n'a pas été mise en œuvre). La tâche exploite le *filtre de paquets* de l'interface d'entrée/sortie de MACH [Mogul 87]. Cette fonctionnalité permet de programmer la réception et l'envoi de paquets Ethernet par une tâche utilisateur. Le module d'interface avec MACH fournit des primitives pour le contrôle des flots d'exécution, pour la gestion de tampons et pour la fragmentation et l'assemblage de messages en paquets Ethernet. L'architecture de l'implémentation est présentée dans la Fig. 2.7.

Duplication d'objets

La communication de groupes peut être exploitée dans un système réparti de différentes manières, par exemple, au niveau du système ou au niveau des applications. Au niveau du système, elle peut être utilisée pour gérer la connexion/deconnexion des sites, la mémoire répartie partagé ou un serveur de stockage fiable. Au niveau des applications, nous avons étudié comment des applications réparties peuvent tirer parti d'un tel type de communication. Le modèle objet de Guide cache les communications explicites sous l'invocation d'objet. Pour bien intégrer la communication de groupes au modèle Guide, nous avons décidé d'étendre la notion d'invocation d'une méthode sur un objet à la notion d'invocation d'une méthode sur un groupe d'objets (*multi−invocation*).

Un groupe est désigné par une référence système et traité comme une entité autonome composée d'objets qui ont une relation entre eux. Par exemple un groupe peut rallier des objets qui sont des copies d'un même objet (*objets dupliqués*), qui ont les même droits d'accès (un objet *extent*), qui sont conforment à un type, qui forment un objet composé ou qui participent à la même application [Shimizu 88]. Dans un premier temps, nous avons décidé d'expérimenter avec la duplication d'objets qui permet d'améliorer la fiabilité et la disponibilité, les qualités qui manquaient dans la première implémentation de Guide.

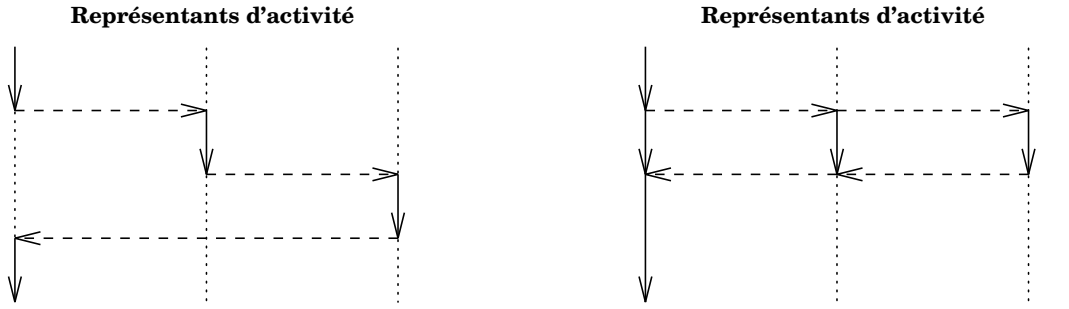

Fig. 2.8 : Activité − groupe des représentants

Pour cela, nous avons défini des *objets dupliqués* sur lesquels une invocation réussit même en présence de pannes de sites [Duda 92b][Duda 93a]. La duplication est le moyen de parvenir à cet objectif. Il y a différentes manières de réaliser la duplication d'objet et dans le Chapitre I nous avons présenté une étude dont les résultats montrent qu'il peut être avantageux de maintenir des copies cohérentes avec des mises−à−jour parallèles en utilisant un protocole de diffusion fiable et ordonnée [Duda 93c]. Un objet dupliqué présente la même interface qu'un objet unique. Il est composé de copies sur tous les sites actifs du système sur lesquelles sont appliquées des invocations de méthodes qui modifient l'état de l'objet. Le principe de la duplication est basé sur deux hypothèses : au départ toutes les copies sont dans le même état et on applique les mêmes opérations dans le même ordre sur toutes les copies. Pour assurer que l'on arrive au même état final, il faut assurer que les méthodes soient déterministes (pour cela, on exclut l'usage d'entrées/sorties dans le corps des méthodes) et qu'elles soient exécutées en exclusion mutuelle. Ce schéma assure la *cohérence forte* entre les copies et la propriété de *sérialisation sur une copie* [Bernstein 87].

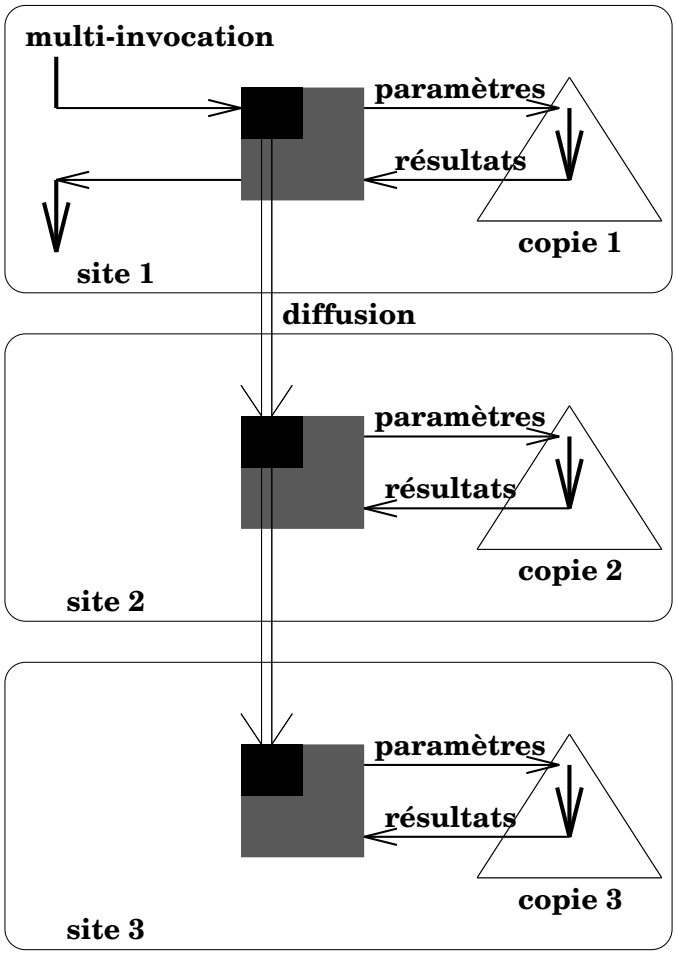

Fig. 2.9 : Principe de la multi−invocation

La réalisation de la duplication d'objets repose sur le protocole de diffusion fiable et ordonnée qui est en charge d'envoyer les demandes d'invocation qui modifient l'état de l'objet. L'ordre total assuré par le protocole garantit que les modifications sont apportées à toutes les copies dans le même ordre.

Le support d'exécution pour les multi−invocations peut être basé sur les mêmes principes que ceux déjà décrits dans II.4.1, à savoir, une activité est considérée comme un groupe de représentants locaux qui exécutent des invocations envoyées en diffusion. Alors que pour l'invocation d'objet unique, les représentants restent bloqués en attendant les résultats (Fig. 2.8), pour la multi−invocation, ils se chargent d'exécuter la méthode en parallèle. Dans ce modèle, l'extension d'activité s'apparente à l'entrée au groupe correspondant à l'activité et la fin d'activité correspond à la sortie du groupe. On peut remarquer que les groupes opaques des sites actifs se prêtent bien à ce modèle.

Le principe de la multi−invocation d'un objet dupliqué est présenté dans la Fig. 2.9. Pour les méthodes qui modifient l'état de l'objet, la référence système et les paramètres de l'invocation sont diffusés à tous les représentants de l'activité appelante. Chaque représentant invoque la méthode localement. Comme sur le site appelant les résultats sont disponibles, ils n'ont pas besoin d'être diffusés. Les invocations qui ne modifient pas l'état de l'objet ne sont exécutées que localement. Les invocations imbriquées (invocation d'un objet dupliqué à partir d'un objet dupliqué) sont aussi possibles. Dans ce cas, l'activité appelante diffuse l'invocation (qui porte un identificateur unique) et tous les autres sites qui ont reçu cette invocation ne la rediffusent pas. L'invocation est effectué sur chaque site et le résultat est utilisé localement pour le retour de l'invocation. Si une invocation d'un objet unique a lieu depuis un objet dupliqué, cette invocation n'est effectuée que sur un site (par exemple le site de l'activité principale) et le résultat est diffusé. La création d'objet est effectué de la même façon pour garantir l'unicité de la référence système attribué à l'objet.

Le modèle que nous proposons se rapproche du modèle d'Orca [Bal 90],[Bal 92]. Dans Orca les objets sont dupliqués et les opérations qui les modifient sont diffusées vers toutes les copies. Orca a toutefois été conçu pour exploiter le parallélisme et non pour améliorer la fiabilité et la disponibilité. En outre, les objets d'Orca ne peuvent pas être imbriqués − ils ne peuvent pas appeler d'autres objets dupliqués. Notre modèle se rapproche également de celui de Jalote [Jalote 89], qui a décrit une implémentation des objets dupliqués pour des réseaux de diffusion fiables. D'autres modèles similaires sont [Banâtre 85],[Cooper 85],[Martin 87] et [Pardyak 94].

L'avantage de notre modèle est sa simplicité et l'utilisation d'un nombre minimal de diffusions. La programmation des applications ne change pas par rapport aux objets simple. La seule différence est le mot clé `REPLICATED` qui doit être ajouté à la définition d'une classe d'objets dupliqués :

```
REPLICATED  CLASS  ClasseDuplique  IMPLEMENTS  unType
...

o  :  REF  unType;
...
//  création  d'un  objet  dupliqué
o  :=  ClasseDuplique.New;
...
//  multi−invocation
o.methode  (parametres);
...
```

II.4.3 Administration des sites

Un système réparti composé d'un grand nombre de composants est difficile à gérer et à administrer. Idéalement, un administrateur devrait pouvoir connaître à tout instant l'état et la configuration du système et pouvoir la modifier ou l'étendre au

cours de son fonctionnement. Un modèle de gestion dynamique de configuration a été proposé par Kramer et Magee [Kramer 85],[Kramer 90]. Il est basé sur la *spécification de la configuration* exprimée dans un langage de spécification qui décrit le comportement et la structure des composants d'un système réparti. Des modifications sont apportées à la configuration à l'aide d'altérations de la spécification. Ces altérations sont ensuite validées pour assurer que les modifications sont compatible avec le système existant. Le résultat des altérations est une nouvelle spécification. Un gestionnaire de configuration génère les primitives systèmes appropriées (des *actions*) pour créer la configuration décrite par la nouvelle spécification. Le principe de ce modèle est illustré dans la Fig. 2.10.

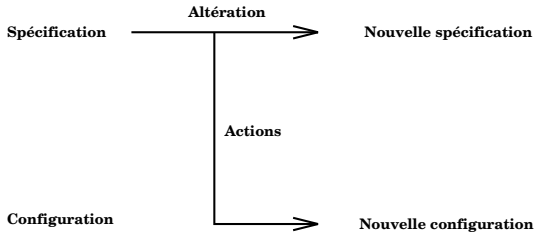

Fig. 2.10 : Gestion dynamique de la configuration

Deux éléments sont nécessaires pour utiliser ce modèle : il faut un langage pour la spécification et des primitives systèmes qui modifient la configuration. Nous avons utilisé ce modèle dans Guide en l'adaptant pour explorer l'approche objet à la spécification de la configuration. Le système Guide est composé de plusieurs entités comme les sites, volumes, usagers et groupes d'usagers. L'approche objet consiste à représenter chaque entité comme un objet appelé un *objet administré*. Un objet administré encapsule toute information nécessaire pour gérer l'entité et fournit des méthodes pour accéder et modifier cette information de manière contrôlée. Une collection d'objets administrés qui décrit un système peut être vu comme une spécification de la configuration. Le langage Guide offre les qualités comme la persistance, la répartition invisible et le nommage qui aident à la programmation des objets administrés. Néanmoins, les objets administrés doivent être fiables et disponibles parce que le bon fonctionnement du système en dépend.

Notre approche à l'administration est basé sur la notion de *cellule*, d'objets administrés et de primitives d'administration [Oueichek 92],[Duda 93b]. La portée de l'administration dans Guide est la cellule − un ensemble de sites qui coopèrent. Toutes les entités d'une cellule sont définies comme des objets administrés qui sont visibles et modifiables dans la cellule. Typiquement, une cellule est composée de

plusieurs dizaines de machines qui sont interconnectées par un réseau local. Ce concept permet d'exploiter la localité de communication − d'habitude un nombre limité de machines coopèrent de manière intensive, tandis que d'autres machines sont moins accédées. En outre, localiser la gestion sur un réseau local (sans ponts ni passerelles) permet d'adopter des hypothèses de fonctionnement (comme par exemple l'absence de partitionnement) qui facilitent la réalisation [Renesse 88].

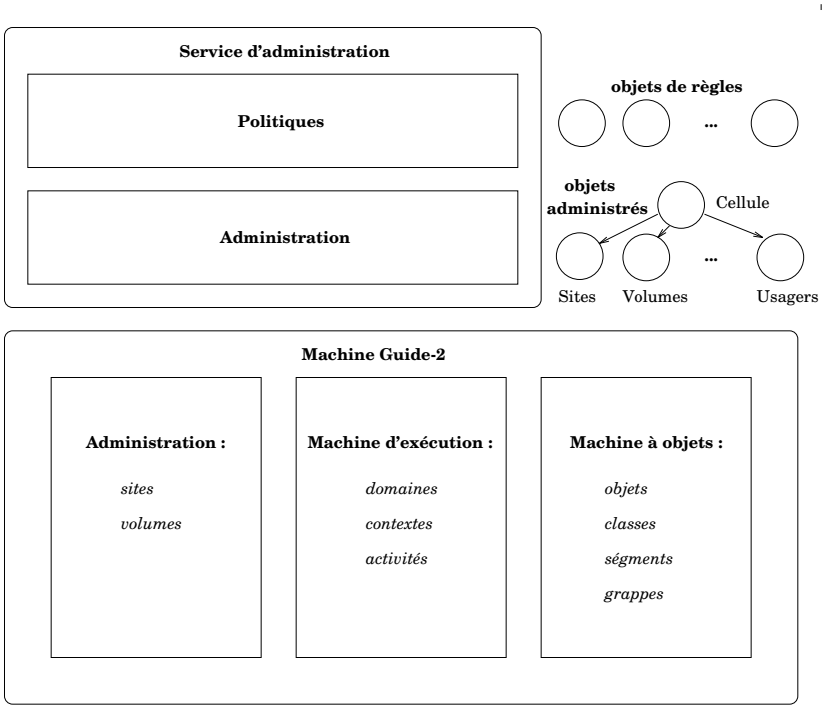

Fig. 2.11 : Objets administrés et la configuration dynamique.

Le service d'administration est structuré en deux niveaux (cf. Fig. 2.11). Le premier contient une base d'information représentée par des objets administrés persistants et fournit des commandes pour définir et modifier la spécification, ainsi que pour effectuer des actions qui altèrent la configuration selon les modifications. Les objets administrés seront gérés comme des *objets dupliqués* décrits précédemment pour garantir la fiabilité et la disponibilité satisfaisante. A ce niveau, nous définissons aussi des *objets de règles* qui expriment les stratégies qui sont appliquées automatiquement pour que le système s'adapte aux conditions changeantes. Par exemple, si un volume est trop souvent accédé à distance, le montage local permet d'améliorer les performances. Les stratégies doivent être définies dans un langage qui permet d'exprimer les règles du style : **`quand`** `condition` **`effectue`** `action`. Le support de synchronisation du langage Guide permet d'exprimer de telles règles.

Le niveau bas réalisé par la machine d'exécution offre des primitives qui effectuent les modifications de la configuration, par exemple les montages de volumes,

connexion/deconnexion de sites. Dans la suite, nous allons décrire la gestion des sites, un aspect important de l'administration d'une cellule.

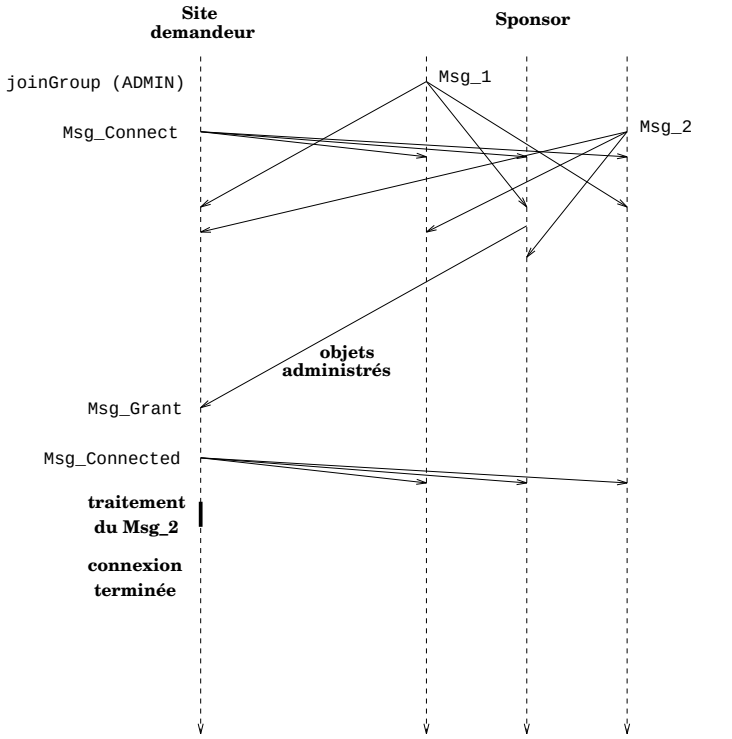

Fig. 2.12 : Protocole de connexion

Les primitives de gestion des sites permettent de démarrer un site, de le connecter à la cellule, de le déconnecter et de l'arrêter. Pour gérer le groupe de sites actifs dans une cellule, nous utilisons le protocole de diffusion fiable et ordonnée décrit précédemment. Le protocole de connexion s'inspire du protocole de Moser, Melliar−Smith et Agrawala [Moser 91]. Il est illustré dans la Fig. 2.12 et s'effectue de manière suivante.

Il existe un site nommé *sponsor* qui gère les demandes de connexion. Le site qui demande la connexion joint le groupe `ADMIN` du protocole de diffusion et commence à recevoir les diffusions d'administration (connexions, montages) destinées à ce groupe. En particulier, toutes les invocations des objets administrés passent par ce groupe. Le site émet le message `Msg_Connect` pour demander la connexion et il stocke tous les messages qu'il reçoit dans un tampon. Le message `Msg_Connect` est

stocké par tous les sites actifs pour pouvoir récupérer les pannes du site sponsor. Si le site demandeur est autorisé à joindre la cellule, le site sponsor lui adresse un message point−à−point `Msg_Grant` qui contient une copie à jour des objets administrés et le numéro de la dernière diffusion reçue. Après avoir reçu ce message, le site demandeur diffuse le message `Msg_Connected` pour confirmer sa connexion. Ensuite, pour arriver au même état des objets administrés que les autres sites, il traite tous les messages reçus précédemment qui ont le numéro de séquence plus grand que le numéro reçu dans le message `Msg_Grant` (ces messages ont pu modifier les objets entre temps). A ce moment, les objets administrés sont les mêmes que sur les autres sites actifs.

Les pannes de sites sont détectées par le protocole de diffusion et dans ce cas un message `Msg_Fail` avertit tous les sites du changement de la configuration. Un site est considéré comme en panne quand il ne répond pas plusieurs fois à un message. Cependant, il est impossible dans un système réparti asynchrone de distinguer entre un site qui est lent et un site qui ne fonctionne pas. Pour décider qu'un site est en panne, une temporisation est utilisée, ce qui dans certains cas peut entraîner une annonce de panne d'un site qui fonctionne. Dans ce cas, le site doit passer par la procédure de connexion pour devenir de nouveau un site actif . Si le site sponsor tombe en panne, un autre site est élu et commence à servir des demandes de connexion.

La validité du protocole de connexion est fondée sur les faits suivants :

1. les sites membres du groupe sont au départ d'accord sur l'appartenance au groupe,
2. si un site tombe en panne, il est retiré de la configuration après la transmission d'un nombre fini de messages,
3. un site qui demande une connexion sera admis au groupe après la transmission d'un nombre fini de messages.

Les propriétés d'ordonnancement du protocole de diffusion assurent que ces conditions sont vérifiées.

II.5 Conclusions

Le travail sur la conception et la réalisation des systèmes répartis nous a permi de confronter les idées issues de l'évaluation des performances avec la réalité. Une leçon importante est l'influence de la complexité − les systèmes répartis sont beaucoup plus complexes que ce que la modélisation permet de prendre en compte. Aussi, pour un concepteur de systèmes, les performances sont un facteur parmi d'autres aussi importants comme par exemple l'adéquation aux besoins des utilisateurs ou la facilité de réalisation. Une importance accrue est donnée par les concepteurs de systèmes répartis aux abstractions qui sont visibles et qui doivent au mieux faciliter le développement des applications réparties. De ce point de vue, les primitives de

communication d'Epsilon, et surtout le mécanisme de déroutement logiciel, constituait un progrès par rapport à la communication par messages offerte par la plupart des systèmes d'exploitation.

Néanmoins, la programmation en terme de processus communicant par messages reste complexe et le langage Guide qui s'appuie sur un modèle à objets et sur un modèle d'exécution spécifiques apporte un degré de facilité en plus. Programmer une application répartie comme une application centralisée est vraiment un progrès par rapport à des solutions précédentes.

Le modèle Guide a beaucoup de mérites. C'est un des rares systèmes qui fournit les fonctionnalités essentielles d'un système réparti à objets de manière intégrée au travers d'un langage : gestion des objets persistants, support d'exécution répartie intégré, répartition invisible, parallélisme apparent, liaison dynamique, expression de la synchronisation, exceptions, transactions. Cette richesse de fonctions serait difficilement utilisable dans un langage existant. Le langage Guide reflète certaines fonctions système, et en cache d'autres pour que l'écriture des applications répartie soit rigoureuse et plus facile. Vu de la perspective du temps, le modèle de Guide a été très évolué à l'époque et reste encore très intéressant aujourd'hui. Il y a peu de systèmes qui offrent des fonctionnalités aussi avancées. On peut par exemple citer Spring et Thor qui sont les plus proches sans toutefois atteindre Guide : Spring est un noyau qui présente un langage de spécification d'interfaces [Hamilton 93]; Thor est plutôt une base de données à objets structurée comme une architecture client/serveur [Liskov 92].

Néanmoins, on peut trouver dans le modèle de Guide quelques défauts. Par exemple, le modèle s'intègre mal avec le modèle des applications interactives et graphiques *X Window System*. Une application X communique avec le serveur X par messages et le serveur lui signale des événements importants qui doivent être traités par le mécanisme de *callback* − une forme de déroutement logiciel où l'application enregistre auprès du serveur des adresses des fonctions qui doivent être appelées. Dans le modèle Guide, il n'y a pas de mécanisme équivalent qui permettrait d'intégrer ce modèle d'exécution. Même si quelques applications comme Grif ont été encapsulées comme des objets Guide et intégrées avec le prototype Guide−1 avec succès [Decouchant 92], le partage de tels objets a posé des problèmes à cause des *callbacks* définis en termes d'adresses dans l'espace d'un processus. Les applications interactives et graphiques auraient été mieux supportées si un langage de commandes (*script*) comme *TCL* [Ousterhout 91] avait completé le langage Guide.

Le travail collaboratif et de manière générale les applications coopératives mettent en évidence d'autres manques dans Guide. En particulier, le support d'événements et un mécanisme d'exécution asynchrone seraient requis. Cet aspect constitue actuellement un thème de recherche au sein du projet Sirac qui est la suite de Guide. Un autre coté faible du modèle est relatif à l'expression de la répartition qui est moins évoluée que dans Emerald. On peut aussi reprocher au modèle Guide le

fait que tous les objets sont persistants. Un modèle où l'on peut exprimer que certains objets sont temporaires et non persistants serait plus adéquat.

Un autre problème est le manque de vulgarisation et de popularité des concepts de Guide − il n'est pas assez connu, ou reconnu, relativement à ses mérites. Quand nous avons fait un séminaire à *SUN Laboratories* (où le développement de Spring a lieu), les chercheurs ont été surpris par la qualité du modèle, des choix d'implémentation et des résultats de Guide. Cela vient probablement d'un manque de visibilité au niveau international en raison de l'accent qui a été mis sur la réimplémentation des mécanismes de base dans Guide−2 au détriment des aspects expérimentaux et d'exploitation plus large de Guide−1.

Vu de la perspective du temps, le projet n'a pas pris en compte l'apparition des technologies nouvelles telles que les réseaux à haut débits (FDDI − 100 Mbits/s, ATM − Gbits/s [Gigabit 92]) qui permettent d'atteindre des performances d'opérations à distance comparables à celles des opérations locales. Ces technologies changent les données du problème − d'autres mécanismes de base et autres stratégies sont nécessaires pour les exploiter.

Les prototypes de Guide ont implémenté la plupart des fonctionnalités du modèle, mais en mettant l'accent sur des aspects différents. Guide−1 était le premier prototype qui devait montrer la faisabilité du modèle et ouvrir le champs d'expérimentations diverses. L'effort s'est concentré sur le support d'exécution fortement couplé avec le langage, le support de la persistance, de la répartition, du parallélisme et de la synchronisation. Guide−2 reprenait certains mécanismes pour offrir un support générique des langages à objets, un support de protection et un support de stockage d'objets à base de systèmes de fichiers.

Guide−1 était un prototype complet sur Unix ce qui facilitait son portage. Il a été développé relativement vite et l'utilisation d'Unix avait l'avantage de faciliter son développement et la mise au point. Néanmoins, plusieurs mécanismes de base d'Unix n'ont pas été adéquats. Par exemple, les sémaphores sont trop coûteux [Decouchant 89c], les processus d'Unix sont trop lourds et le nombre de descripteurs de fichiers ouverts limité [Balter 91]. Ces limitations d'Unix ne permettaient pas d'implémenter correctement les domaines et les activités, par exemple l'instruction `CO_BEGIN` se traduisait en un *fork* d'Unix, une opération très coûteuse qui peut durer jusqu'à 2 secondes. Un autre choix possible aurait été d'utiliser une librairie de processus légers, au prix d'une mise au point extrêmement difficile. À cause des difficultés rencontrées dans l'utilisation de la mémoire partagée d'Unix, la mémoire d'objets sur un site a été partagée par tous les domaines présents sur ce site. Donc, une erreur dans une application pouvait perturber l'exécution des autres. D'autres faiblesses de Guide−1 sont l'effet du choix délibéré de laisser (dans un premier temps) de coté certains aspects comme la protection, la répartition de charge, l'expression de la répartition et du parallélisme. Une autre grande faiblesse a été l'absence d'outils et d'environnement de programmation qui auraient permi d'exploiter tout son

potentiel. Ceci a été amélioré quelques années plus tard par le développement d'un metteur au point [Jamrozik 91], d'un outil d'observation [Roisin 91] et d'un outil de visualisation [Vion−Dury 93]. Même avec ses défauts, Guide−1 a servi comme une plate−forme pour plusieurs applications comme Griffon, la version coopérative de Grif, *X500*, une implémentation d'un service de nommage, et *CIDRE*, une application de circulation de documents.

On peut regretter que Guide−1 a été abandonné à peine arrivé à son stade de fonctionnement. Il n'a pas été suffisamment mesuré et évalué pour comprendre son comportement et pour en tirer des leçons. Ce prototype était un investissement qu'il fallait exploiter comme un support d'investigation de différents problèmes dans la construction des systèmes répartis à objets. Il aurait été intéressant d'étudier différentes implémentations des mécanismes de base comme la mémoire d'objets, la gestion de domaines et d'activités et la répartition. En outre, on aurait pu considérer des stratégies possibles pour l'utilisation de ces mécanismes. On entend souvent l'opinion disant qu'il faut utiliser son propre prototype pour pouvoir vraiment l'évaluer, y apporter des corrections et le valider. Nous n'avons pas eu le temps nécessaire pour le faire, même pour le développement des outils, des service et de l'environnement de développement.

Guide−2 a réimplémenté la plupart des mécanismes de base sur une autre plate−forme, Mach 3.0. Mach présente des mécanismes intéressants qui correspondent bien au modèle de Guide, comme par exemple les *tasks* et les *threads*, et qui facilitent la mise en œuvre du modèle, comme les ports de communication et la gestion externe de pagination (*mappers*). Mach 3.0 avait un grand défaut, à savoir les performances des communications. Les communications étant gérées par un serveur spécial (*netmsgserver*) et les protocoles étant placés dans le serveur Unix, un message faisait plusieurs tours entre le noyau et les serveurs avant d'être émis sur un réseau local. Ce défaut a disparu quand, en abandonnant l'un des principes des micro−noyaux, la gestion des communications et le protocole ont été placés dans le noyau (version *NORMA*). Les performances de cette version sont comparables avec Unix. Mach a aussi posé d'autres problèmes détaillés dans [Chevalier 93a],[Chevalier 93b]. Au vu de l'expérience, le choix de Mach peut paraître mauvais − le prototype *OODE*, la suite de Guide−2, est réalisée actuellement sur Unix, Mach étant abandonné.

Guide−2 a permit d'améliorer les performances de l'invocation d'objet local par l'utilisation d'un mécanisme "*à la Multics*" [Daley 68] − au premier appel, le chaînage d'indirections vers le code d'une classe est mis sur place, et aux appels suivants, il permet d'invoquer une méthode avec des performances comparables à celles d'un appel de procédure. La protection a été réalisée dans Guide−2 au niveau des contextes qui comprennent des objets ayant les mêmes droits. Guide−2 offre un support générique aux langages à objets, qui se limite pour l'instant au langage Guide et à une extension de C++ persistant. Un protocole de diffusion fiable a été

spécifié et réalisé au−dessus de Mach. Il est pour l'instant utilisé pour l'administration des sites.

II.6 Bibliographie

[Accetta 86] M.J. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian et M. Young, ''Mach: A new kernel foundation for Unix development'', *Proc. of the USENIX 1986 Summer Conference*, pp. 93−112, été 1986.

[Ahamad 90] M. Ahamad, P. Dasgupta, R.J. LeBlanc Jr., ''Fault−tolerant atomic computations in an object−based distributed system'', *Distributed Computing*, 4(1990), pp. 69−80.

[Almes 85] G.T. Almes, A.P. Black, E.D. Lazowska et J.D. Noe, ''The Eden System: A Technical Review'', *IEEE Trans. Software Engineering*, SE−11(1), pp. 43−59, janvier 1985.

[Artsy 87] Y. Artsy, H. Chang et R. Finkel, ''Interprocess Communication in Charlotte'', *IEEE Software*, janvier 1987.

[Bal 89] H. Bal, F. Kaashoek, A. Tanenbaum, *Replication Techniques for Speeding Up Parallel Applications on Distributed Systems*, (IR−202), Vrije Universiteit, décembre 1989.

[Bal 90] H. Bal, F. Kaashoek, A. Tanenbaum, ''A Distributed Implementation of the Shared Data−Object Model'', *1st Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pp. 1−19, octobre 1990.

[Bal 92] H. Bal, M.F. Kaashoek et A. Tanenbaum, ''Orca: A Language for Parallel Programming of Distributed Systems'', *IEEE Trans. on Software Engineering*, 18(3), pp. 190−205, mars 1992.

[Balter 88] R.Balter, D. Decouchant, A. Duda, S.Krakowiak, V.Normand, X. Rousset de Pina, ''Exécution répartie dans le système Guide'', *Note technique Guide*, juin 1988.

[Balter 90] R. Balter, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, M. Riveill , C. Roisin, X. Rousset de Pina, R. Scioville, G. Vandome, ''Experience with Object−Based Distributed Computation in the Guide Operating System'', *Proc. Workshop on Workstation Operating Systems*, pp. 16−19, Pacific Grove, octobre 1989.

[Balter 91] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, G. Vandôme, ''Architecture and Implementation of Guide, an Object−Oriented Distributed System'', *Computing Systems*, 4(1), pp. 31−67, 1991.

[Balter 94] R. Balter, S. Lacourte, M. Riveill, ''The Guide language'', *The Computer Journal*, The Computer Journal à paraître 1994.

[Banâtre 85] J−M. Banâtre et al., ''The Concept of Multi−function: a General Structuring Tool for Distributed Operating System'', *Proc. 5th Int. Conference on Distributed Computing Systems*, pp. 478−485, 1985.

[Bernabeu 88] J.M. Bernabeu−Auban, P.W. Hutto, M.Y.A. Khalidi, M. Ahamad, W.F. Appelbe, P. Dasgupta, R.J. LeBlanc et U. Ramachandran, *The Architecture of Ra: A Kernel for Clouds*, (GIT−ICS−88/25), School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, 1988.

[Bernard 87] G. Bernard, A. Duda, Y. Haddad, G. Harrus, ''Implementing Distributed Applications on a Heterogenous Local Area Network'', *Proc. 2th Int. Symposium on Computer and Information Sciences*, pp. 276−287, 1987.

[Bernard 89a] G. Bernard, A. Duda, Y. Haddad, G. Harrus, ''Primitives for Distributed Computing in a Heterogenous Local Area Network Environement'', *IEEE Transactions on Software Engineering*, 15(12), pp. 1567−1578, 1989.

[Bernard 89b] G. Bernard, *Performances et systèmes répartis : de la modélisation à la réalisation*, Thèse de Docteur ès Sciences, Université de Paris−Sud, Orsay, avril 1989.

[Bernstein 87] P. Bernstein et al., *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987.

[Birman 87] K. Birman, T. Joseph, ''Reliable Communication in the Presence of Failures'', *ACM Trans. on Computer Systems*, 5(1), pp. 47−76, février 1987.

[Birrell 84] A.D. Birrell, et B.J. Nelson, ''Implementing Remote Procedure Calls'', *ACM Trans. on Computer Systems*, 2(1), février 1984.

[Black 85] A.P. Black, ''Supporting distributed applications: experience with Eden'', *10th ACM Symposium on Operating System Principles, SIGOPS Operating Systems Review*, 19(5), pp. 181−193, Decembre 1985.

[Black 86] A.P. Black, N. Hutchinson, E. Jul et H. Levy, ''Object structure in the Emerald system'', *Proc. 1st ACM Conference on Object−Oriented Systems, Languages and Applications (OOPSLA)*, septembre 1986.

[Black 87] A.P. Black, N. Hutchinson, E. Jul, H. Levy et L. F. Cabrera, ''Distribution and Abstract Types in Emerald'', *IEEE Trans. Software Engineering*, SE−13(1), pp. 65−76, 1987.

[Boyer 91] F. Boyer, J. Cayuela, P−Y. Chevalier, A. Freyssinet et D. Hagimont, ‘‘Supporting an Object−Oriented Distributed System: Experience with Unix, Chorus and Mach’’, *Symposium on Experience with Distributed and Multiprocessor Systems*, Atlanta, mars 1991.

[Chang 84] J. Chang, N. Maxemchuk, ‘‘Reliable Broadcast Protocols’’, *ACM Trans. on Computer Systems*, 2(3), pp. 251−273, août 1984.

[Cheriton 86] D.R. Cheriton, ‘‘VMTP: A Transport Protocol for the Next Generation of Communication Systems’’, *Proc. SIGCOMM’86 Conference*, 1986.

[Chevalier 92a] P.Y. Chevalier, D. Hagimont, S. Krakowiak, X. Rousset de Pina, ‘‘The Case for Dynamic Binding’’, *Proc. Third Workshop on Workstation Operating Systems*, juin 92.

[Chevalier 92b] P−Y. Chevalier, D. Hagimont, S. Krakowiak et X. Rousset de Pina, ‘‘System Support for Shared Objects’’, *Proc. 5th European SIGOPS Workshop*, Le Mont Saint−Michel, septembre 1992.

[Chevalier 92c] P−Y. Chevalier, ‘‘A Replicated Object Server for Distributed Object−Oriented System’’, *Proc. 11th Symposium on Reliable Distributed Systems*, octobre 1992.

[Chevalier 93a] P. Y. Chevalier, A. Freyssinet, D. Hagimont, S. Lacourte, X. Rousset de Pina, ‘‘Is the Microkernel Technology Well Suited for the Support of Object−Oriented Operating Systems: the Guide Experience’’, *Proc. Microkernel and Other Kernel Architecture*, 1993.

[Chevalier 93b] P. Y. Chevalier, A. Freyssinet, D. Hagimont, S. Lacourte, X. Rousset de Pina, ‘‘Experience with Shared Object Support in the Guide System’’, *Symposium on Experiences on Distributed Systems and Multiprocessors*, 1993.

[Chevalier 94] P.Y. Chevalier, F. Saunier, M. Riveill, ‘‘Towards a Generic System Support for Co−operative Applications’’, *IEEE Third Workshop on Enabling Technologies − Infrastructure for Collaborative Enterprises*, pp. 1−10, avril 1994.

[Clark 85] D.D. Clark, ‘‘The Structuring of Systems Using Upcalls’’, *Proc. 10th Symposium on Operating Systems Principles*, 1985.

[Cooper 85] E. C. Cooper, ‘‘Replicated Distributed Programs’’, *Ph.D. Thesis, Report No. UCB/CSD85/231, University of California, Berkeley*, mai 1985.

[Daley 68] Robert C. Daley et Jack B. Dennis, ‘‘Virtual Memory, Processes, and Sharing in MULTICS’’, *CACM*, 11(4), pp. 308−318, mai 1968.

[Dasgupta 90] P. Dasgupta, R.C. Chen, S. Menon, M.P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R.J. LeBlanc, W.F. Appelbe, J.M. Bernabeu−Auban, P.W. Hutto, M.Y.A. Khalidi et C.J. Wilkenloh, ‘‘The Design and Implementation of the Clouds Distributed Operating System’’, *Computing Systems*, 3(1), pp. 11−45, hiver 1990.

[Dasgupta91] P. Dasgupta, R. Ananthanarayanan, S. Menon, A. Mohindra et M.P. Pearson, ‘‘Language and Operating System Support for Distributed Programming in Clouds’’, *Proc. the 2nd Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, mars 1991.

[Decouchant 88] D. Decouchant, A. Duda, A. Freyssinet, E. Paire, M. Riveill, X. Rousset de Pina, G. Vandôme, ‘‘Guide: An Implementation of the COMANDOS Object−Oriented Distributed System Architecture on UNIX’’, *Proc. EUUG ’88 Conference*, pp. 181−193, Cascais, octobre 1988.

[Decouchant 89a] D. Decouchant, A. Duda, A. Freyssinet, H. Nguyen Van, M. Riveill, X. Rousset de Pina, ‘‘GUIDE, un système réparti à objet’’, *Actes de la Conférence UNIX − AFUU*, pp. 297−316, Paris, mars 1989.

[Decouchant 89b] D. Decouchant, S. Krakowiak, M. Meysembourg, M. Riveill et X. Rousset de Pina, ‘‘A Synchronization Mechanism for Typed Objects in a Distributed System’’, *Sigplan Notices*, 24(4), pp. 105−107, avril 1989.

[Decouchant 89c] D. Decouchant, E. Paire et M. Riveill, ‘‘Efficient Implementation of Low−Level Synchronization Primitives in the Unix−Based GUIDE Kernel’’, *Proc. EUUG*, pp. 283−294, Vienne, septembre 1989.

[Decouchant 89d] D. Decouchant, E. Finn, C. Horn et M. Riveill, ‘‘Experience with Implementing and Using an Object−Oriented Distributed System’’, *Proc. Workshop on Experiences with Distributed and Multiprocessors Systems*, octobre 1989.

[Decouchant 89e] D. Decouchant, S. Krakowiak et M. Riveill, ‘‘Management of Concurrent Distributed Computation in the Guide Object−Oriented System’’, *Proc. Workshop on Large−Grained Garallelism*, Pittsburgh, octobre 1989.

[Decouchant 90] D. Decouchant, A. Duda, ‘‘Remote Execution and Communication in Guide − an Object−Oriented Distributed System’’, *Proc. 2nd IEEE Workshop on Experimental Distributed Systems*, pp. 49−53, octobre 1990.

[Decouchant 91] D. Decouchant, P. Ledot, M. Riveill, C. Roisin et X. Rousset de Pina, ‘‘A Synchronization Mechanism for Typed Objects in a Distributed System’’, *Proc. 11th Int. Conference on Distributed Computing Systems*, pp. 152−159, Arlington, mai 1991.

[Decouchant 92] D. Decouchant, V. Quint, I. Vatton, ‘‘L’édition coopérative de documents avec Griffon’’, *Actes du Colloque IHM’92*, decembre 1992.

[Duda 92a] A. Duda, *Communication in Guide−2*, note Bull−IMAG Systèmes, mai 1992.

[Duda 92b] A. Duda, D. Veillard, *Resilient Objects in the Guide Distributed System*, note Bull−IMAG Systèmes, September 1992.

[Duda 93a] A. Duda, D. Veillard, ''Communication Support for a Replicated Object Service'', *Fifth European Workshop on Dependable Computing*, Lisboa, mars 1993.

[Duda 93b] A. Duda, J. Cayuela, ''System Management in the Guide Distributed System'', *First International Workshop on Systems Management*, avril 1993.

[Duda 93c] A. Duda, ''Analysis of Multicast−based Object Replication Strategies in Distributed Systems'', *Proc. 13th Int. Conference on Distributed Computing Systems*, pp. 311−318, Pittsbourgh, mai 1993.

[Freyssinet 91a] A. Freyssinet, *Architecture et réalisation d'un système réparti à objets*, Thèse de doctorat, Université Joseph Fourier Grenoble, juillet 1991.

[Freyssinet 91b] A. Freyssinet, S. Krakowiak et S. Lacourte, ''A Generic Object−Oriented Virtual Machine'', *Proc. First International Workshop on Object−Orientation in Operating Systems*, Palo Alto, octobre 1991.

[Gigabit 92] Special Issue on Gigabit Networks, *IEEE Communications*, avril 1992.

[Hagimont 92] D. Hagimont, S. Krakowiak et X. Rousset de Pina, ''Protection in an Object−Oriented Distributed Virtual Machine'', *Proc. Second International Workshop on Object−Orientation in Operating Systems*, Paris, septembre 1992.

[Hagimont 93] D. Hagimont, *Adressage et protection dans un système réparti*, Thèse de doctorat, Université Joseph Fourier Grenoble, octobre 1993.

[Hagimont 94] D. Hagimont, ''Protection in the Guide Object Oriented Distributed System'', *Proc. ECOOP*, pp. 1−18, Bologna, juillet 1994.

[Hamilton 93] G. Hamilton et P. Kougiouris, ''The Spring Nucleus: A Microkernel for Objects'', *Proc. 1993 Summer USENIX Conference*, pp. 147−160, juin 1993.

[Howard 88] J.H. Howard et al., ''Scale and Performance in a Distributed File System'', *ACM Trans. on Computer Systems*, 6(1), pp. 51−81, février 1988.

[Jalote 89] P. Jalote, ''Resilient Objects in Broadcast Networks'', *IEEE Trans. Software Engineering*, 15(1), pp. 68−72, janvier 1989.

[Jamrozik 91] H. Jamrozik, M.Santana et C. Roisin, ''A Graphical Debugger for Object−Oriented Distributed Programs'', *Proc. TOOLS−USA*, Santa Barbara, juillet 1991.

[Jul 88] E. Jul, *Object Mobility in a Distributed Object−Oriented System*, (88−12−06), Phd. Dissertation, University of Washington, Seattle, décembre 1988.

[Kaashoek 91] F. Kaashoek, A. Tanenbaum, ''Group Communication in the Amoeba Distributed Operating System'', *Proc. 11th Int. Conference on Distributed Computing Systems* , pp. 222−230, mai 1991.

[Krakowiak 90] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin et X. Rousset de Pina, ''Design and Implementation of an Object−Oriented, Strongly Typed Language for Distributed Applications'', *Journal of Object Oriented Programming*, 3(3), pp. 11−22, septembre/octobre 1990.

[Kramer 85] J. Kramer, J. Magee, ''Dynamic Configuration for Distributed Systems'', *IEEE Trans. Software Engineering*, 11(4), pp. 424−436, avril 1985.

[Kramer 90] J. Kramer, J. Magee, ''The Evolving Philosophers Problem: Dynamic Change Management'', *IEEE Trans. Software Engineering*, 16(11), pp. 1293−1306, novembre 1990.

[Lacourte 91a] S. Lacourte, *Exception dans les langages à objets*, Thèse de doctorat, Université Joseph Fourier Grenoble, juillet 1991.

[Lacourte 91b] S. Lacourte, ''Exceptions in Guide, an Object−Oriented Language for Distributed Applications'', *Proc. ECOOP'91*, Genève, Juillet 1991.

[Leblanc 88] R.J. Leblanc, W.F. Appelbe, ''The Clouds Distributed Operating System'', *Proc. 8th Int. Conf. on Distributed Computing Systems*, pp. 2−9, San Jose, juin 1988.

[Levy 89] H.M. Levy et E.D. Tempero, ''On the Non−Duality of Modules and Objects for Distributed Programming'', *Technical Report 98−04−04, University of Washington*, avril 1989.

[Liskov 84] B. Liskov, *Overview of the Argus Language and System*, (Programming Methodology Group Memo 40), Laboratory of Computer Science, MIT, 1984.

[Liskov 85] B.H. Liskov, *The Argus language and system, Distributed systems: methods and tools for specification*, Lecture Notes in Computer Science, Vol. 190, Springer−Verlag, pp. 343−430, 1985.

[Liskov 92] B. Liskov, *Preliminary design of the Thor object−oriented database system*, (Programming Methodology Group Memo 74), Laboratory of Computer Science, MIT, 1992.

[Lunati 88] J−M. Lunati, *Migration de processus et partage de charge dans les systèmes répartis*, rapport de DEA, septembre 1988.

[Martin 87] Martin, Bergan, Russ, ‘‘PARPC: A System for Parallel Procedure Calls’’, *Proc. ICPP*, pp. 449−452, août 1987.

[Melliar−Smith 90] P.M. Melliar−Smith, L.E. Moser, V. Agrawala, ‘‘Broadcast Protocols for Distributed Systems’’, *IEEE Trans. Parallel and Distributed Systems*, 1(1), pp. 17−25, January 1990.

[Mogul 87] J. Mogul et al., ‘‘The Packet Filter : An Efficient Mecanism for User−level Network Code’’, *Proc. SIGCOMM 87*, 1987.

[Moser 91] L.E. Moser, P.M. Melliar−Smith, V. Agrawala, ‘‘Membership Aplgorithms for Asynchronous Distributed Systems’’, *Proc. 11th Int. Conference on Distributed Computing Systems*, pp. 480−488, Arlington, mai 1991.

[Mullender 86] S.J. Mullender et A.S. Tananbaum, ‘‘The Design of a Capability−Based Distributed Operating System’’, *Computer Journal*, 29(8), pp. 289−299, août 1986.

[Mullender 87] S.J. Mullender, éditeur, *The Amoeba Distributed Operating System: Selected papers 1984 − 1987*, CWI tract 41, 1987.

[Nelson 81] B.J. Nelson, *Remote Procedure Call*, PhD Dissertation, Carnegi−Mellon University, mai 1981.

[Oueichek 92] I. Oueichek, *Administration du système réparti Guide−2*, rapport DEA, juin 1992.

[Ousterhout 91] J.K. Ousterhout, ‘‘An X11 Toolkit Based on the Tcl Language’’, *Proc. USENIX Association 1991 Winter Conference*, pp. 105−−115, janvier 1991.

[Pardyak 94] P. Pardyak et B.N. Bershad, ‘‘A Group Structuring Mechanism for a Dsitributed Object−Oriented Language’’, *Proc. 14th Int. Conference on Distributed Computing Systems*, pp. 312−319, Poznan, juin 1994.

[Peterson 89] J. Peterson et al., ‘‘Preserving and Using Context Information in Interprocess Communication’’, *ACM Trans. Computer Systems*, 7(3), pp. 217−246, août 1989.

[Renesse 88] R. van Renesse, A. Tanenbaum, ‘‘Voting with Ghosts’’, *Proc. 8th Int. Conference on Distributed Computing Systems*, pp. 456−461, 1988.

[Ricart 81] G. Ricart et A.K. Agrawala, ‘‘An Optimal Algorithm for Mutual Exclusion in Computer Networks’’, *CACM*, 24(1), pp. 9−17, janvier 1981.

[Riveill 92] M. Riveill, ‘‘An Overview of the Guide Language’’, *Proc. Second Workshop on Objects in Large Distributed Applications*, Vancouver (Canada), octobre 1992.

[Riveill 93] M. Riveill, *Langages et systèmes pour applications réparties*, Habilitation à diriger des recherches, Institut National Polytechnique de Grenoble, janvier 1993.

[Roisin 91] C. Roisin, M. Santana, *The Observer: A Tool for Observing Object−Oriented Distributed Applications*, (91−08), Bull−Imag Systèmes, janvier 1991.

[Rousset 93] X. Rousset de Pina, *Conception et construction de noyaux de systèmes opératoires*, Habilitation à diriger des recherches, Institut National Polytechnique de Grenoble, janvier 1993.

[Rozier 88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, P. Léonard, S. Langlois et W. Neuhauser, ''Chorus Distributed Operating Systems'', *Computing Systems*, 1(4), pp. 305−370, fin 1988.

[Saltzer 81] J.H. Saltzer et al., ''End−to−End Arguments in System Design'', *Proc. 2nd Int. Conference on Distributed Computing Systems*, Paris, avril 1981.

[Shapiro 86] M. Shapiro, ''SOS: a Distributed Object−Oriented Operating System'', *Proc. 2nd European Workshop on Making Distributed Systems Work*, Amsterdam, septembre 1986.

[Shimizu 88] H. Shimizu et al., ''Hierarchical Object Groups in Distributed Operating Systems'', *Proc. 8th Int. Conference on Distributed Computing Systems*, pp. 18−24, juin 1988.

[Sun 86] Sun Microsystems, *External Data Represntation Protocol Specification*, janvier 1986.

[Sun 88a] Sun Microsystems, *Network Programming*, mai 1988.

[Sun 88b] Sun Microsystems, ''RPC: Remote Procedure Call Protocol Specification Version 2; RFC1058'', *Internet Request for Comments*, (1057), juin 1988.

[Sun 89] Sun Microsystems, ''NFS: Network File System Protocol Specification; RFC1094'', *Internet Request for Comments*, (1094), mars 1989.

[Tanenbaum 85] A. Tanenbaum et R. van Renesse, ''Distributed Operating Systems'', *ACM Computing Surveys*, 17(4), décembre 1985.

[Tanenbaum 89] A. Tanenbaum, *Computer Networks*, 2nd edition, Prentice−Hall, 1989.

[Veillard 92a] D. Veillard et F. Coutant, *Protocoles de diffusion fiable pour l'invocation multiple d'objets Guide*, projet de 3me année ENSIMAG, juin 1992.

[Veillard 92b] D. Veillard, *Appel d'objets à distance et protocoles de diffusion*, rapport DEA, septembre 1992.

[Vion−Dury 93] JY. Vion−Dury, *Etude et réalisation d'un metteur au point graphique pour applications réparties à objets*, Mémoire Ingénieur CNAM, décembre 1993.

[Wilbur 87] S. Wilbur et B. Bacarisse, ‘‘Building Distributed Systems with Remote Procedure Call’’, *Software Engineering Journal*, 2(5), pp. 148−159, septembre 1987.

Chapitre II

Conception et réalisation de systèmes répartis