DISIX : système de gestion de fichiers distribué. Implantation sur SPS7 et UNIX

André Baux

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

CENTRE AGREE

DE GRENOBLE (C.U.E.F.A.)

MEMOIRE

présenté en vue d'obtenir

le DIPLOME D'INGENIEUR C.N.A.M.

en

informatique

par

André Baux

DISIX

Système de gestion de fichiers distribué
Implantation sur SPS7 et UNIX

Soutenu le octobre 1985

JURY

Président : MM L. Bolliet et J.Y. Ranchin
Membres : M G. Vandome
M J. Coré
M M. Habert

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

CENTRE AGREE

DE GRENOBLE (C.U.E.F.A)

MEMOIRE

présenté en vue d'obtenir

le DIPLOME D'INGENIEUR C.N.A.M.

en

INFORMATIQUE

par

André BAUX

DISIX

Système de gestion de fichiers distribué.
Implantation sur SPS7 et UNIX.

Les travaux relatifs au présent mémoire ont été effectués au centre de recherche de Bull IMAG sous la direction de Monsieur Gérard Vandome et à Bull Sems en collaboration avec les services de Monsieur Michel Habert et Monsieur Jacques Febvre.

DISIX

DIStributed unIX

INTRODUCTION.

1 - ARCHITECTURE GENERALE.

- SCHEMA GENERAL.
- PROTOCOLES.
- MONTAGE.

2 - REALISATION.

- NOTION DE CLIENT / SERVEUR.
- ORGANISATION EN PROCESSUS.
- MONTAGE / DEMONTAGE.
- CLIENT / SERVEUR.
- DIFFERENTS TYPES D'EXECUTION.
- DROIT D'ACCES.
- ACCES.
- ILLUSTRATION.

3 - SYNTHESE DE DEVELOPPEMENT.

CONCLUSION.

Je tiens à remercier,

Monsieur J.Y. RANCHIN, Professeur du Conservatoire National des Arts et Métiers, et Monsieur L. BOLLIET, Professeur à l'Université des Sciences Sociales de Grenoble, qui m'ont fait l'honneur de présider le jury de soutenance de ce mémoire.

Monsieur Gérard VANDOME du centre de recherche de Bull Grenoble, de m'avoir proposé ce sujet particulièrement complet et intéressant et toute l'équipe des chercheurs de Bull et de l'IMAG pour leurs conseils et leurs encouragements.

Monsieur Michel HABERT de Bull Sems et son équipe réseau et télécommunications pour leurs explications patientes sur les principes et l'utilisation des différents services de télécommunications qu'ils ont développés.

Monsieur Jacques FEBVRE de Bull Sems et son équipe système pour les éclaircissements fournis sur des points particuliers du système UNIX.

Monsieur Jean CORE, chef du service Gestion et Informatique d'ATOCHEM, usine de Jarrie, de m'avoir fait l'honneur de participer à ce jury.

SOMMAIRE

Ce mémoire correspond à un projet de recherche et d'étude de faisabilité d'un système distribué de fichiers sous le système UNIX (1). Ce système distribué de fichiers est bati au-dessus d'un service de transport orienté connexion. Le mémoire comporte trois parties : l'étude des mécanismes de communication entre processus sur les différents systèmes UNIX, les choix d'architecture et d'implantation de la couche d'accès aux services de communication et le protocole de cette couche d'accès, enfin, un développement sur UNIX système V sur un SPS7 de Bull Sems.

---

(1) UNIX est une marque AT&T - Bell Laboratories

GLOSSAIRE

DISIX
"DIstributed unIX". Système de gestion de fichiers distribué sur UNIX pour machines et réseaux hétérogènes.

ETHERNET
Marque déposée RANK XEROX. Réseau local sur cable coaxial, avec méthode d'accès de type CSMA-CD et débit de 10 MB/s.

IPC
"InterProcess Communication" : mécanismes permettant la communication entre processus. L'IPC peut être uniquement local (interne) ou généralisé (interne et externe).

MAD
Méthode d'Accès Distribué développée chez Bull Sems pour permettre l'accès à différent services de communications sur la machine SPS7.

ME
Module d'échange du SPS7 (microprocesseur 8 ou 16 bits).

MMC
Module de mémoire commune du SPS7 accessible à tout MT, ME (1 ou 2 Mb).

MML
Module de mémoire locale du SPS7 accessible de façon privilégiée par MT sur bus local (1 ou 2 Mb).

MT
Module de traitement du SPS7 (microprocesseur 16 ou 32 bits).

RPC
"Remote Procedure Call". Mécanisme de communication entre client et serveur par appel procédural sur système distant.

SCB
"Session Control Block". Bloc paramètre standardisé utilisé pour l'accès à la session.

SESSION
Ensemble des actions et des ressources nécessaires pour l'établissement, la fermeture et l'échange d'information entre deux SSAP.

SMBUS
Bus de communication du SPS7 (microprocesseur 16 bits).

SPART
Système d'exploitation temps réel utilisé sur les ME (SPART-UE) et les MT (SPART-UT) du SPS7 où sont réparties les différentes couches des services de communication.

SPIX
SPs7 unIX. Version 2 du système V de AT&T.

SPS7
SM90 sous licence CNET.

SSAP
"Session Service Access Point". Identifiant de l'utilisateur (local ou distant).

TSAP
"Transport Service Access Point". Identifiant de l'occurence d'une connexion session dans un SPS7.

UCB
"User Control Block". Bloc paramètre standardisé référencé par les primitives de communication de la MAD.

0. INTRODUCTION 1

I. Présentation d'UNIX 3

I.1. Introduction 3

I.2. Les processus 3

I.3. Environnement courant du processus 8

I.4. IPC : la communication entre processus 9

I.5. Le système de fichiers 18

I.6. "Drivers" 24

I.7. Aspects particuliers d'UNIX 25

II. L'ENVIRONNEMENT DE COMMUNICATION DU SPS7 27

II.1. Le SPS7 27

II.2. Architecture pour Applications distribuées 29

II.3. La Méthode d'Accès Distribuée MAD 31

II.4. Conception d'une application distribuée 36

III. PRESENTATION DE DISIX 39

III.1. Introduction 39

III.2. Montage d'un système distant 39

III.3. Droit d'accès au système distant 42

III.4. Facilités offertes à l'utilisateur 42

III.5. Moyens à mettre en oeuvre 44

III.6. Compilation et édition de liens avec DISIX 45

IV. ETUDE DE L'ARCHITECTURE 47

IV.1. Choix d'une architecture 47

IV.2. Introduction à l'architecture de DISIX 48

IV.3. Problème posé par le nom 51

IV.4. Synoptique de l'architecture 52

IV.5. Exécution distante 56

IV.6. Description des processus 57

IV.7. Protocole 61

V. IMPLANTATION 65

V.1. Les choix d'implantation 65

V.2. Les commandes DISIX 74

V.3. Les processus DISIX 78

V.4. Les appels systèmes et les services 91

VI. INSTALLATION 105

VI.1. Hiérarchie du système de fichiers 105

VI.2. Implantation des bibliothèques 107

VI.3. Compilation ccd 109

VI.4. Commandes et programmes utilisateurs 109

VII. CONCLUSION 110

ANNEXES

Codes d'erreurs (errno)

Manuel de commandes (1D)

Manuel d'administration (1MD)

Manuel de fonctions (3CD)

BIBLIOGRAPHIE

0. INTRODUCTION.

Le système UNIX (UNIX est une marque AT&T - Bell Laboratories) s'est développé rapidement suite à la demande importante d'un système d'exploitation temps partagé, simple et efficace, surtout en milieu universitaire. Parallèlement, s'est accrue la nécessité importante de communication entre les utilisateurs de machines impliquant la coopération entre les systèmes utilisés, ou le partage de certaines ressources comme l'imprimante d'où est issu ce texte. Ce besoin s'est traduit par le développement des réseaux locaux.

La simultanéité de ces développements dans les milieux universitaires et chez les constructeurs Américains a abouti logiquement à la recherche de différentes solutions de systèmes distribués basés sur UNIX et Ethernet.

Les systèmes distribués peuvent être divisés en deux catégories principales : les systèmes qui apparaissent à l'utilisateur comme un seul système central, et ceux qui permettent la communication et la coopération entre des systèmes autonomes. La première catégorie, que l'on peut appeler "Systèmes Distribués Intégrés", utilise généralement des réseaux trés performants et fournit un accès transparent aux ressources distribuées (fichiers, périphériques ...). La deuxième catégorie, que l'on peut appeler "Systèmes Distribués Coopérants", peut utiliser des réseaux à longue distance et chaque système conserve son autonomie et gère ses propres ressources. Cette catégorie permet principalement l'échange d'information (courrier, fichiers ...) entre ses utilisateurs; la distribution peut être plus sophistiquée (base de données partageable, traitement réparti), mais cela est réalisé au niveau application alors que, dans la première catégorie, la distribution est transparente pour les applications.

Le projet DISIX (DIStributed unIX) se situe dans la catégorie des "Systèmes Distribués Intégrés" et pose le problème suivant : soit plusieurs types de machines dotées d'un système d'exploitation UNIX et connectées à un réseau local ou public. Tout utilisateur d'une machine UNIX se voit proposer l'accès et la manipulation, dans le cadre des protections mises en place, de l'ensemble des fichiers des systèmes de fichiers des machines UNIX; et cela, de façon transparente pour l'utilisateur.

Sachant qu'UNIX est basé essentiellement sur son système de fichiers, si on distribue celui-ci, on distribue 90% d'UNIX.

Nous nous proposons de réaliser la maquette d'un système dans la philosophie de la Newcastle Connection mais en utilisant un service de communication orienté connexion plutôt qu'en datagramme, et de réaliser des RPC (appels de procédures à distance) sur une connexion déjà établie; puis de faire une comparaison entre les deux modes de connexion.

Egalement, dans le cadre de la collaboration entre le centre de recherche de la Bull à l'IMAG et Bull Sems, le projet consiste à réaliser

l'application sur les machines SPS7 connectées à Ethernet, en utilisant les services de communication développés récemment par la Sems, notamment la Méthode d'accès Distribuée.

Plusieurs problèmes se posent dans ce type d'application. Entre autres, les choix d'implantation de la couche de communication externe et les répercussions sur la communication interne, le transfert de données entre des machines de types différents, l'exécution de fichiers à distance.

Ce rapport comporte dans une première partie une présentation du système UNIX pour les points qui nous intéressent essentiellement. Cette présentation porte l'accent sur le système V mais fait plusieurs parallèles avec les autres versions d'UNIX. Puis, nous décrirons les principes de l'accès aux services de communications dans l'environnement de la machine SPS7. Ensuite, le rapport situe le projet dans le cadre des systèmes distribués, présente les facilités offertes par DISIX, puis décrit l'architecture proposée pour cette couche et enfin son implantation. En conclusion, un aperçu des améliorations envisageables de la couche DISIX ainsi qu'une comparaison avec d'autres architectures possibles sont présentés.

I. Présentation d'UNIX.

I.1. Introduction.

Le système d'exploitation UNIX a été conçu et mis au point par Ken Thomson et Dennis Ritchie dans les années 70 aux Laboratoires Bell. La première version disponible écrite en langage C (Version 6) a été réalisée en Mai 1975. Actuellement, il existe plusieurs versions, la version System V de la Western Electric Company et la version BSD 4.2 de l'université de Berkeley sont les deux versions les plus récentes.

Le système V diffusé depuis Janvier 1983 par la BELL, n'offre pas de grandes différences par rapport à UNIX V7. Il inclut néanmoins de nouveaux utilitaires et quelques extensions systèmes, en particulier au niveau de l'IPC (communication entre processus).

Dans le cadre du projet, nous utiliserons la version système V implantée sur les machines SPS7 (licence CNET-SM90) de Bull Sems (Echirolles). Il s'agit en fait du logiciel d'exploitation temps partagé SPIX (SPIX est une marque déposée Bull Sems) qui inclut le système V version 2 et sert de support à un ensemble de services supplémentaires tels que atelier logiciel et télécommunications.

Nous présenterons les aspects principaux de ce système qui ont influencé l'architecture de l'application et sa mise en oeuvre, et dont la compréhension des mécanismes généraux nous était indispensable.

I.2. Les processus.

I.2.1. Définition.

Un processus est l'exécution sur une unité de traitement d'une image composée d'un ensemble d'objets tels que code, données, jeu de registres.

Le processus, composée de 3 segments (code, données, pile), s'exécute soit dans l'espace noyau, soit dans l'espace utilisateur.

I.2.2. Espace noyau.

- Le segment de code est le code du système d'exploitation.
- Le segment de données contient les tables gérées par le système.

• Le segment de pile est utilisé principalement pour la gestion des interruptions, des déroutements et des appels systèmes.

I.2.3. Espace utilisateur.

Il s'agit d'un espace virtuel segmenté composé également de 3 segments :

• Le segment de code est généralement protégé en écriture, et peut alors être partagé entre plusieurs utilisateurs.

• Les données sont de type statique (dites externes) ou dynamique.

• La pile contient les paramètres des appels de fonctions et la valeur des registres sauvegardés pendant le traitement de ces appels. Ces variables ont une durée de vie égale à celle du traitement des fonctions auxquelles elles sont associées. L'espace alloué à la pile peut être modifié dynamiquement pendant l'exécution du processus.

I.2.4. Modes d'exécution.

I.2.4.1. Principes.

Le code du processus peut s'exécuter soit en mode utilisateur, soit en mode noyau. Les utilisateurs ne peuvent pas exécuter directement des programmes ou manipuler des données dans l'environnement système. Par contre, ils peuvent demander au système d'effectuer certaines tâches pour eux par le biais d'un mécanisme particulier appelé "appel système".

Le mode système d'UNIX est implanté sur le mode noyau. En mode système les processus sont autorisés à exécuter des instructions privilégiées et à manipuler des données privilégiées.

En contrôlant l'accès du code utilisateur au mode système, celui-ci assure sa protection et son intégrité.

I.2.4.2. Passage en mode système.

Deux types d'événements ont pour résultat le passage en mode système :

• Les événements internes ou synchrones provoqués par le processus en train de s'exécuter : appels systèmes, déroutements.

• Les événements externes ou asynchrones provoqués par les

périphériques et l'horloge interne de "slicing" qui a la priorité la plus haute.

I.2.4.3. Appels systèmes.

Ce sont des types particuliers de déroutement qui ne peuvent se produire que lorsque le processus s'exécute en mode utilisateur.

Ils sont l'interface entre l'utilisateur et le système. Ils permettent :

- Les entrées/sorties (READ, WRITE, IOCTL ...).
- La gestion et le controle des fichiers (OPEN, CLOSE, MKNOD ...).
- l'accès aux données système (GETPID, GETUID ...).
- La gestion et la synchronisation des processus (SIGNAL, WAIT ...).
- La gestion du stockage des données du processus (BRK, SBRK).

I.2.5. Etats d'un processus.

I.2.5.1. Principes.

Un processus qui demande une entrée/sortie, ou une ressource partagée déja utilisée, ou qui cherche à se synchroniser avec un autre processus, sera endormi dans une file d'attente dépendante du type d'événement. Lorsque sa requête sera satisfaite, il sera simplement placé dans la file d'attente des processus éligibles. Enfin, plus tard, il sera élu et son exécution reprendra.

I.2.5.2. Swapping.

Le processus qui déroule le code de "swapping" a pour numéro 0. C'est le premier processus créé lors du lancement initial du système. Il lancera le processus de numéro 1 qui se charge de l'initialisation de tous les terminaux raccordés à la machine. Il ne peut être "swappé".

Ce processus est responsable de la gestion de la mémoire principale. Son travail, lorsqu'il est actif, est de "swapper" le maximum de processus endormis et éligibles, c'est-à-dire : copier sur une zone de mémoire secondaire réservée à cet effet, l'image du processus, sauf les données système, et libérer les emplacements mémoire correspondant.

Dans le cas où plusieurs utilisateurs partagent un segment de texte, à la première utilisation du segment, celui-ci sera amené en mémoire principale. Si le segment n'est plus utilisé et qu'il y a un besoin en mémoire principale, l'espace mémoire du segment sera simplement libéré.

I.2.6. Création d'un processus.

I.2.6.1. Principe.

Le seul moyen, dans UNIX, de créer un processus est d'exécuter l'appel système FORK. Le processus qui fait le fork, processus père, crée un processus fils qui aura des segments de code, données et pile identiques à ceux du père.

C'est ainsi que les fichiers ouverts par le père le resteront pour le fils, avec le même pointeur de lecture/écriture. Les procédures de traitement des interruptions seront également héritées par le fils.

La différence au niveau système consiste à la réinitialisation des paramètres de comptabilité du processus. Pour l'utilisateur, le seul moyen de différencier le père du fils se situe au niveau de la valeur de retour de l'appel système exécuté sans erreur : 0 pour le fils et numéro du processus fils pour le père.

Le processus fils commence son exécution avant le père.

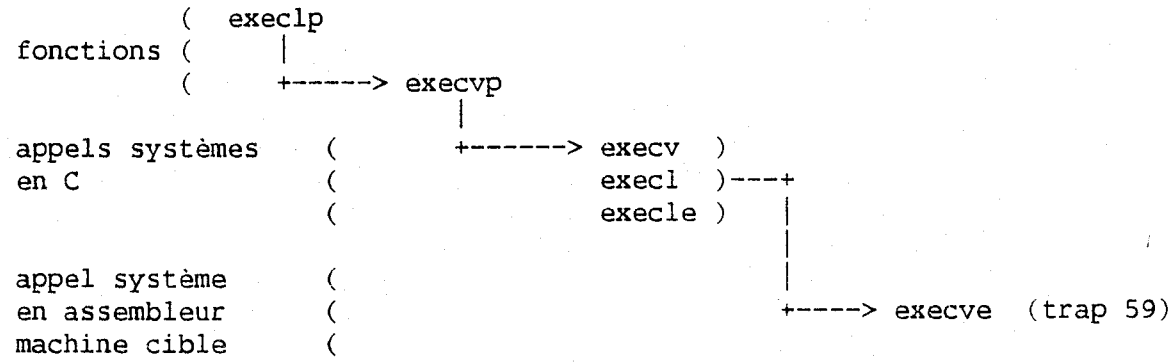
I.2.6.2. Exécution d'un programme.

Il existe 4 appels système et 2 fonctions pour exécuter un programme. Les appels système EXECL et EXECV sont utilisés selon que le nombre d'arguments à passer au programme est fixe ou variable. Les appels système EXECLE et EXECVE permettent, en plus, de modifier l'environnement courant du processus. Les 2 fontions EXECLP et EXECVP permettent, par rapport aux 2 premiers appels systèmes, de donner le nom du programme en relatif, avec les règles de recherche de la variable d'environnement PATH.

En ce qui concerne les arguments passés, le premier (numéro 0) doit être identique au nom du programme lancé et, en aucun cas, il ne doit être omis ; le second argument de l'appel système (numéro 1) sera le premier disponible pour le programme lancé.

De plus, certains programmes recueillent de l'information sur le premier argument. Exemple, dans le cas du "shell", si l'argument 0 est le caractère '-' il s'agit d'un "shell" de "login", et des commandes et des paramètres d'initialisation seront lus dans le fichier ".profile" (.login dans le cas du Cshell).

Cependant, tous ces appel systèmes et fonctions se ramènent à l'exécution d'un seul appel système : EXECVE.



Exemple :

- execve (nom, argv, envp)

où :

'nom' est le nom du fichier à exécuter,

'argv' est un pointeur sur les arguments associés au fichier à exécuter,

'envp' est un pointeur sur l'environnement de la future exécution qui peut avoir été modifié par le processus à l'origine de l'appel système.

Quand on exécute cet appel système, l'image mémoire est recouverte par celle du nouveau processus qui conserve les tables systèmes, en particulier les descripteurs de fichiers ouverts. Par contre, le traitement des signaux est réinitialisé à la valeur par défaut, c'est-à-dire la terminaison du processus.

IL n'y a pas de retour d'un appel système de ce type exécuté avec succès. Pour garder une trace de l'exécution d'un programme, la séquence normale est la suivante :

```
si fork() égale 0
alors exécuter le programme
sinon attendre que le fils ait terminé
fsi
```

I.2.7. Terminaison de processus.

Cette terminaison peut se faire de manière normale, provoquée par la fin du programme ou par l'appel système EXIT, ou de manière anormale par la réception d'une interruption non traitée.

A la fin de son exécution le processus entre dans la fonction exit(). Il perd son espace d'adressage et passe à l'état zombie, attendant que son père prenne en compte son statut de terminaison (appel système WAIT).

Si le père se termine avant le fils, le fils est rattaché au processus numéro 1 (init).

I.3. Environnement courant du processus.

Chaque processus sous UNIX possède un environnement sous la forme de variables d'environnement stockées dans un tableau de chaînes de caractères.

Ce tableau est pointé par la variable externe "environ". IL est structuré de la façon suivante : chaque chaîne de caractères représente une variable et sa valeur, séparées par le caractère '=' ("variable=valeur").

Il existe des variables d'environnement standards :

- PATH : liste des répertoires à balayer pour trouver un fichier dont le nom est en relatif. Elle sert en particulier pour le "shell".

- HOME : nom du répertoire de "login" et du répertoire par défaut; variable utilisée également par le "shell".

- TERM : nom du terminal; variable utilisée par les éditeurs pleine page.

- USER : nom du propriétaire du processus.

etc...

Ces variables standards sont en fait créées par le "shell" au moment du "login" et sont définies dans un fichier particulier ".profile" situé dans le répertoire de "login" (.login pour Cshell).

En plus de ces variables d'environnement standards, l'utilisateur peut créer autant de variables d'environnement qu'il juge nécessaire tout en

respectant la syntaxe ci-dessus.

Exemple :

VERSION=SPIX pour paramètrer la version de système UNIX dans un fichier de commandes de compilation.

Il existe une fonction prédéfinie ("getenv") qui donne la valeur d'une variable d'environnement définie. Par contre, la fonction qui permettrait d'assigner une valeur à une variable d'environnement définie ("setenv") n'est pas disponible dans les bibliothèques. Le problème est identique pour le système V et Berkeley 4.2.

Nous avons vu que le seul moyen de faire exécuter un programme est l'appel système EXECxx qui recouvre l'image mémoire par celle du programme à exécuter, d'où la perte des valeurs de toutes les variables. Si l'on désire passer des valeurs au processus fils, on utilisera les variables d'environnement.

I.4. IPC : la communication entre processus.

I.4.1. But.

Le but de l'IPC est de permettre l'échange d'information et la synchronisation entre des processus, avec ou sans filiation, s'exécutant sur la même unité de traitement.

I.4.2. Critères de sélection.

I.4.2.1. Echange ou synchronisation.

- Echange d'information :
  - les "pipes",
  - les messages,
  - la mémoire partagée,
  - la valeur retournée à la terminaison d'un processus.
- Synchronisation :
  - les signaux,
  - les sémaphores,
  - l'attente de la mort d'un processus fils.

I.4.2.2. Processus avec ou sans filiation.

- Processus avec liens de parenté :

  - les "pipes",
  - l'attente de la mort d'un processus fils,
  - la terminaison d'un processus.

- Processus liés par un même identificateur (utilisateur, goupe) :

  - les signaux.

- processus sans liens aucun :

La version V7 d'UNIX ne dispose que des fichiers disques potentiellement partageables entre tous les processus du système.

La version système III d'UNIX propose des "pipes" nommés ou fichiers FIFO possédant un nom (MKNOD), comme un fichier disque, et pouvant être ouverts par un processus sans filiation avec celui qui a créé le fichier; mais avec les inconvénients liés à la manipulation de fichiers ("swapping" par exemple).

La version système V d'UNIX propose en plus les messages, la mémoire partagée et les sémaphores. Ces trois mécanismes sont mis en oeuvre de façon identique par l'utilisateur. Seul les messages, qui sont utilisés pour l'application du projet DISIX, seront décrits plus loin.

I.4.3. Les signaux.

Les signaux fournissent un moyen d'indiquer de manière asynchrone à l'utilisateur, les événements intérieurs et extérieurs au processus qui s'exécute.

Ils sont au nombre de 19 sur système V et 27 sur BSD 4.2.

Les signaux que reçoit un processus sont enregistrés dans un champ de la table d'entrée du processus résidante en mémoire. Chaque élément binaire (bit) de ce champ concerne un type de signal.

Il faut noter que les signaux ne sont pas mis en file d'attente. Si un signal est positionné et qu'un autre du même type est positionné avant le traitement du précédent, alors le traitement n'aura lieu qu'une fois.

Le traitement d'un signal est réalisé par le processus qui le reçoit. Il doit donc être effectué quand ce processus est actif. De plus, le test d'existence d'un signal est fait lorsque le processus est en mode système. C'est pourquoi le traitement d'un signal est réalisé quand le processus

s'exécute en mode système ou à l'aide d'un trap quand le processus est en mode utilisateur.

Dans le cas où l'interruption est arrivée alors que le processus est endormi, dans l'attente, par exemple, de la fin de l'exécution d'un appel système d'entrée/sortie, le processus endormi est reveillé, son état est réinitialisé à celui du début de l'appel système et une erreur indique la fin anormale de l'appel système suite à la prise en compte de l'interruption.

Les applications sont différentes :

- tout processus peut traiter la plupart des signaux en leur associant par l'appel système SIGNAL une procédure de traitement spécifique.

- Les processus qui ont le même identificateur d'utilisateur peuvent communiquer par KILL.

- Les processus issus du même terminal seront concernés par les interruptions générées au clavier : break, interrupt, quit.

- Le processus père aura connaissance par le signal SIGCLD de la mort d'un fils. Si le père était endormi sur WAIT, il sera réveillé et reprendra son exécution.

- A l'intérieur d'un programme on pourra traiter les erreurs survenues au niveau des appels système.

Après avoir été pris en compte par une procédure utilisateur, le traitement du signal est remis à son état par défaut, c'est-à-dire que la prochaine réception de ce signal entraînerait la mort du processus. Il est donc nécessaire de réarmer la même procédure de traitement dans la procédure utilisateur.

I.4.4. Les Pipes.

Appel système PIPE.

Il s'agit d'un canal de communication uni-directionnel, assimilable à une file d'attente FIFO et associé à un fichier (numéro d' "inode"), avec un accès en écriture et un en lecture. l'utilisateur se servira d'un descripteur de fichier pour lire et d'un pour écrire sur le "pipe".

Ce moyen de communiquer entre processus peut paraître pratique pour l'utilisateur, mais comporte en fait 2 contraintes :

- Pour que 2 processus puissent s'envoyer des messages par "pipe", il faut qu'ils aient un ancêtre commun qui aura ouvert le "pipe". Les fils

hériteront des descripteurs du fichier associé. Cette limitation rend impossible l'utilisation de ce mécanisme dans le cadre de notre application où les processus n'ont à priori aucun lien entre eux.

• Le "pipe" est associé à un fichier, et utilise le système de gestion des fichiers en mémoire, avec la gestion des tables correspondantes et des tampons qui peuvent être "swappés". Ce mécanisme limite les performances des communications via des "pipes".

• La gestion d'échange de messages par "pipe" nécessite de structurer l'information et de synchroniser les entrées/sorties. D'où une programmation plus complexe.

I.4.5. Les messages.

Les messages n'ont pas les 3 inconvénients des "pipes" décrits plus haut mais nécessitent l'utilisation de primitives spécifiques que nous verront plus loin. Ces primitives posent un problème de portabilité avec les autres versions d'UNIX.

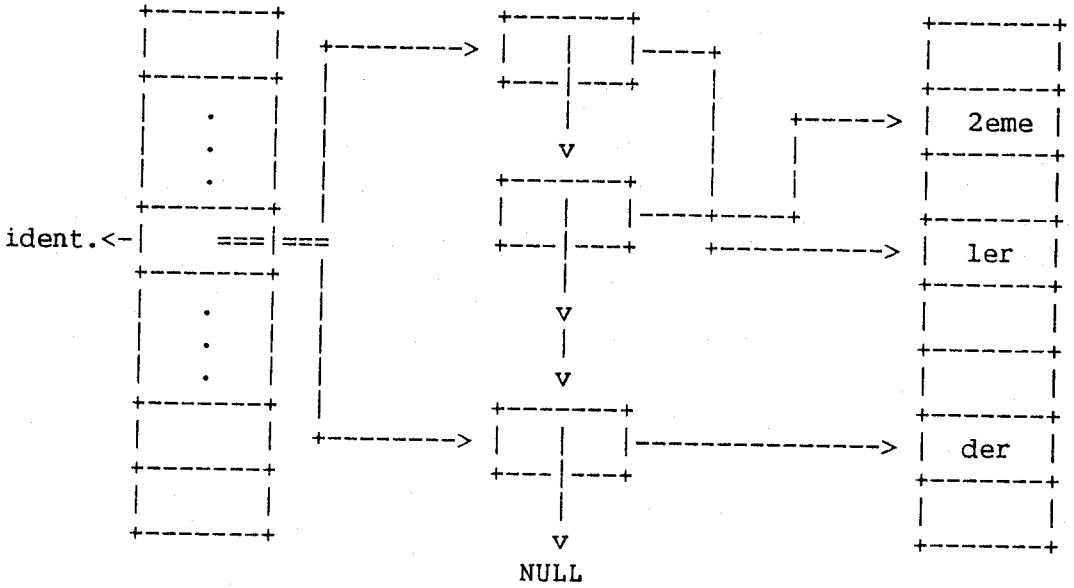
• La filiation n'est pas nécessaire et il n'y a pas de limitation, hors contraintes système, dans le nombre de processus pouvant communiquer ensemble sur la même queue de messages, ou le nombre de queues de messages que peut utiliser un processus.

• L'implantation est faite dans une zone mémoire résidante offrant un mécanisme performant. La gestion est de type FIFO avec la possibilité d'accès sélectif.

I.4.5.1. Structure et identification.

Les queues de messages sont implantées sur des en-têtes de queues de messages, chaque en-tête pointant sur des tampons contenant les données.

Les en-têtes sont chaînés sur des listes séparées.



Implantation des queues de messages.

Chaque entrée allouée de la table des queues de messages contient un pointeur sur le premier en-tête, un sur le dernier et une information structurée spécifiant les droits d'accès à la ressource, la taille maximum de la queue et l'état suite à la dernière opération effectuée; par exemple, le nombre de messages en queue.

L'appel système MSGGET :

ident <--- msgget (clé, drapeau)

retourne à l'utilisateur un identificateur unique associé à la clé et correspondant à une entrée dans la table des queues de messages. Pour une clé identique, le premier appel de msgget créera une queue de messages, les appels suivants se contentant de l'ouvrir.

Pour s'assurer que la queue de messages n'est pas accédée accidentellement par des processus étrangers à son application, le programmeur pourra utiliser la fonction FTOK rendant la clé unique à l'aide d'un nom de fichier, accessible au processus exécutant la fonction, et d'un caractère particulier :

clé <--- ftok (nom_de_fichier, char)

La clé peut être IPC_PRIVATE et dans ce cas la queue de messages est privée au créateur et à ses fils (fork).

I.4.5.2. Protection et contrôles.

L'accès à la queue de messages est protégé par les mêmes mécanismes que pour les fichiers UNIX, à savoir : identité du créateur de la queue de messages, droits en lecture et écriture pour le propriétaire, les utilisateurs du même groupe que celui du propriétaire et les autres. Le super-utilisateur a tous les droits.

L'appel système MSGCTL :

msgctl (ident, commande, tampon)

permet suivant la commande :

- de connaître l'état de la queue associée à l'identificateur "ident",
- de modifier les protections et la taille de la queue,
- de supprimer la queue et l'entrée correspondante dans la table.

I.4.5.3. Emission de message.

L'appel système MSGSND :

msgsnd (ident, message, taille, drapeau)

permet d'envoyer un message dans la queue de messages associée à "ident".

Le message est composé de 2 champs :

. Le type qui permet de sélectionner le processus destinataire du texte du message,

. Le texte proprement dit.

Si la queue de messages est pleine ou si les limites imposées par le système sont atteintes, le programmeur peut choisir d'attendre sur l'appel système, ou non, en positionnant "drapeau".

Dans le cas de non attente, le message n'est pas envoyé et l'appel système retourne immédiatement. Dans l'autre cas, l'exécution du processus est suspendue jusqu'à ce que les conditions de suspension disparaissent.

Des précautions sont à prendre pour le cas où le processus reçoit une interruption entre temps : le message n'est pas envoyé et un code d'erreur est retourné.

I.4.5.4. Réception de message.

L'appel système MSGRCV :

msgrcv (ident, message, taille, type, drapeau)

permet de lire un message dans la queue de messages associée à "ident".

Le message est composé de 2 champs :

. Le type correspondant à celui du message du processus émetteur,

. Le texte proprement dit.

"taille" est la taille maximum du texte que l'on veut recevoir.

"type" permet de sélectionner le message que l'on veut lire : le premier message s'accordant à "type" sera lu.

Si la queue de messages est vide pour le type désiré, le programmeur peut choisir d'attendre sur l'appel système, ou non, en positionnant "drapeau".

Dans le cas de non attente, l'appel système retourne immédiatement avec un code d'erreur. Dans l'autre cas, l'exécution du processus est suspendue jusqu'à ce qu'un messsage attendu soit placé dans la queue. Le problème de l'interruption du processus suspendu est identique à celui de l'émission de messages.

I.4.6. Exemple d'utilisation par DISIX.

Tout processus est identifié par le système par un numéro unique. En prenant comme type du message émis le numéro du processus destinataire, et en mettant dans le paramètre "type" de l'appel système de réception le numéro du processus exécutant cet appel système, on obtient un mécanisme d'échange fiable entre 2 processus sans filiation. C'est ce mécanisme qui sera utilisé dans la communication locale pour DISIX.

I.4.7. Résumé des problèmes d'IPC.

Ce résumé permet de mieux appréhender les problèmes d'IPC dans les versions différentes d'UNIX.

- transfert de données ("pipes") :
  - incapacité d'attendre des entrées de plusieurs sources.
  - pas de communication entre des processus sans lien de parenté.
  - pas d'identification de l'origine ou du type des données.
- synchronisation (signaux) :
  - avortement des appels systèmes en attente (entrées/sorties).
  - des événements peuvent passer inaperçus : seul le dernier signal est gardé.
  - pas de protection dans le même groupe de processus.
  - pas de communication entre des processus n'appartenant pas au même groupe.
  - pas d'identification de l'origine du signal.
  - pas de données associées au signal.
- échange de messages :
  - ce mécanisme semble bien adapté à la communication entre processus situés sur une même machine.

I.4.8. Récapitulatif des systèmes d'IPC.

UNIX	V7	S III	S V	BSD 4.2
signaux	15	19	19	27
pipes	4096o (s)	5120o (s)	5120o (s)	4096o (s)
pipes nommés	N	5120o (s)	5120o (s)	N
messages	N	N	queues de messages (r)	sockets domaine UNIX
memoire partagée	N	N	O	N
sémaphore	N	N	O	N
IPC generalisé	N	N	(1)	sockets domaine INTERNET

(1) : mécanisme offert par la MAD sur SPS7.
(s) : "swappable".
(r) : résident.
O : oui.
N : non.

I.4.9. Options et portabilité.

Le mécanisme d'échange de messages du système V, décrit ci-dessus, convient parfaitement à un IPC local. IL est plus souple et efficace que celui des "pipes", mais il pose des problèmes de portabilité sur V7, système III et les versions de Berkeley. Pour l'architecture de DISIX retenue, il sera clair que le portage sur V7 est impossible [Martin84] paragraphe 3.4, à moins d'utiliser les fichiers normaux. Par contre, une version sur BSD 4.2 est envisageable en reprogrammant uniquement l'interface de communication dans le domaine local avec les "sockets". Une version est également possible à réaliser sur système III grâce aux "pipes" nommés, avec une réduction notable des performances.

Dans le système V, nous remarquons l'absence d'un mécanisme d'attente multiple tel que celui offert par la primitive select de BSD 4.2. Ce qui nécessitera la création de processus spécialisés dans la réception de messages venant du domaine local ou du réseau, plus un mécanisme de concertation entre les dits processus.

l'IPC système V est orienté communications internes. Il n'offre pas un mécanisme généralisé tel que les "sockets" de BSD 4.2 dont la mise en oeuvre est relativement compliquée pour l'IPC interne. Il faut donc prévoir une interface supplémentaire pour les communications externes.

I.5. Le système de fichiers.

Le système de fichiers est l'un des concepts fondamentaux d'UNIX. Sa simplicité et son efficacité expliquent le succès de ce système d'exploitation et les réalisations nombreuses d'applications permettant l'accès distribué aux fichiers à travers un réseau.

I.5.1. Buts.

Il s'agit principalement de permettre l'accès à un fichier du système de fichiers par un nom, sans se préoccuper du périphérique sur lequel réside ce fichier.

Pour cela, le système de fichiers fournit une structure logique pour l'ensemble des fichiers du système, un moyen d'identifier de manière unique chaque fichier et de restucturer la hiérarchie.

I.5.2. Généralités.

On trouve sur le système UNIX 4 types de fichiers :

- Les fichiers normaux contiennent tout ce que l'utilisateur veut y mettre et sous la forme qu'il désire. Par exemple : le texte de ce rapport, les programmes sources de DISIX en langage C, les binaires ...

- Les répertoires contiennent des données structurées (nom du fichier, numéro d' "inode") qui permettent de faire la correspondance entre un nom de fichier et le fichier lui-même. Cette structure est contrôlée par le système qui seul peut accéder en écriture à ce type de fichier. L'utilisateur a uniquement le droit de lecture. Les 2 premières entrées d'un répertoire concernent lui-même et son ascendant direct. On voit ainsi se dessiner la structure hiérarchique du système de fichiers.

- Les fichiers spéciaux représentent les périphériques de transmission et de stockage des données. L'information n'est pas stockée sur le système de fichiers mais passe directement de la mémoire cache aux coupleurs de périphériques. l'utilisateur a l'accès en lecture et en écriture moyennant les protections et le type de périphérique.

- Les "pipes" nommés qui sont utilisés pour la communication entre processus depuis le système III sur les versions d'AT&T. Ces fichiers contiennent ce que l'utilisateur veut y mettre dans la limite du stockage (5120 octets) qui se fait uniquement dans le cache. La structure est contrôlée par l'utilisateur.

L' "inode" est une information interne au noyau UNIX, invisible à l'utilisateur normal, qui permet de référencer un fichier et un seul. L' "inode" contient tous les attributs d'un fichier pour la manipulation et le stockage : type, protection, taille en octets, adresses des données par indirections successives, dates d'accès, nombre de liens (plusieurs noms de fichiers pouvant être associés au même "inode"). Dans le cas des fichiers spéciaux, les champs adresse ne contiennent pas des pointeurs sur des blocs de données mais un numéro identifiant le type de périphérique (majeur) et un numéro de canal (mineur).

I.5.3. Structure.

Le système de fichier est structuré logiquement sous forme d'arborescence. L'arbre débute à la racine ('/') qui est un répertoire spécial marquant la frontière ascendante et descendante. L'accès à l'arbre se fait par des noms de fichiers de manière absolue par rapport à la racine, ou relative à la position courante.

Il n'est pas évident pour l'utilisateur face à cette hiérarchie que les fichiers sont physiquement séparés.

Physiquement, un système de fichiers est un ensemble de blocs contigus sur un stockage de mémoire secondaire. Le système de fichiers est organisé de la façon suivante :

- Le premier bloc contient le programme de lancement du système (boot) si le système de fichiers concerné est celui à partir duquel UNIX doit être lancé, sinon il ne sera pas utilisé.

- Le deuxième bloc, appelé super-bloc, contient des informations particulières destinées au système pour l'opération de "montage" que nous verrons plus loin et pour des utilitaires de maintenance.

- Les blocs suivants contiennent la "iliste" des "inodes" (plusieurs par blocs) des fichiers stockés sur le système de fichiers. l' "inode" numéro 2 est celui de la racine du système de fichiers.

- Tous les autres blocs comportent les données, sans organisation ou structure particulière, contenues dans des blocs pointés par les "inodes".

I.5.4. Implantation.

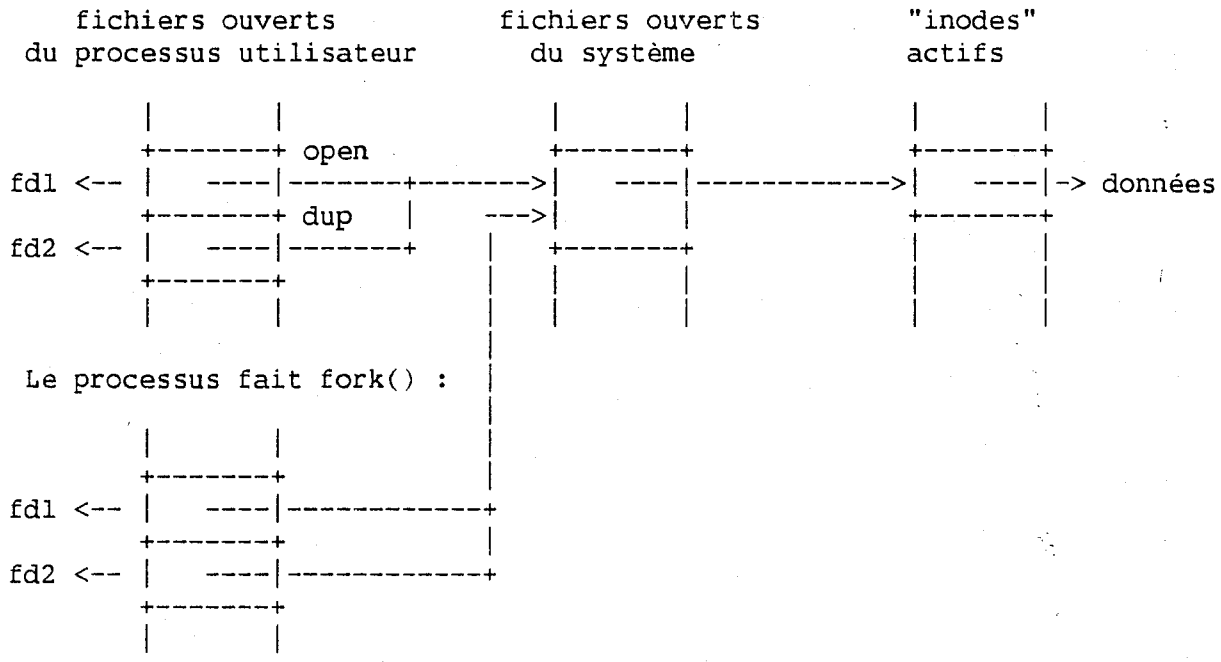
En mémoire résidante se trouvent le super-bloc du système de fichiers, la table des "inodes" actifs et la table des fichiers ouverts.

- La table des "inodes" permet de gérer des copies des "inodes" des fichiers actifs et d'avoir un seul point de contrôle si plusieurs processus se partagent l'accès à un fichier. Chaque "inode" de cette table est référencé de manière unique par le majeur et le mineur du périphérique, et par le numéro d' "inode" qui est en fait le rang par rapport au début de la "iliste".

- La table des fichiers permet de stocker les informations pour chaque fichier ouvert : type d'accès, nombre de descripteurs ouverts, pointeur sur la table des "inodes". Une entrée est créée suite aux appels systèmes OPEN, CREAT, PIPE.

l'utilisateur se sert d'un descripteur de fichier (fd) qui est un index dans la table (gérée par le système) des fichiers ouverts par lui-même,

une entrée dans cette table pointant sur la table générale des fichiers.



Exemple dans un cas de création de processus.

I.5.5. Montage.

Le système de fichiers est en fait une collection de systèmes de fichiers tels que nous les avons décrits. Ils sont accessibles aprés avoir été "montés". L'opération de montage va rattacher logiquement la hiérarchie du système de fichiers à celle existante et créer une entrée dans la table de montage.

- la commande : /etc/mount périph dir

monte le système de fichiers du périphérique de stockage "périph" sur le répertoire "dir", c'est-à-dire que "dir" se substitue à la racine du système de fichiers du périphérique. Ce qui était sous "dir" est perdu logiquement.

- la commande : /etc/umount périph

réalise l'opération inverse.

Ces commandes sont réservées à l'usage du super-utilisateur.

La table de montage permet l'accès physique à n'importe quel fichier

de stockage par les numéros de majeur et mineur, un pointeur sur l'entrée de la table des "inodes" actifs qui contient l' "inode" du répertoire sur lequel a été monté le système de fichier et un pointeur sur l'entrée de la table des "inodes" actifs qui contient l' "inode" de la racine du système de fichier.

La table de montage permet donc de résoudre les noms de fichiers quand il faut passer d'un périphérique à l'autre : si la recherche sur nom de fichier aboutit à la table de montage, elle continuera sur l' "inode" de la racine du système de fichier correspondant.

I.5.6. Accès.

L'accès à un fichier se fait de l'une des deux manières suivantes :

- Soit à partir d'un "path", c'est-à-dire grâce à une chaîne de caractères qui permet d'accéder au fichier à partir de la racine :

  - d'une manière unique et absolue : /dir1/dir2/fic1. On qualifie d'absolu le fait que le nom parte de la racine de l'arborescence '/'.

  - de manière relative en parcourant l'arbre du système de fichier à partir de la position courante ('.'). On descendra en désignant fichiers et répertoires par leur nom, on remontera au père du répertoire courant par '..'. Plusieurs combinaisons sont possibles. Il sera donc nécessaire lors du traitement d'un "path", de le rendre unique et absolu.

- Soit à partir d'un entier nommé descripteur de fichier (fd), qui est retourné par les appels systèmes d'ouverture ou de création de fichier. Le système V autorise 64 fichiers ouverts simultanément, donc 64 descripteurs compris entre 0 et 63, BSD 4.2 en autorise 20. Par défaut le descripteur 0 est l'entrée standard, le descripteur 1 est la sortie standard et le descripteur 2 est la sortie standard pour les messages d'erreurs, le standard étant le terminal. Les appels systèmes d'entrées/sorties et de contrôle des fichiers utiliseront ce descripteur.

I.5.7. Protection.

Un utilisateur est connu par le système grâce à deux nombres :

- l'identificateur d'utilisateur

- l'identificateur de groupe.

le super-utilisateur a toujours comme identificateur d'utilisateur le numéro 0.

Le nom de l'utilisateur n'est généralement pas utilisé. Ces informations sont stockées dans le fichier "/etc/passwd", ainsi que le mot de passe enregistré sous une forme cryptée avec une clé générée aléatoirement par le système. Pour les systèmes UNIX que nous connaissons (système V sur SPS7, V7 sur Mini6, BSD 4.2 sur Vax), le mot de passe crypté est composé de 13 caractères : 2 lettres forment la clé, suivies de 11 caractères générées à partir de la clé et du mot de passe en clair.

Sur UNIX, il existe quatre catégories d'utilisateurs : le super-utilisateur qui a tous les droits , le propriétaire du fichier, les utilisateurs appartenant au même groupe que celui du propriétaire, et les autres. Nous rappelons que le fichier est référencé de manière unique par le numéro d' "inode" et que dans l' "inode" se trouve les attributs du fichier dont un champ utilisé pour les droits d'accès.

La protection du fichier est obtenue par combinaison de 12 bits de ce champ :

- bit 11 : "set user id". Le système changera temporairement l'identificateur de l'utilisateur en celui du propriétaire du fichier chaque fois que ce fichier sera exécuté. De manière générale ce mécanisme permet d'interdire l'accès direct à certaines ressources, mais autorise un accès contrôlé par l'exécution de certains programmes où ce bit est positionné (ex : mail, uucp).

- bit 10 : "set group id". Idem dans le cadre du groupe auquel appartient le propriétaire. l'utilisateur aura temporairement les droits de ce groupe.

- bit 9 : "sticky". Il permet d'empêcher le "swapping" entre la première et la dernière utilisation d'un segment de texte partageable.

- bit 8 : lecture -
- bit 7 : écriture | pour le propriétaire
- bit 6 : exécutable -

- bit 5 : lecture -
- bit 4 : écriture | pour les membres du même groupe
- bit 3 : exécutable -

- bit 2 : lecture -
- bit 1 : écriture | pour les autres utilisateurs
- bit 0 : exécutable -

I.6. "Drivers".

Dans la suite, le mot "driver" sera utilisé pour désigner un gestionnaire de périphérique.

I.6.1. Objet.

Une des caractéristiques principales d'UNIX, pour les applications, est de fournir une vue sur le matériel de stockage, de restitution et de communication avec le monde extérieur, complètement indépendante du type de périphérique.

L'interface entre UNIX et un type de matériel périphérique est fournie par un "driver" de périphérique qui permet de faire communiquer les primitives standard d'UNIX avec le contrôleur spécifique à chaque type de périphérique.

I.6.2. Généralités.

Les périphériques contenant des données adressables et réutilisables (disques) sont considérés comme étant de type bloc, les autres étant de type caractère (interfaces de ligne de communication, terminaux). Ces derniers effectuent généralement des entrées/sorties pour un nombre variable d'octets, de manière asynchrone et à des vitesses relativement lentes, alors que, par exemple, les entrées/sorties sur disque sont à l'initiative du système et portent sur des blocs. Lorsque l'entrée/sortie est terminée, le système est informé par une interruption générée par le périphérique.

Dans tous les cas les entrées/sorties sont tamponnées. Pour les disques, cet ensemble de tampons contitue un cache. Sur une lecture, le cache est examiné en premier; si le bloc est trouvé, les données sont rendues disponibles à l'utilisateur sans opération physique; sinon, le bloc le plus ancien recevra les données requises.

Le cache a deux avantages :

- minimiser les commutations de processus sur les opérations d'entrées/sorties en évitant celles-ci à chaque mise à jour d'un octet.

- l'unité minimum d'entrée/sortie est l'octet. Si l'utilisateur ne fait pas des entrées/sorties sur des blocs physiques de disques (secteurs), il faut pouvoir stocker les données non désirées.

Et un inconvénient majeur :

- il peut y avoir un problème de cohérence des données en cas de panne, si les dernières modifications n'ont pas été portées sur mémoire

secondaire.

I.6.3. Structure.

Un "driver" est composé d'un ensemble de procédures indépendantes. Ces procédures sont appelées par le niveau supérieur à travers le système de fichiers:

- OPEN initialise le contrôleur de périphérique et lui assigne un niveau d'interruption.

- CLOSE nettoie les tampons d'entrée/sortie en exécutant celles qui sont en cours puis arrête le contrôleur.

- READ et WRITE transforment une demande d'entrée/sortie du niveau supérieur en bloc de contrôle d'entré/sortie exécutable par le contrôleur.

- IOCTL modifie les paramètres d'un périphérique : vitesse, validité, flux ...

D'autres procédures sont exécutées sur réception d'une interruption du périphérique.

A chaque type de "driver" est associé une table de configuration qui est le seul lien entre le système de fichiers et les "drivers". Une entrée dans cette table contient les adresses de toutes le procédures d'un "driver" rangées dans un ordre déterminé. Le système de fichier accède à une entrée de la table par le majeur du périphérique et active une procédure du "driver" en lui passant en paramètre le mineur.

I.7. Aspects particuliers d'UNIX.

I.7.1. Le "shell".

Le "shell" est l'interprétateur du langage de commande sous UNIX. C'est lui qui prend en compte toutes les commandes tapées au terminal et crée les processus nécessaires en initialisant leur environnement.

Il permet de rediriger sur des fichiers ou des processus les entrées/sorties standards qui sont par défaut le clavier et l'écran. Pour cela, il effectue l'ouverture ou la création des fichiers nécessaires et ouvre les "pipes" pour la communication entre processus.

Le "shell" est un point clé de l'utilisation d'UNIX. Il est important d'en connaître les mécanismes de base dans le cadre d'un système distribué.

I.7.2. STDIO.

STDIO propose un ensemble de fonctions qui permettent de tamponner, au niveau utilisateur, des entrées/sorties structurées. Toute fonction de STDIO (FOPEN, FCLOSE, FREAD, FWRITE...), effectue des tâches administratives (notamment gestion d'un tableau de structures contenant : descripteur de fichier, adresse tampon, taille tampon, pointeur courant, type d'accès), dans l'environnement de l'utilisateur et ensuite fait des appels systèmes.

Exemple :

Lorsqu'un programme demande à effectuer une lecture sur un fichier disque avec la fonction FREAD, le première fois le système alloue un tampon de la taille du bloc du type de disque, remplit le tampon avec l'appel système READ et fournit au programmeur le nombre de caractères requis tant que le tampon n'est pas vide et tant que la fin de fichier n'est pas atteinte. Lorsque le tampon est vide, on le remplit et ainsi de suite.

II. L'ENVIRONNEMENT DE COMMUNICATION DU SPS7.

II.1. Le SPS7.

II.1.1. Généralités.

Cette machine est issue d'une expérimentation entreprise par le CNET PARIS A et d'un prototype réalisé en 1981 pour répondre essentiellement aux points suivants :

- Accroissement possible de la puissance d'une architecture au moindre coût (bureautique, ordinateurs personnels).

- Intégration facile à des systèmes d'informatique répartie (résaux téléinformatiques).

- Haut degré de disponibilité.

Enrichie par l'INRIA, industrialisée par Bull, la structure multiprocesseurs du SPS7 offre un potentiel d'adapation important aux besoins des utilisateurs dans tous les domaines de l'informatique.

Le traitement des données et des entrées/sorties est réparti sur ce multiprocesseur qui peut réunir dans une même configuration 8 processeurs (MT) de 16 ou 32 bits, 16 modules d'échange (ME) de 8 ou 16 bits véritables canaux capables de gérer aussi bien la périphérie classique que des fonctions complexes de transmission : mémoire de masse, impression, visualisation et connexion aux réseaux publics, privés et locaux.

Les processeurs et les modules d'échanges communiquent entre eux par l'intermédiaire d'un bus commun, le SMBUS, indépendamment de la nature des microprocesseurs qui les équipent. Egalement raccordés au SMBUS, figurent des modules de mémoire commune (MMC) et un module de maintenance et de surveillance.

II.1.2. Les services de communication.

Le SPS7 offre des services de communication qui permettent de distribuer des applications sur plusieurs systèmes d'exploitation :

- dans le domaine interne limité à la machine.

- dans un domaine externe fermé, constitué de machines homogènes connectées à un réseau local de type Ethernet.

• dans le domaine OSI ouvert, constitué de machines hétérogènes interconnectées par des réseaux de type X25 public ou privé.

Plusieurs configurations sont possibles pour répartir, dans l'environnement décrit ci-dessus, les différents systèmes et services de communication nécessaires à l'utilisateur.

De façon générale, on trouvera dans les MT les interfaces utilisateur et les couches hautes de l'OSI (4 et 5), et dans les ME les couches basses de l'OSI (2 et 3).

Exemple :

• un MT SPIX pour les besoins généraux et la préparation de programme.

• un MT SPART pour les interfaces d'accès à la session et au transport, plus la session et le transport.

• un ME X25-2, X25-3 sous SPART.

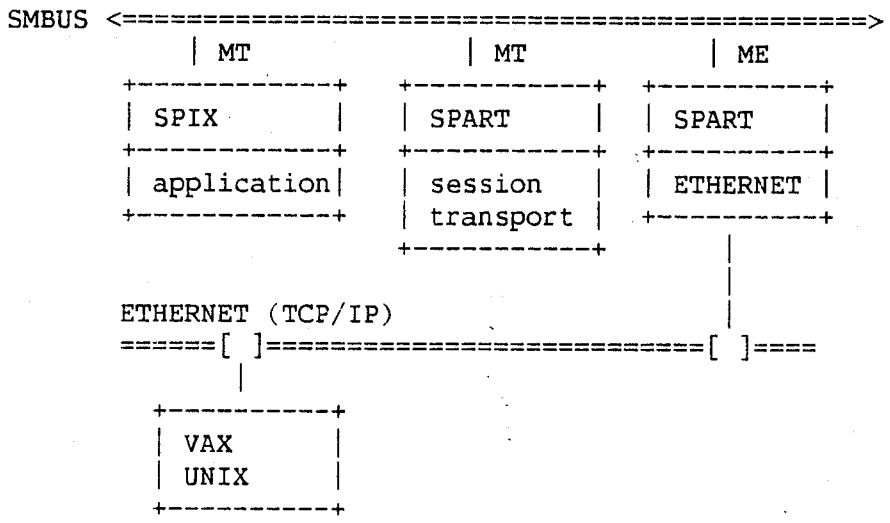
• un ME Ethernet sous SPART.

• des ME's disque, bande, imprimante, lignes asynchrones.

II.1.3. Le module Ethernet.

Le module d'échange Ethernet permet la gestion des couches réseau propre à Ethernet (3 premiers niveaux de l'OSI). Il offre la possibilité de communications à haut débit à l'intérieur d'un réseau local ou l'accès à un site distant à travers un réseau public via une passerelle Ethernet/X25.

Il prend en compte les procédures d'émission et réception, gère les trames perdues et compte les erreurs. Il supporte un débit de 100 trames par seconde, pour des trames de 1250 octets, et accepte jusqu'à 64 connexions simultanées.



Exemple de communication entre systèmes.

II.2. Architecture pour Applications distribuées.

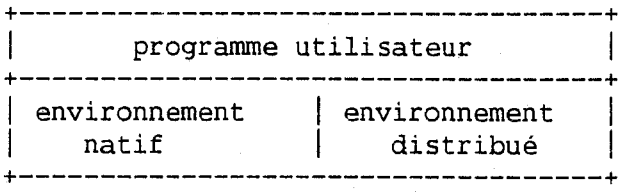
Nous utiliserons pour DISIX la MAD qui est une composante de DAA (Architecture pour Applications Distribuées).

II.2.1. Introduction.

L'objectif de cette architecture pour applications distribuées est de fournir des services et des structures pour supporter le traitement interactif distribué.

Les caractéristiques principales de cette architecture sont :

- la prise en compte des travaux de l'OSI dans ce domaine.
- l'adaptation aux différents domaines de distribution, consistant à offrir un service identique tout en améliorant les performances par des simplifications liées au domaine. Par exemple, dans le domaine interne, la communication entre processus ne nécessite pas les protocoles mis en oeuvre sur les réseaux.
- la fourniture d'un environnement concerné par l'exécution immédiate coordonnée de programmes distribués.



Environnement distribué dans un processeur.

II.2.2. l'environnement distribué.

L'environnement distribué offre ses services par une interface d'accès sous la forme de primitives disponibles sur SPIX et SPART, permettant à l'utilisateur de développer des applications dont les programmes s'exécutent sous différents environnements; par exemple, SPIX-MT et SPART-MT (session, transport).

II.2.2.1. Communication et synchronisation.

Les communications internes ou externes entre des entités qui appartiennent au système de communication, ou entre le système de communication et des programmes utilisateurs sont basées sur des "échanges" de messages appelés interactions.

La synchronisation entre les processus qui s'échangent des interactions est basée sur l'attente bloquante d'interaction :

- un processus est suspendu quand il est en attente d'interaction.
- il est réveillé quand une interaction qui lui est destinée arrive.

Une autre forme d'interaction est également envisageable. De type non bloquant, elle permet au processus émetteur de poursuivre son activité.

II.2.2.2. Composants de l'environnement.

l'environnement est composé de trois composants de base :

- la méthode d'accès distribuée (MAD); sour forme de "driver" sur UNIX et d'agence sur SPART, elle assure l'interface avec les programmes utilisateurs.
- le gestionnaire mémoire; c'est un service d'optimisation de l'occupation de l'espace mémoire et du traitement; il est utilisé princi-

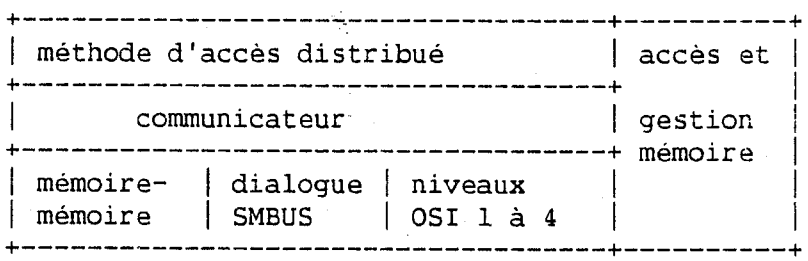
palement par la MAD et le communicateur.

• le communicateur; il assure la transmission des interactions de la MAD de manière fiable et adaptée au médium de transmission en étant présent dans chaque processeur appartenant au domaine de distribution. IL comprend trois niveaux qui seront utilisés, ou non, compte tenu de l'adresse du destinataire du message à transmettre :

- le niveau mémoire-mémoire utilise un mécanisme de boites aux lettres pour communiquer au sein d'un MT (IPC local), ou entre MT's par MML.

- le niveau SMBUS fait dialoguer les MT's et ME's via le SMBUS.

- le niveau transport-OSI permet l'échange entre machines interconnectées par réseau local (Ethernet) ou X25.



Composants de l'environnement distribué.

II.3. La Méthode d'Accès Distribuée MAD.

II.3.1. But.

Le but de la MAD est d'assurer l'interface de l'environnement distribué avec les programmes des utilisateurs.

Elle est présente sur tous les processeurs qui appartiennent au domaine de distribution.

La MAD est un système de communication par messages entre processus (IPC) généralisé qui unifie les communications internes entre des processus activés sur le même module ou sur des modules différents de la même machine, et externes entre des processus activés sur des modules de machines différentes interconnectées sur un ou plusieurs réseaux hétérogènes.

II.3.2. Les services.

La MAD gère des points d'accès au service de traitement interactif distribué qui sont apellés des "échanges", et offre différentes primitives qui permettent de gérer :

- les points d'accès : ouverture, fermeture.

- les services vis à vis du système de communication : activation de service et terminaison d'activation.

- la communication de type appel procédural (requête-réponse), ou IPC (émission).

- La synchronisation avec un mécanisme de réception avec ou sans attente.

La MAD est réalisée sous la forme d'un "driver" en environnement UNIX (se reporter au chapitre présentation d'UNIX), et d'une agence en environnement temps réel.

Les primitives ont été unifiées sous les deux environnement afin de faciliter la portabilité éventuelle des programmes d'un environnement à un autre.

II.3.3. l'accès sur UNIX.

l'accès à la MAD sur UNIX (SPIX) se fait sous la forme d'un "driver" (le "driver" MAD). Chaque point d'accès sera donc un fichier spécial (/dev/...).

II.3.3.1. Création d'échange.

L'"échange" est un objet de communication constitué d'une file de tampons associée à un dispositif de synchronisation. Un "échange" permet à un processus d'émettre et de recevoir des messages. Si "l'échange" est "vide", le processus peut demander à être suspendu; il sera réveillé lors de l'arrivée d'un message.

Créer un "échange" consiste à ouvrir un fichier spécial; "l'échange" sera identifié par un descripteur de fichier (fd).

- fd = open (nom)

"nom" est un nom de fichier spécial UNIX de la forme : /dev/mad...

Le mode d'ouverture du fichier n'apparaît pas car il n'est pas signi-

ficatif dans ce contexte.

II.3.3.2. Suppression d'échange.

Cette primitive supprime le point d'accès et libère une entrée dans le "driver" MAD.

- close (fd)

II.3.3.3. Activation de service.

Une application de type serveur doit se faire connaître vis à vis du système de communication, en faisant part à celui-ci de son "échange".

- ioctl (fd, ACTISER, struct)

"ACTISER" est le code fonction d'activation.

"struct" est l'adresse d'une structure qui contient l'identification de service.

II.3.3.4. Terminaison de service.

Un processus serveur qui se termine doit d'abord indiquer au système de communication l'arrêt de son service.

- ioctl (fd, DESACTISER, struct)

"DESACTISER" est le code fonction de terminaison du service.

"struct" est l'adresse d'une structure qui contient l'identification de service.

II.3.3.5. Emission avec réponse.

Ce type de communication se fait dans le but d'obtenir un service en réponse à une demande d'un processus client.

- ioctl (fd, REQUETE, struct)

"REQUETE" est le code fonction d'émission avec réponse.

"struct" est l'adresse d'une structure qui contient un pointeur sur l'UCB.

La primitive d'émission référence un bloc UCB destiné au processus serveur. Ce bloc comprends trois blocs :

- un bloc d'en-tête qui contient des paramètres de description, de routage et de compte rendu.

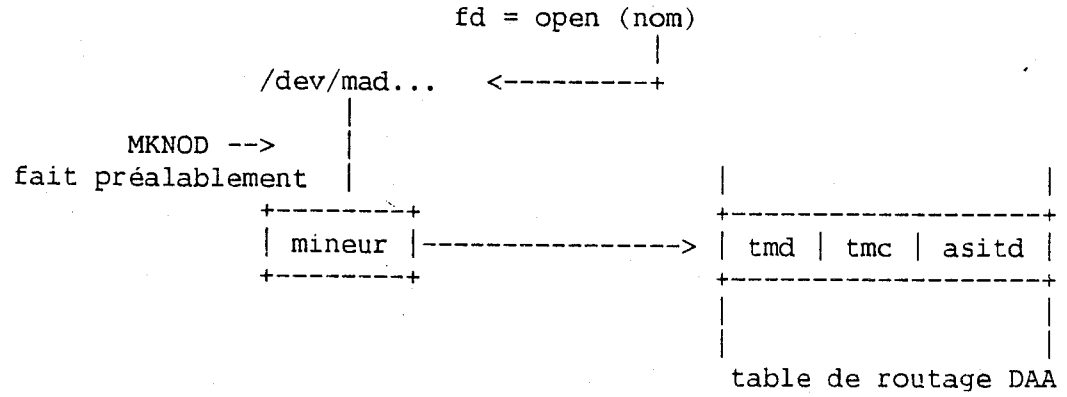
- un bloc de paramètres par valeur.

- un bloc de paramètres par adresses destinés plus précisément aux messages de processus clients et serveurs.

Le traitement immédiat de la requête consiste à retrouver les paramètres de routage du bloc UCB :

- soit l'adressage est explicite et les informations figurent dans le bloc UCB; cette option simplifie l'interface utilisateur, mais n'est pas adaptée aux changements de configurations.

- soit l'adressage est implicite; les paramètres sont dans une table de routage dont l'accès est fonction du numéro de mineur dans le "driver" MAD. Cette option est plus adaptée aux différentes solutions d'implantation matérielle, mais elle nécessite d'associer de façon figée l'application de l'utilisateur et un fichier spécial du "driver" MAD, et elle requiert la réservation de ce fichier pour ses besoins.



tmd : type et adresse physique du module destination.
tmc : type et adresse du module qui contient le communicateur inter-site.
asitd : caractéristiques de l'accès au site distant.

Opérations nécessitées par l'adressage implicite

II.3.3.6. Emission d'une réponse.

Utilisée par un processus serveur pour répondre à une requête client. Le bloc référencé est le même que celui reçu, avec les zones réponses documentées.

- ioctl (fd, REPONSE, struct)

"REPONSE" est le code fonction d'émission de réponse.

"struct" est l'adresse d'une structure qui contient un pointeur sur l'UCB de réponse.

II.3.3.7. Emission sans réponse.

Destinée à la communication entre processus de type IPC, cette primitive référence un bloc de type particulier (ICB) mais semblable dans sa structure à l'UCB.

- ioctl (fd, REPONSE, struct)

"REPONSE" est le code fonction d'émission de réponse.

"struct" est l'adresse d'une structure qui contient un pointeur sur l'UCB de réponse.

II.3.3.8. Réception.

Réception sur un "échange", utilisée par un processus pour l'attente d'un message de type requête, réponse ou émission.

- ioctl (fd, RECEPTION, struct)

"RECEPTION" est le code fonction de réception.

"struct" est l'adresse d'une structure qui contient 2 champs :

- le premier indique si la primitive est bloquante ou non.
- le deuxième contient l'adresse d'un tampon pour stocker le message.

II.4. Conception d'une application distribuée.

II.4.1. Généralités.

Dans l'environnement de communication du SPS7 une application distribuée est composée de plusieurs programmes qui s'exécutent dans différents modules de traitement ou d'échange.

Les différents programmes qui vont s'exécuter et interagir doivent être activés. Cette activation est effectuée soit automatiquement au lancement de chaque système sur chaque processeur, soit par une commande opérateur ou un programme spécifique.

Au lancement, un programme d'une application distribuée doit s'activer vis à vis de l'environnement distribué au moyen de la primitive d'activation de service.

II.4.2. Types d'interactions.

Les interactions mises en jeu dans une application distribuée sont de deux types:

- appel procédural.
- communication inter processus (IPC).

II.4.3. L'appel procédural distant.

Le mode requête-réponse se prête bien à l'appel procédural distant.

Un appel procédural distant s'obtient par la réalisation d'un sous programme qui réalise les fonctions suivantes :

- allocation d'un bloc UCB.
- initialisation de ce bloc : en-tête, paramètres par valeur et par adresses.
- appel de la primitive MAD requête.
- appel de la primitive MAD réception avec attente.
- extraction du code retour et des paramètres par adresse de retour.
- libération du bloc UCB.

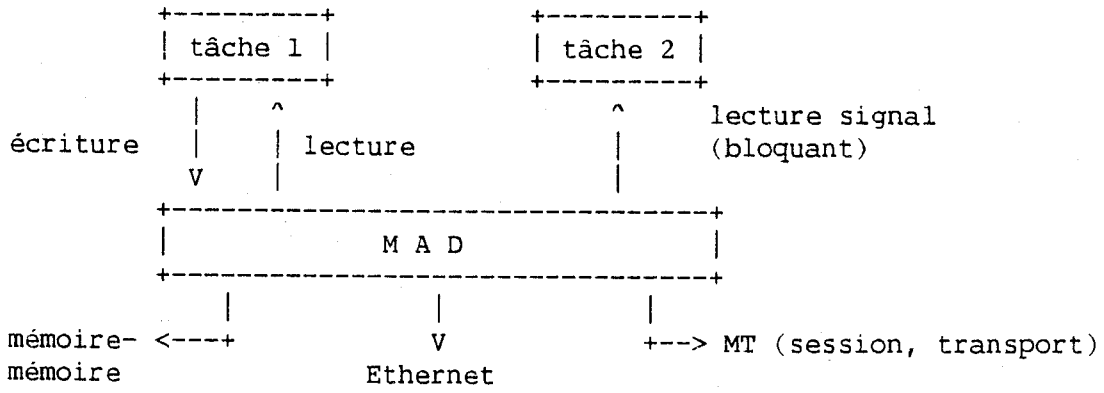
- retour au programme appelant.

Il faut noter que cet appel procédural peut être rendu asynchrone en plaçant, dans un autre sous-programme, la primitive de réception de type bloquant ou non. Dans ce cas, l'utilisateur pourra exécuter plusieurs requêtes et attendre une réponse sur la liste des requêtes précédemment exécutées.

II.4.4. La signalisation.

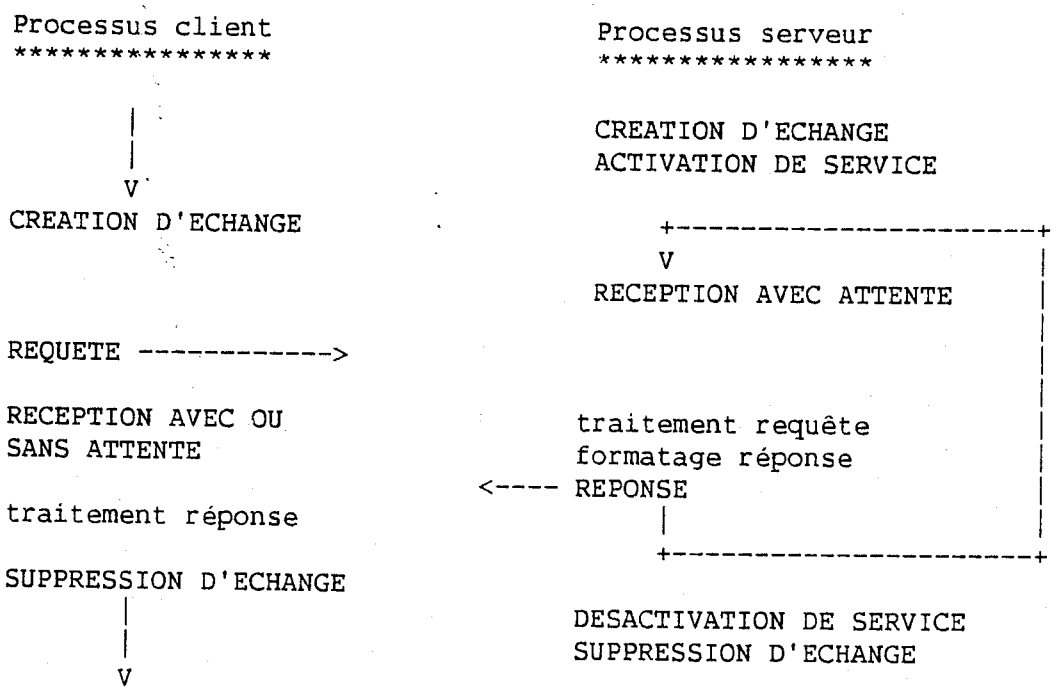
Pour prévenir les programmes de la terminaison anormale d'autres programmes de l'application et de permettre ainsi le relachement des ressources, un mécanisme de signalisation asynchrone a été mis en place.

Ce mécanisme supplémentaire doit être mis en place entre les processus clients et serveurs, sous la forme d'une requête de "lecture signal" dans chaque application concernée.



Exemple d'application.

II.4.5. Exemple de schémas de programme.



Exemple d'utilisation générale.

III. PRESENTATION DE DISIX.

III.1. Introduction.

Cette partie présente les facilités offertes à l'utilisateur par DISIX ; rappelons que DISIX est une méthode d'accès entre machines géographiquement éloignées, connectées via un réseau local Ethernet, et tournant sur le système d'exploitation UNIX.

Nous appelerons par la suite "système local" le système sur lequel travaille effectivement l'utilisateur. Nous appelerons "système distant" tout autre système connecté au réseau.

Dans un premier paragraphe, sera décrite la manière de "monter" l'arborescence d'un système distant sur l'arborescence du système local. Ensuite, figurera une description sommaire des droits de l'utilisateur à se connecter, ainsi que des facilités qui lui sont offertes (appels système, sous-programmes...). Enfin, nous parlerons des moyens dont il faut disposer pour implanter DISIX.

III.2. Montage d'un système distant.

Le montage d'un système distant sur l'arborescence du système local va permettre d'établir la connexion entre les deux systèmes via le réseau Ethernet, à travers la couche transport; il va lancer de chaque côté des processus qui vont permettre à l'utilisateur d'accéder directement aux ressources distantes : sur le système local un processus "sender" chargé d'envoyer les demandes de service, et sur le système distant un processus "receiver" chargé d'écouter les demandes de service en provenance du réseau.

Au niveau de l'utilisateur, le montage va simplement compléter l'arborescence du système de fichiers local en accolant l'arborescence du système distant. L'utilisateur n'a plus alors qu'à accéder aux fichiers distants comme il accède aux fichiers locaux.

Soit deux systèmes UNIX distincts reliés via le réseau local Ethernet. Chaque système a son propre système de fichiers. Notre objectif est de "monter" logiquement le système de fichiers d'un système distant sur un répertoire quelconque du système de fichiers du système local , de la même manière que l'on effectue l'opération de montage du système de fichiers d'un disque (avec la commande : mount).

Prenons un exemple concret qui va permettre d'illustrer les mécanismes engendrés par l'opération de montage de l'arborescence d'un système distant sur l'arborescence du système local, que nous baptisons "mounts".

Soient les deux systèmes suivants :

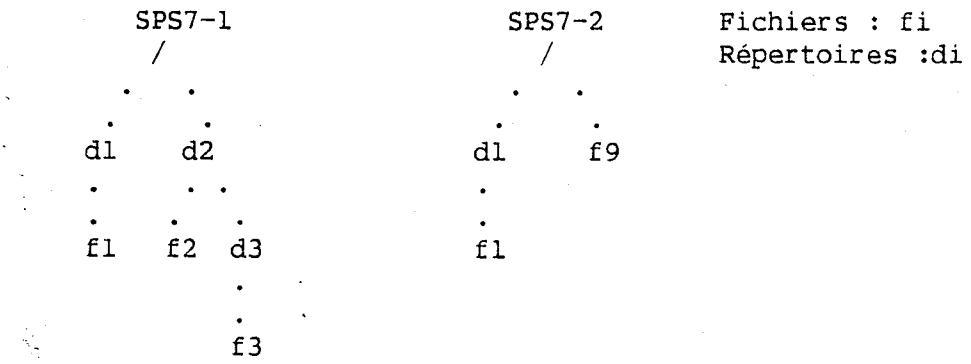


Figure III.1 : Systèmes initiaux.

Notre objectif est de monter SPS7-2 sur un répertoire "syst-d" du SPS7-1. Nous pouvons découper l'opération de montage en trois phases principales.

III.2.1. Création du répertoire.

Il s'agit de créer le répertoire "syst-d". Par exemple, nous aurons :

```
cd /d2/d3
mkdir syst-d
```

III.2.2. Montage de SPS7-2 sur syst-d.

Le montage s'effectue grâce à la commande mounts qui, à l'image du mount d'un disque, obéit à la syntaxe suivante :

- mounts..... rend la liste des systèmes distants déjà montés.
- mounts Syst Dir ..... montage du système de nom "Syst" sur le répertoire de nom "Dir".

Le "mounts" possède les mêmes caractéristiques que mount :

- Il ne peut être appelé que dans un mode privilégié nommé le super-utilisateur.

- Le fait de monter un système sur le répertoire "Dir" cache les fichiers qui s'y trouvaient précédemment (le passage sur Dir entraîne automatiquement un accès au système distant Sys).

- On ne peut monter un système sur la racine, ou sur un noeud de l'arbre ancêtre ou descendant d'un système déjà monté.

De même, le démontage d'un système distant se fait grâce à la commande :

- umounts Syst

autorisée en mode super-utilisateur, qui permet, si aucun processus n'utilise ce système, de le retirer de la table des systèmes montés.

Dans l'exemple précédent, l'opération souhaitée s'écrit donc :
mounts SPS7-2 /d2/d3/syst-d

La simple commande mounts permet alors de visualiser :
SPS7-2 on /d2/d3/syst-d

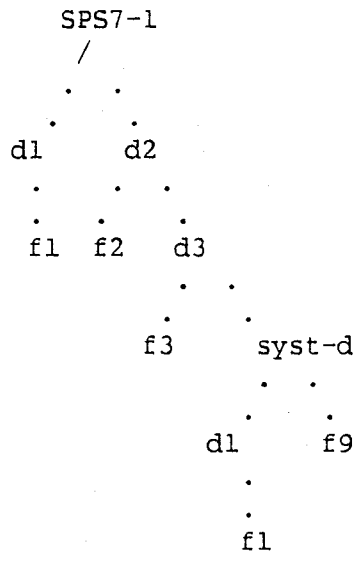


Figure III.2 : Arborescence logique après le montage.

Après la phase de montage, l'utilisateur sur le système local opère sur les fichiers distants comme si ils avaient un présence locale effective.

III.3. Droit d'accès au système distant.

Après la phase de montage, qui met en relation les deux systèmes, l'utilisateur ne peut accéder au système distant que s'il est connu (possède un identificateur d'utilisateur et un identificateur de groupe) sur ce dernier.

Nous verrons dans le chapitre IV.3 (Problèmes posés par le nom), que la solution la plus élégante consiste à mettre en relation un couple (nom1, mot de passe1) représentant l'utilisateur sur le système local à un couple (nom2, mot de passe2) le représentant sur le système distant.

Pour associer ces deux couples, on doit créer une commande spécifique, nommée ici rpasswd qui demande à l'utilisateur sous quel couple (nom2, mot de passe2) il compte travailler sur la machine distante. Le couple (nom2, mot de passe2) doit déjà être créé sur le système distant (avec la commande "passwd").

La syntaxe de la commande est donc :

- rpasswd Syst

où "Syst" est un système distant déjà monté.

Cette commande exécutée par l'utilisateur (nom1, mot de passe1) lui permet donc de s'associer au couple de travail (nom2, mot de passe2) sur le système distant.

Toutes les opérations futures se dérouleront donc :

- sur le système local, comme précédemment.

- sur le système distant, comme si l'utilisateur de ce dernier était défini par (nom2, mot de passe2).

III.4. Facilités offertes à l'utilisateur.

L'utilisateur peut donc opérer sur le système distant comme il opère sur le système local. On a ainsi une extension du traitement traditionnel en ce qui concerne les appels systèmes, les sous-programmes et les commandes. D'autre part, l'architecture que nous proposons permet également d'effectuer des exécutions distantes faisant des opérations d'entrées/sorties sur le système local de l'utilisateur.

III.4.1. Les appels système.

On peut scinder les appels systèmes (manuel 2 de SPIX) en deux catégories : ceux qui font référence à des fichiers, et les autres.

La seconde catégorie n'est pas modifiée par DISIX. Quant aux appels référençant des fichiers, DISIX apporte pour la plupart des extensions; les appels modifiés créent pour l'utilisateur de nouveaux messages d'erreur (errnos) dont la liste est donnée en annexe 1 et que l'utilisateur pourra tracer grâçe à la fonction PERROR de DISIX.

Rappelons qu'un accès à un fichier s'effectue de deux manières :

- Par une chaîne de caractères référençant l'arborescence du système de fichiers. Cette chaîne peut être absolue (écrite à partir de la racine "/"), ou bien relative par rapport au répertoire courant ("."). On adresse un fichier distant de la même manière, sachant que l'arborescence du système distant est montée sur le système local.

- Par un descripteur de fichier. Sur le système local, les descripteurs sont compris entre 0 et 63 sur système V, et entre 0 et 19 sur BSD 4.2.

III.4.2. Les sous-programmes.

L'utilisation des sous-programmes (manuel 3 de SPIX) est en tout point analogue à celle des appels système. Il est cependant à noter que la plupart des sous-programmes font référence à des "streams". Nous avons choisi de conserver exactement la même architecture pour les "streams" : 64 "streams" disponibles dans le système V, référençant chacun un fichier; 20 "streams" dans la version BSD 4.2..

Le traîtement au niveau utilisateur n'est donc pas modifié.

III.4.3. Les descripteurs de fichier.

Sur le système local, les descripteurs sont compris entre 0 et 63 sur système V, et entre 0 et 19 sur BSD 4.2. Pour les problèmes particuliers à Berkeley et pour conserver à l'application le maximum de portabilité, l'option suivante a été prise :

	31 - - - - - - - - - 8	7	6	5 4 3 2 1 0
	non utilisé	dist		local
4.2	"	64 à 127		0 à 19
S V	"	64 à 127		0 à 63

Descripteur de fichier DISIX.

DISIX fournit pour les fichiers distants des descripteurs compris entre 64 et 127, ces derniers ne constituant pas une suite cohérente. L'utilisateur les manipule cependant de la même manière qu'il manipule des descripteurs locaux.

III.4.4. Les sous-programmes.

III.4.5. Les commandes.

Les commandes (manuel 1 de SPIX), tel un programme utilisateur, utilisent les deux types d'appels précédents. Aussi, les commandes référençant des fichiers distants sont exécutées de manière habituelle, avec les appels DISIX. Ces commandes permettent donc de manipuler directement au terminal des fichiers distants.

III.4.6. L'exécution à distance.

Est qualifiée d'exécution à distance une exécution dont le programme binaire est situé sur le système distant. Dans une première version, nous autorisons l'utilisateur à faire des exécutions distantes tant que les entrées/sorties s'effectuent sur la machine locale (à l'utilisateur).

III.5. Moyens à mettre en oeuvre.

DISIX repose sur un réseau local Ethernet auquel sont connectées les différentes machines. On doit donc disposer d'un réseau Ethernet, ainsi que d'un "driver" transport : en effet, dans le cadre du projet, les machines utilisent la couche transport (couche ISO numéro 4) pour communiquer.

III.5.1. Le réseau local Ethernet.

Rappelons les principales caractéristiques d'un réseau local Ethernet [Cornafion 81].

Le réseau local Ethernet a été développé par la firme Xerox entre 1973 et 1975 pour les besoins internes de son centre de développement; la voie de communication ("Ether") est unique. Elle est constituée d'un câble coaxial partagé selon le principe de la diffusion avec réémission en cas de conflit. Elle permet à deux stations quelconques d'échanger des informations, découpées en paquets, avec un débit de 10 Mbit/s. Les caractéristiques du câble limitent sa longueur à 1 km environ, mais plusieurs câbles peuvent être reliés par des répétiteurs pour former un réseau arborescent.

III.5.2. La couche transport.

DISIX permet donc de mettre en relation deux systèmes distants, en utilisant la couche transport.

Cette couche a pour but d'optimiser l'utilisation des services de réseau disponibles. Elle assure un transfert de données transparent entre entités de session avec la responsabilité du contrôle de "bout-en-bout". La couche transport permet d'accéder aux protocoles de niveau inférieur des niveaux d'interconnexion des systèmes ouverts (OSI). Elle est composée de plusieurs classes qui correspondent à des niveaux de service différents. La classe est choisie à la connexion de transport. Nous choisirons la classe 4 qui offre le multiplexage et la meilleure fiabilité.

III.6. Compilation et édition de liens avec DISIX.

DISIX nécessite un remaniement des appels système ainsi que certains sous-programmes. Il faut donc que l'utilisateur précise si il désire travailler éventuellement à distance, avec la bibliothèque DISIX ou bien localement avec la bibliothèque standard.

La phase où ceci est précisé est bien sûr l'édition de liens avec la bibliothèque. Deux cas se présentent alors, suivant que l'utilisateur désire ou non faire une édition de liens séparée de la compilation, avec ses propres fichiers ou avec d'autres bibliothèques.

III.6.1. Sans édition de liens séparée.

Rappelons que la commande cc , sans l'option "-c" (édition de lien séparée), effectue la compilation ainsi que l'édition de liens avec la

bibliothèque standard d'appels système et de sous-programmes ("/lib/libc.a").

On a donc créé une commande ccd qui permet de faire l'édition de lien avec la bibliothèque DISIX ("/usr/lib/libc_d.a").
Cette commande a la même syntaxe que "cc". Si l'option "-c" est donnée, "ccd" a le même effet que "cc".

III.6.2. Avec édition de liens séparée.

Si l'utilisateur désire effectuer une édition de liens séparée, par exemple avec ses propres programmes, il peut compiler avec l'option "-c".

La phase d'édition de liens par la commande ld permet ensuite de mettre en relation les différentes tables symboliques décrivant les objets externes ou globaux des différents modules objets. L'option "-lx" , où x est une chaîne de caractères permettant de signaler à l'éditeur de liens qu'il doit lier les modules objets à la bibliothèque "/lib/libx.a" , ou à défaut /usr/lib/libx.a.
Ainsi, l'option "-lc" référence la bibliothèque standard; l'option "-lc_d" référence la bibliothèque DISIX.

Le choix est alors laissé à l'utilisateur. Nous reviendrons au chapitre VI sur la constitution d'une telle bibliothèque.

IV. ETUDE DE L'ARCHITECTURE.

Ce chapitre propose une architecture pour l'application DISIX dans le cadre d'un développement en mode utilisateur. IL présente l'aspect conceptuel des mécanismes de base de DISIX indépendamment des architectures matérielles et des interfaces d'accès aux services de communications.

IV.1. Choix d'une architecture.

L'article [Martin84] pose le problème du développement de logiciels réseaux sous UNIX. Il analyse les problèmes posés par l'implantation des couches correspondantes soit dans l'espace utilisateur, soit dans le noyau, et l'interface vis à vis des programmes utilisateurs.

Dans la plupart des versions d'UNIX, les processus ont des espaces d'adressage disjoints. Il s'ensuit que le développement d'un service qui nécessite échanges et multiplexages entre différents processus, ne peut se réaliser que sous la forme de procédures dans le noyau ou d'un processus serveur.

Dans le cas d'une implantation dans le noyau, la gestion des tampons permet l'accès des données à différents processus.

Dans le cas d'une implantation dans l'espace utilisateur, les données appartiennent au processus serveur et les transferts : soit du processus serveur vers les processus utilisateurs, soit des processus utilisateurs vers le serveur, doivent se faire par les mécanismes de communication entre processus (IPC).

Nous avons vu dans le chapitre précédent, les différents mécanismes existant (ou leur absence), l'inadéquation de certains, les performances dont on peut en attendre et les inévitables problèmes de portabilité posés par le choix d'un mécanisme particulier.

La conclusion logique pour le développement de notre couche de communication entre sites distants, serait de l'implanter dans le noyau sous la forme d'un "driver" particulier, tel que cela a été fait dans la plupart des systèmes UNIX distribués.

Cette solution résoud, en les supprimant, tous les problèmes d'IPC local et offre, par conséquent, un bon niveau de portabilité entre systèmes UNIX. En outre, l'intégration dans le noyau permet de résoudre les problèmes de protection, de performance et d'interface qui apparaissent dans les développements en espace utilisateur.

Cependant, nous rappelons que l'objectif de ce projet est d'étudier la faisabilité d'un système de gestion de fichiers développé entièrement en

espace utilisateur, en utilisant les services de communications présents sur une machine et d'en étudier les performances. De plus, dans le cas de la réalisation d'une maquette, il apparaissait difficile de modifier le noyau sans avoir l'entière disponibilité de la machine, ce qui n'était pas le cas.

Pour ces raisons, le développement a été effectué sous la forme d'un processus serveur qui assure l'interface local entre les processus utilisateurs et les services de communication.

IV.2. Introduction à l'architecture de DISIX.

IV.2.1. Généralités.

DISIX est une couche intermédiaire comprise entre le noyau et l'espace utilisateur (figure IV.1).

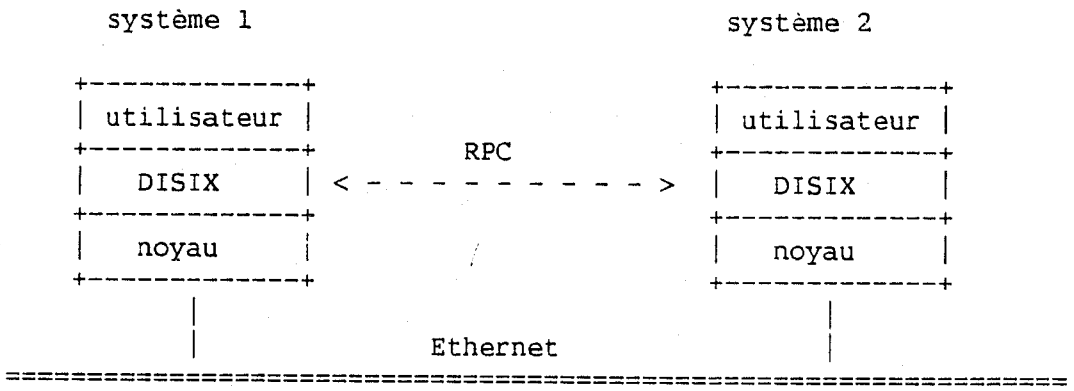


Figure IV.1 : Schéma général de l'architecture de DISIX.

Elle comprend un certain nombre de processus chargés d'effectuer, de manière transparente à l'utilisateur, des services qui peuvent être traités sur le système distant.

Les programmes qui communiquent à travers un réseau ont besoin d'une méthode de communication. La méthode utilisée par DISIX est le procédé des RPC (Remote Procedure Call), où un processus client communique avec un processus serveur (figure IV.2).

Dans ce procédé, le client appelle d'abord une procédure pour envoyer un message au serveur. Quand le message arrive, le serveur appelle un sous-programme d'acheminement, effectue le service demandé, renvoie la réponse et l'appel de procédure retourne au client.

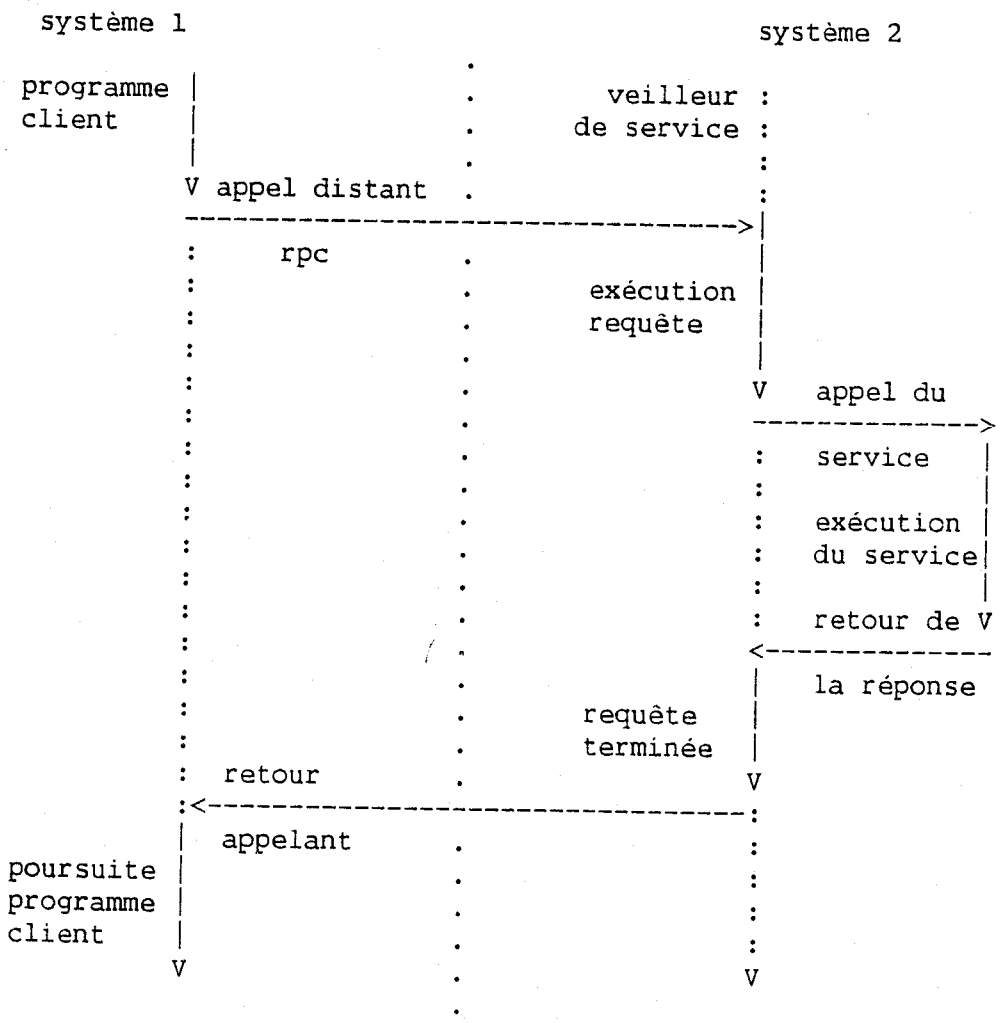


Figure IV.2 : Communication sur réseau avec RPC.

IV.2.2. Principe.

La couche DISIX intercepte un certain nombre d'appels système issus d'un processus utilisateur (ceux pouvant être l'objet d'un appel à une ressource distante) et teste s'ils désirent accéder à des ressources locales ou distantes. Si un appel système désire accéder à une ressource locale, alors le processus utilisateur effectue l'appel système local, sinon, si la ressource appartient à un système distant monté auparavant, DISIX émet un message vers le système distant concerné, qui créera un processus serveur chargé d'exécuter le service demandé par le processus client. Un utilisateur ne pourra donc accéder à un fichier distant que si le système auquel le fichier appartient est monté.

Considérons un appel système quelconque SYSCALL. DISIX va renommer l'ancien appel système "d_SYSCALL" et va réécrire SYSCALL de la manière suivante :

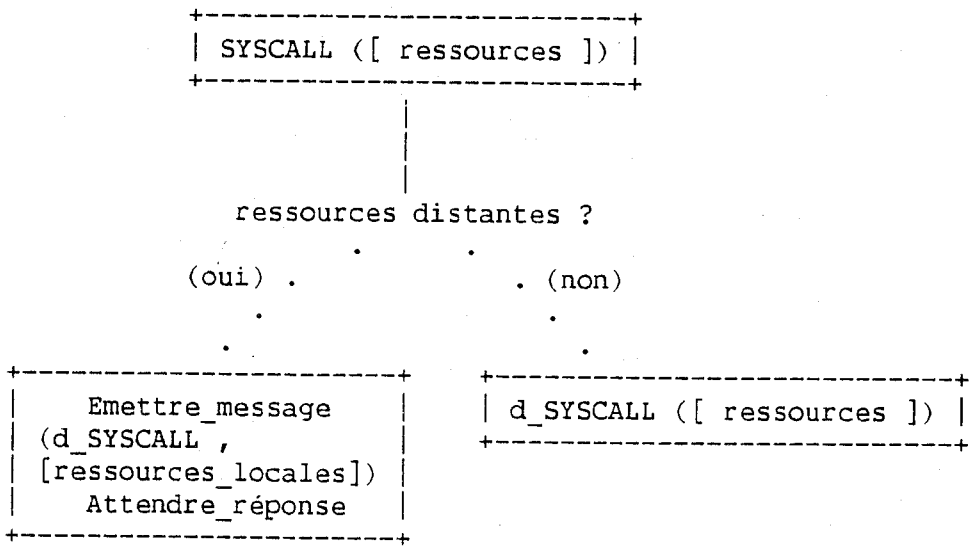


Figure IV.3 : Traitement d'un appel système.

- si les ressources arguments de SYSCALL sont locales, on effectue simplement le traitement avec l'ancien appel système.

- si les ressources sont distantes, on envoie au système distant un message chargeant un serveur d'effectuer le service sur la machine distante. En fait, le cheminement de la demande de service sera détaillé plus précisémment au IV.4 "Synoptique de l'architecture".

Ceci est le principe fondamental de DISIX. Nous allons maintenant détailler divers problèmes tels que le problème que nous a posé l'identification de l'utlisateur sur la machine distante; ensuite, nous proposerons un schéma général d'architecture inter-processus, et enfin nous détaillerons chaque processus.

IV.3. Problème posé par le nom.

Le système connaît le propriétaire d'un fichier grâce à deux identificateurs : l'identificateur de l'utilisateur et celui du groupe auquel appartient le propriétaire. Ces deux identificateurs sont des entiers non signés et non des chaînes de caractères.

Or quand nous nous connectons (login), nous fournissons notre nom sous la forme d'une chaîne de caractères. Grâce au fichier spécial "/etc/passwd", le système associera ce nom aux deux identificateurs. Un problème résulte de l'utilisation de ces deux identificateurs; comment le système génère-t-il ces deux nombres ?

Ces nombres sont créés au moment de l'introduction d'un utilisateur dans le système. L'administrateur choisira un numéro d'utilisateur libre et indiquera le numéro du groupe auquel appartient l'utilisateur.

Par conséquent, un couple d'identificateurs ne dépend que du système où il a été défini. Sur un autre système, soit il ne signifie rien (c'est-à-dire identificateur de l'utilisateur ou de groupe inconnu), soit il représente une autre personne complètement différente (ce qui est pire).

Ceci engendre un nouveau problème : une même personne peut-elle avoir deux noms différents sur deux machines différentes ?

On désire qu'un utilisateur puisse accéder aux fichiers des deux systèmes avec le même droit : celui d'être le propriétaire de ces fichiers.

La première solution consiste à ce qu'au moment de la demande de connexion par un processus client, le message de demande de connexion contienne aussi le nom (chaîne de caractères) du propriétaire du processus client.

Cette solution a 2 inconvénients :

- tous les utilisateurs doivent avoir le même nom sur toutes les machines auxquelles ils ont accès.

- cela n'empêche pas les homonymies (sur deux systèmes, deux utilisateurs possédant le même nom et un mot de passe différent).

La deuxième solution est d'envoyer en même temps que le nom, le mot de passe associé au moment de la demande de connexion. Cette solution résoud le problème des homonymies, mais pas le premier inconvénient.

Mais pourquoi envoyer le nom qu'un utilisateur a sur le système local et non celui sur le système distant ?

La troisième et dernière solution est donc d'ajouter au message de la demande de connexion du processus client, le nom et le mot de passe de l'utilisateur que celui-ci possède sur le système distant. Cette solution n'a plus les deux inconvénients décrits ci-dessus.

Comme nous l'avons dit dans la présentation d'UNIX (chapitre I), le mot de passe est crypté dans le fichier "/etc/passwd". On ne connaît pas l'algorithme permettant le décryptage d'un mot de passe, même en connaissant la clé . Par conséquent on sera obligé d'envoyer non pas le mot de passe en clair mais le mot de passe crypté avec la même clé que celui du système distant (c'est-à-dire que le mot de passe crypté émis et celui se trouvant dans le fichier "/etc/passwd" du système distant sont identiques).

C'est pourquoi, il est nécessaire de créer une commande presque équivalente à "passwd". Cette commande (appelée "rpasswd") est chargée de demander à l'utilisateur sous quel nom il désire travailler sur le système distant concerné. Pour cela il doit fournir le mot de passe associé à ce nom.

Cette solution a également le mérite d'apporter une certaine souplesse : on a la possibilité d'accéder aux fichiers distants sous n'importe quel nom (y compris celui du super-utilisateur qui est "root") à condition de connaître le mot de passe associé.

IV.4. Synoptique de l'architecture.

Nous avons vu (IV.2.2) que divers appels systèmes et sous-programmes du processus utilisateur sont interceptés et effectivement traités par DISIX s'ils référencent une ressource appartenant à un système distant monté.

IV.4.1. Etude des échanges entre deux systèmes définis.

Soit donc le système local et un système distant qui a été mis en relation, par la commande "mounts", au système local : Les échanges entre ces deux systèmes obéissent aux caractéristiques suivantes :

• Les appels système référençant des ressources appartenant à la machine distante considérée, émettent des messages vers le "sender", pro-

cessus de la machine locale. Il existe un "sender" unique par système distant monté (lancé par la commande "mounts").

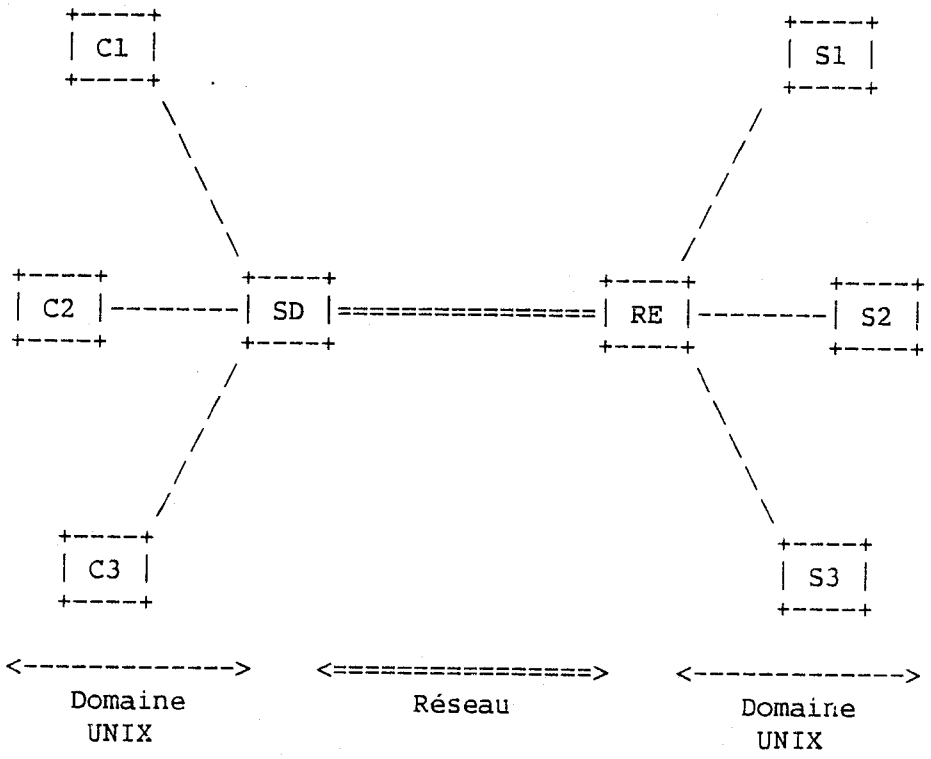
Les processus émetteurs d'un tel appel système sont appelés "processus clients", dans le sens où ils demandent un service à la machine distante.

• Le "sender" émet sur le réseau un message vers le "receiver" contenant des informations telles que l'identité du processus client, le service demandé...

• Le "receiver", selon le message, fait exécuter le service par un processus nommé "serveur" associé au processus client de la machine locale.

Lors de la première demande de service d'un processus client en direction de cette machine, un serveur associé est créé par le "receiver" sur le système distant. Lors des demandes suivantes, le serveur est nommément référencé par le processus client.

L'architecture générale est représentée sur la figure IV.4.



Avec Ci : Clients
SD : Sender
RE : Receiver
Si : Serveurs

Figure IV.4 : Echanges entre deux systèmes.

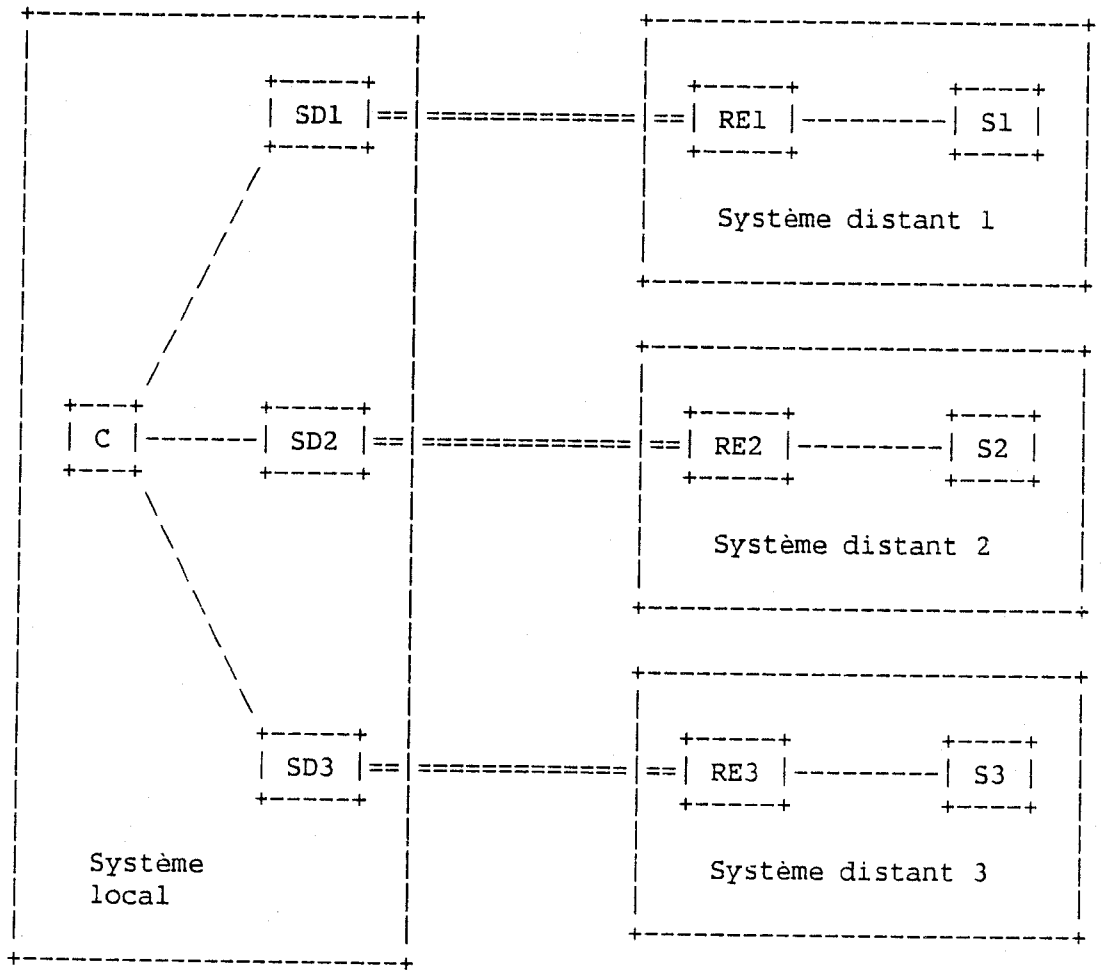
En retour de service, les résultats empruntent le chemin inverse : serveur, "receiver", "sender", puis processus client.

IV.4.2. Etude des échanges entre un processus et des systèmes distants.

Le paragraphe précédent relate comment deux systèmes sont mis en relation. Le phénomène est cependant plus complexe : un processus utilisateur unique peut demander divers services à plusieurs systèmes distants.

Dans ce cas, à la première demande de service du processus client sur une machine distante montée, un serveur associé est automatiquement lancé sur cette dernière.

A un processus client donné, il peut donc y avoir plusieurs serveurs associés sur différents systèmes distants (voir figure IV.5). Par contre, un serveur n'est associé qu'à un seul processus client.



Avec C : Client
SDi : Senders
REi : Receivers
Si : Serveurs

Figure IV.5 : Echanges entre processus client et systèmes distants

Ces différents processus seront détaillés dans le paragraphe IV.6 "Description des processus".

IV.5. Exécution distante.

Le "shell", prenant en compte une commande doit chercher où le fichier de cette commande se trouve par l'intermédaire de la variable d'environnement PATH. Il se peut donc que ce fichier se trouve sur un système distant monté. Le shell doit alors lancer l'exécution de cette commande sur le système distant concerné pour deux raisons :

- la première est d'éviter de transférer le binaire de cette commande sur la machine locale.

- la seconde est le fait que bien que les versions des systèmes UNIX peuvent être les mêmes, les machines sur lesquelles sont implantés ces systèmes peuvent être de types différents : SPS7, Vax ou MINI6. Donc le code binaire stocké sur une machine ne sera peut-être pas exécutable sur une autre.

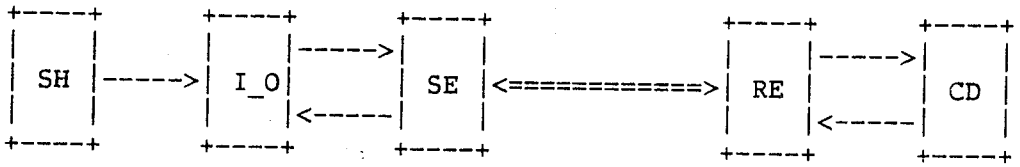
Un problème se pose :

Comment effectuer les entrées/sorties d'un processus s'exécutant à distance ? (dans le cadre de notre projet, nous ne considérerons que les entrées/sorties locales - sur la machine de l'utilisateur - redirigées ou non par le shell).

Il faut donc un processus serveur d'entrées/sorties (appelé "in_out") qui sera chargé d'effectuer localement les entrées/sorties standards du processus s'exécutant à distance.

Nous verrons que le processus "in_out" ressemblera beaucoup à un processus serveur et qu'il est lié à l'appel système "execve".

Puisque le processus "in_out" est sur la machine locale, il enverra les réponses des services de la commande en passant par le "sender". A l'opposé, le processus d'exécution distante enverra ses demandes de services d'entrées/sorties par l'intermédaire du "receiver" (figure IV.6).



SH : shell
I_O : in_out
SE : sender
RE : receiver
CD : commande distante

Figure IV.6 : Schéma des processus pour une exécution distante.

IV.6. Description des processus.

Ce chapitre va approfondir les fonctions des processus entrevus précédemment : "client", "sender", "receiver", "serveur", "In_out". Un processus nommé "spawner" est également introduit, bien que ne figurant pas sur les figures IV.4 et IV.5.

IV.6.1. Processus client.

Le processus client contient des appels système et sous-programmes pouvant avoir pour arguments des ressources distantes.

Ces appels système ont été remaniés par DISIX : s'ils doivent opérer sur une ressource distante, ils envoient un message en direction du "sender" associé au système distant possesseur de la ressource. Ce message contient :

- l'identification de l'appel système appelé.
- les arguments ne référençant pas de ressources distantes.
- les arguments de ressources distantes ayant été transformés de manière à être locaux au système distant considéré.

Ainsi, si le cheminement dans l'arborescence du système de fichiers local, passe par un répertoire sur lequel un système distant a été monté, le chemin aboutit à un fichier distant. La chaine de caractères représentant le fichier distant est ramenée à l'arborescence locale en substituant dans la chaîne de caractères initiale le préfixe qui mène jusqu'au répertoire, par la racine du système distant ("/").

Dans l'exemple du chapitre III.2, la référence à /d2/d3/syst-d/f1 génère un message portant sur le fichier local au système distant.

Les descripteurs, quant à eux, sont ramenés à la forme locale. De 64 à 127 chez le client, pour les fichiers distants, ils seront transformés à leur valeur réelle : 0 à 63, avant d'être introduits dans les messages à expédier.

Après avoir émis le message, l'appel système DISIX se met en attente d'une réponse venant du "sender". Il retourne ensuite au processus client.

Lors de la première demande d'accès d'un processus à une machine distante, on envoie au "sender" associé une requête de connexion. Ce dernier la transmet au processus "receiver" qui crée un serveur associé au processus client. Il est necessaire de passer alors au serveur plusieurs arguments :

i) le masque de création courant du processus client, afin que le serveur travaille sur ce masque. Ensuite, à chaque fois que le processus client changera son masque (par l'appel système "umask"), le serveur fera de même.

ii) la longueur des différents types de données : entier, entier court, entier long, signé ou non signé. Ceci est une caractéristique fondamentale de DISIX. En effet, deux machines différentes travaillant sous UNIX peuvent avoir des types codés différemment : par exemple, sur le SPS7 ou sur le Vax, le type "entier" est sur 4 octets; alors que sur le Mini6, il est sur 2 octets.

Ceci implique une conversion des données (en supposant qu'il n'y a pas débordement) entre deux machines sur lesquelles les types sont codés différemment. Nous avons opté pour que ce soit le processus serveur qui fasse régulièrement cette conversion. En effet, on peut supposer que dans la plupart des cas, le système distant sera plus important (et plus rapide) que le système local. C'est pourquoi on lui laissera faire ce traitement.

Le processus client, lors de la création du serveur, lui envoie donc les caractéristiques des types de données de sa machine locale.

IV.6.2. Spawner.

C'est un processus système, de type "veilleur", chargé d'écouter toutes les demandes de connexion émises par les "senders" appartenant à d'autres systèmes.

Il est initialisé par le processus "/etc/init" après le lancement initial du système.

Dès qu'une demande arrive, il crée le processus système "receiver" qui sera associé au "sender".

Le nom de "spawner" vient du fait que ce processus n'est pas seulement un veilleur (listener), mais qu'il fait également de la génération de processus.

Chaque système a un processus "spawner" et un seul.

IV.6.3. Sender.

C'est le processus système qui servira de liaison entre les processus clients et le système distant. Il est créé au moment du montage d'un système distant.

Son travail sera d'envoyer toutes les demandes de services des processus clients vers le "receiver" par l'intermédiaire de messages formatés selon le protocole défini (voir IV.7 "Protocole"). Il délivre les réponses des serveurs au bon processus client (multiplexage). Dans l'opération de connexion, il fournit au client le numéro de processus de son serveur, et conserve dans une table l'association processus client- processus serveur.

Les communications vers les processus clients se feront via les queues de messages de l'IPC système V, tandis que la communication vers le "receiver" se fera par l'intermédiaire de la couche transport.

Si l'administrateur décide de démonter le système distant alors le "sender" enverra un message particulier signifiant au "receiver" qu'il doit se tuer.

IV.6.4. Receiver.

C'est le processus système symétrique au processus "sender" sur le système distant (il est à remarquer qu'on aura toujours un couple "sender-receiver"). Il est chargé de créer un processus serveur par client.

Il vérifie tout d'abord que le propriétaire du processus client a le droit de se connecter sur la machine distante en comparant le nom et le mot de passe envoyés avec la demande de connexion par le processus client avec ceux se trouvant dans le fichier "/etc/passwd" du système distant (voir IV.3 "Problème posé par le nom"). Si cela coïncide alors le processus "receiver" crée le processus serveur associé au processus client, sinon il renvoie une déconnexion au "sender".

Le "receiver" en créant le serveur (execl) lui passera en arguments :

• les deux numéros d'identificateurs correspondant à celui du propriétaire du serveur. Par le nom du client reçu, il pourra lui associer l'identificateur de l'utilisateur et celui du groupe (setuid, setgid).

• l'identification de la queue de message pour l'IPC local.

• le numéro du processus client pour les réponses.

Le "receiver" aura les mêmes stratégies pour la synchronisation des traitements des réponses des serveurs que le "sender". Il retournera à tour de rôle vers le "sender" les réponses que lui délivrent les serveurs.

IV.6.5. Serveur.

Le serveur est chargé d'exécuter sur la machine distante toutes les demandes de service issues du processus client qui lui est associé.

Lors de la première référence du processus client à la machine distante, un serveur est créé par le "receiver" et associé à ce processus. Le serveur reçoit alors notamment (voir "Processus client") :

• le masque de création du processus client, qu'il va adopter.

• la longueur des types de données de la machine du client. A partir de ces longueurs, le serveur convertira les données qui proviennent du processus client, ou symétriquement qui iront à destination de ce dernier.

Ensuite, et jusqu'à la mort (ou une demande de déconnexion) du processus client, le serveur reste à l'écoute des demandes d'accès à des ressources distantes issues des appels système du processus client.

A chaque appel système, est associé un service distant. Le serveur, après avoir identifié l'appel système demandeur, lance ainsi le service distant associé.

Le retour du service distant est ensuite passé au "receiver", puis au "sender" et aboutit jusqu'à l'appel système demandeur.

Le service est en fait l'entité d'exécution associé à un appel système, de la même manière qu'un serveur est associé à un processus client ou qu'un "receiver" est associé à un "sender".

IV.6.6. In-out.

C'est le processus chargé d'exécuter les entrées/sorties d'un processus s'exécutant sur un système distant.

Il ressemblera beaucoup à un processus serveur sauf que ses services seront plus limités. Ses services seront ceux qui concernent les appels systèmes dont un des arguments est un descripteur de fichier. Et les numéros des descripteurs de fichiers seront uniquement ceux des entrées/sorties standards : 0, 1 et 2. Pour les autres, ils ne concernent pas le processus "in_out", car ils seront pris par la couche DISIX pour des descripteurs de fichiers locaux.

Ce processus est créé en général au moment de l'appel système "execve" sur un fichier distant. En particulier il sera créé par le shell si on demande une exécution distante.

Le processus "in_out" restera en vie jusqu'à la mort du processus distant associé.

IV.7. Protocole.

IV.7.1. Format des messages.

La couche DISIX est composée de deux sous-couches (figure IV.7). La première concerne les processus "sender" et "receiver". La seconde concerne les processus clients et serveurs.

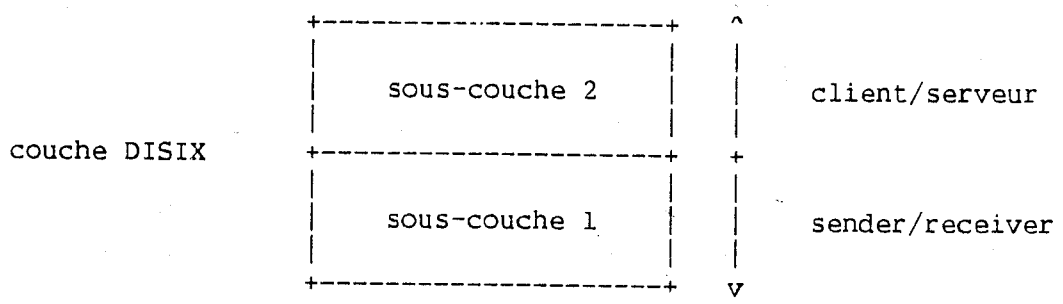


Figure IV.7 : décomposition de la couche DISIX.

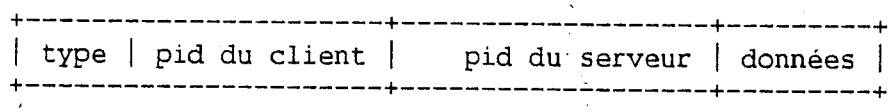
Dans ce paragraphe, nous ne parlerons que du protocole de la sous-couche 1. Le protocole de la sous-couche 2, format du champ "texte du message" [T], sera définie dans la partie implantation (partie V).

Nous avons défini cinq types de messages :

- demande de connexion du client (CR).
- confirmation de la demande de connexion (CC).

- refus de la demande de connexion et déconnexion du client (NC).
- transfert de données (D).
- requête de mort du "receiver" (KILL).
- déconnexion ou mort du client,
  ou mort du serveur détectée par le "receiver" (DISC).

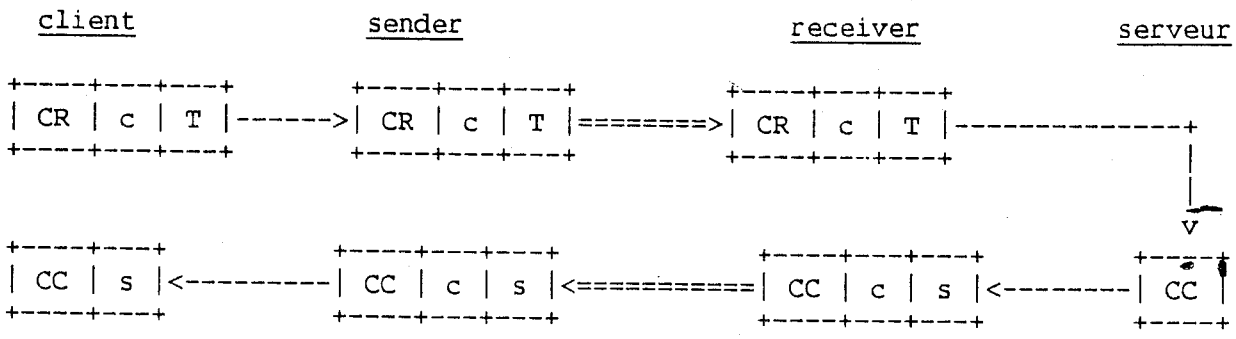
Voici le format général des messages :



pid : numéro du processus.
type : identification du message.
données : texte du message.

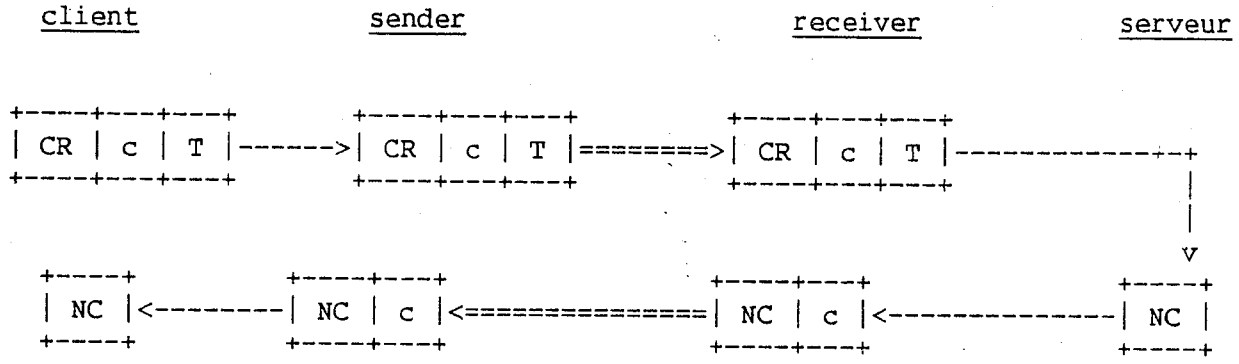
Les champs concernant les "pid" sont ajoutés par les processus clients et serveurs. Ils sont utilisés par le "sender" et le "receiver" pour poster les messages au destinataire final (voir le chapitre UNIX / IPC). Ces champs sont supprimés au moment de la livraison des messages aux clients et aux serveurs.

IV.7.2. Demande de connexion d'un processus client et confirmation.

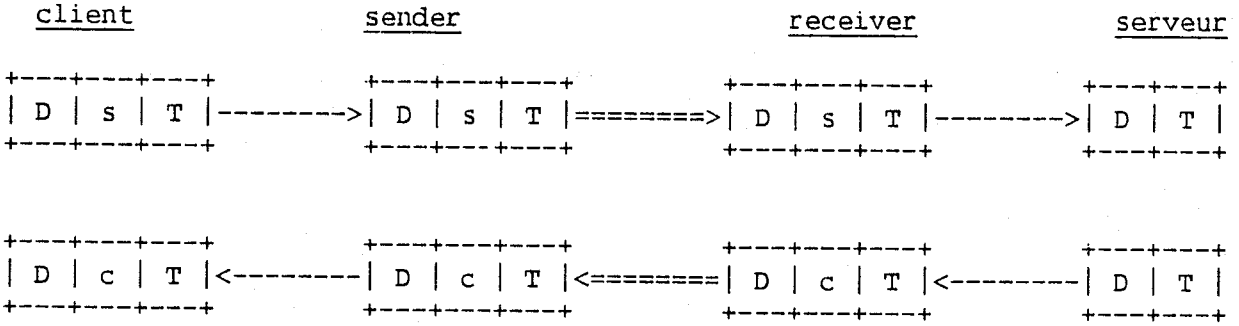


c : pid client
s : pid serveur

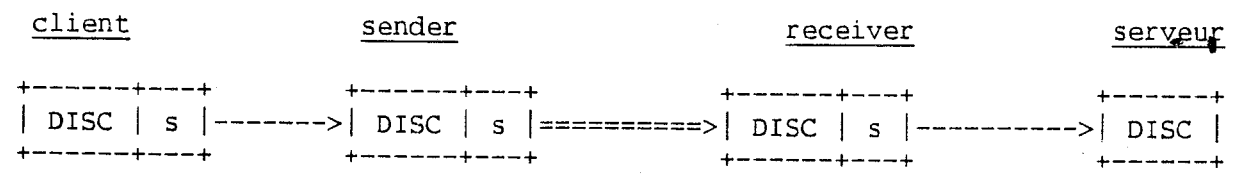
IV.7.3. Demande de connexion d'un processus client et refus.



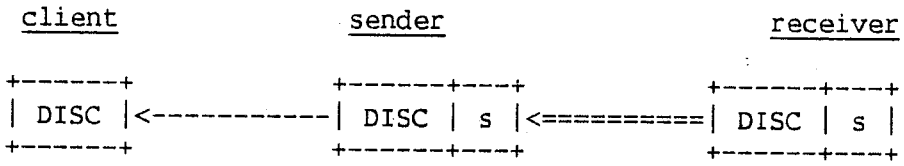
IV.7.4. Transfert de données.



IV.7.5. Déconnexion ou mort du client.

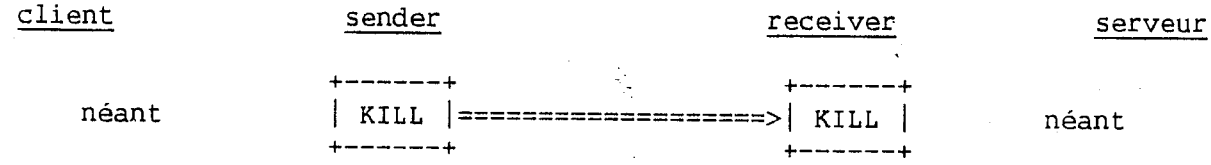


IV.7.6. Mort du serveur.



- Le "sender" sait retrouver le "c" correspondant à "s".

IV.7.7. Mort du receiver (démontage du système distant).



Les messages dont les types sont DISC et KILL n'ont pas le champ contenant les données ('T' pour Texte).

V. IMPLANTATION.

V.1. Les choix d'implantation.

V.1.1. Les critères de choix.

Le choix d'implantation ne peut pas se faire sans une étude préalable basée sur les objectifs que nous nous sommes fixés et sur les différents services de communications internes et externes offerts par le système d'exploitation SPIX et la Méthode d'Accès distribuée.

V.1.2. Les objectifs.

Ils ont principalement au nombre de 4 :

(1). avoir un système orienté connexion.

(2). s'adapter à la philosophie UNIX : système de gestion de fichiers, protection, création des processus.

(3). obtenir des performances acceptables étant donné les transferts de messages entre les différents systèmes de communication.

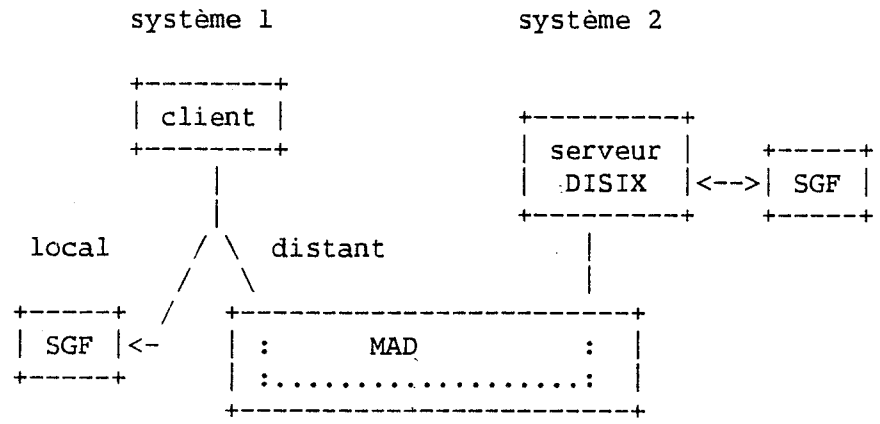
(4). réaliser des tests pour prouver la faisabilité et comparer les performances.

V.1.3. L'accès au réseau.

V.1.3.1. Méthode "client-serveur".

La possibilité nous était offerte d'utiliser le mode "client-serveur" développé sur le "driver" MAD et les couches ISO.

Dans cette solution, un serveur s'active vis-à-vis du "driver" MAD et se met en attente des demandes de services.



SGF : sytème de gestion de fichiers

Figure V.1 : Méthode "client-serveur"

Cette solution était simple à mettre en oeuvre, elle minimisait l'occupation des ressources et augmentait les performances globales du système. Mais elle a posé 2 problèmes :

- le système n'obéit pas à la philosophie UNIX (problèmes de "fork").

- il n'est pas envisageable sur système V car les primitives qui permettent de modifier l'identification d'utilisateur et de groupe (setuid, setgid), modifient en même temps l'identificateur réel et l'identificateur effectif. Ainsi, le serveur qui est super-utilisateur, ne pourra pas s'adapter temporairement, le temps de rendre le service, à l'identité du client. Car, s'il le faisait, il ne pourrait plus redevenir super-utilisateur. Cela nécessite, pour l'application de gérer ce qui est fait par le système; d'où complexité et perte de performance. A titre indicatif, les primitives de BSD 4.2 permettent de modifier l'identificateur effectif, sans modifier l'identificateur réel. Ce qui permet au super-utilisateur de changer d'identité temporairement.

Pour ces deux raisons, et aussi parce que ce système n'était pas complètement testé, nous avons préféré rechercher une autre solution.

Remarque :

Cette méthode semble être adaptée plus particulièrement à des traitements de type transactionnels avec un ensemble préétabli de ressources à gérer (serveur de fichiers). DISIX ne rentre pas dans ce type de traitement car il offre, en fait, la majeure partie du système d'exploitation comme ressource, et que d'un utilisateur à un autre, la demande peut être trés différente.

V.1.3.2. Méthode "client-serveur" plus "sender-receiver".

Cette solution, prolongement de la précédente, devait permettre grâce au "sender" et au "receiver" de résoudre les problèmes de transmission au serveur de l'identité du client.

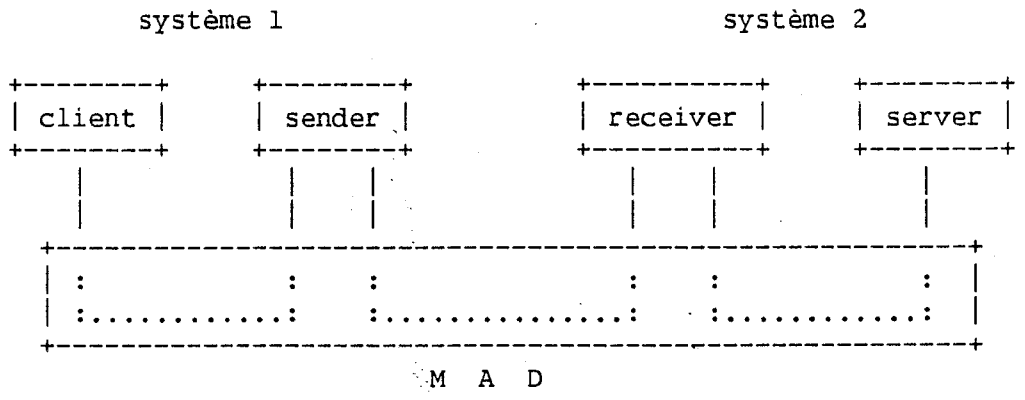


Figure V.2 : Introductions des "sender" et "receiver"

Dans cette solution, toutes les communications entre processus, qu'il s'agisse de la communication locale entre , par exemple, client "c" et "sender", ou sur le réseau entre le "sender" et le "receiver", se fait par la méthode "client-serveur".

Chaque processus devait être considéré à la fois comme client et comme serveur. Donc, chaque processus devait créer un accès au "driver" MAD pour émettre des messages et un accès sur lequel "s'activer" pour recevoir des messages. Or, du côté client, nous n'avons pas à limiter le nombre de processus qui veulent accéder au service DISIX. Nous nous sommes aperçus que les ressources du système (table des services, points d'accès au "driver" MAD), seraient rapidement saturées et que le système risquait de se dégrader.

Cette solution était complexe à mettre en oeuvre (gestion d'un "pool" d'UCB's pour assurer le multiplexage). De plus, si elle répondait au critère (2), elle ne correspondait pas au trois autres. En fait, ce qui nous génait le plus, c'était d'intercaler une interface utilisateur supplémentaire entre DISIX et l'accès au réseau.

V.1.3.3. IPC système V et Méthode "client-serveur".

Cette solution a été abordée, en temps qu'étape intermédiaire, pour évaluer les possibilités des mécanismes entre processus offert par le

système V. C'est ainsi que nous avons étudié une interface de communication entre les clients (serveurs) et le "sender" (receiver). Pour cette interface, les mécanismes des queues de messages et la mémoire partagée ont été étudiés. Celui des queues de messages nous a semblé offrir les garanties nécessaires de fiabilité, de performance et de facilité de mise en oeuvre.

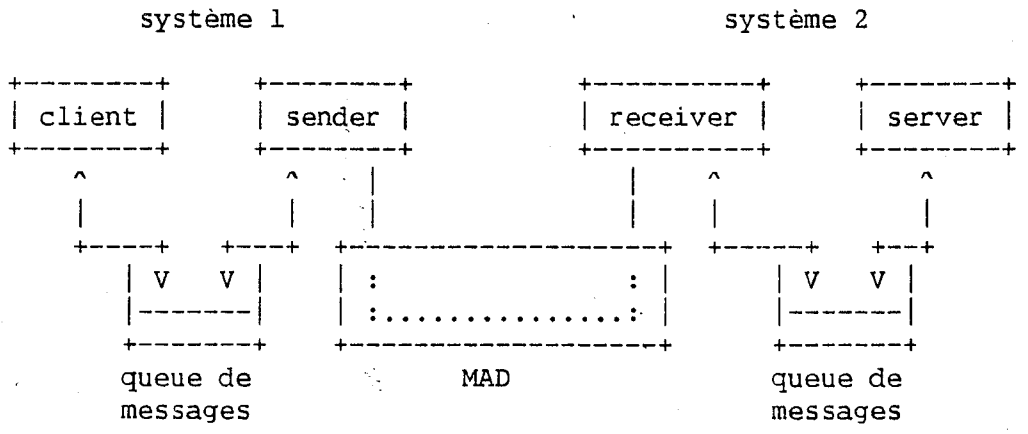


Figure V.3 : Introductions des queues de messages

Cette solution, si elle simplifie la communication entre processus dans le domaine local, présente les mêmes limitations que dans le cas précédent.

V.1.3.4. IPC système V et accès au transport.

C'est la solution qui a été adoptée et développée. Elle correspond aux quatre critères requis et à l'architecture définie au chapitre IV. Les processus "sender" et "receiver" utilisent le mécanisme de "requête-réponse" vis-à-vis du service transport. La communication locale utilise les queues de messages de l'IPC système V.

Cependant un problème subsiste :

Les processus "sender" et "receiver" sont en attente de messages en provenance du réseau et des processus clients ou serveurs. Il est donc nécessaire d'avoir un mécanisme d'attente multiple tel que le "select" de BSD 4.2. Cette primitive n'existe pas sur le système V.

Les messages provenant du réseau sont à traiter en priorité. Nous avons donc pensé à la possibilité de signaler à la fonction "attente sur le réseau" l' arrivée d'un message client ou serveur. Par exemple, le client émet un message vers la queue de messages et envoi un signal particulier au "sender"; celui-ci traite le message du client et se remet à l'écoute du

réseau.

Il y a, en fait, 2 problèmes :

• le mécanisme des signaux, qui fait que seul le dernier est enregitré (voir le chapitre UNIX).

• à la date du présent développement, les primitives MAD n'étaient pas interruptibles.

Pour régler ce problème d'attente multiple, nous avons dupliquer les processus "sender" et "receiver".

C'est ainsi que :

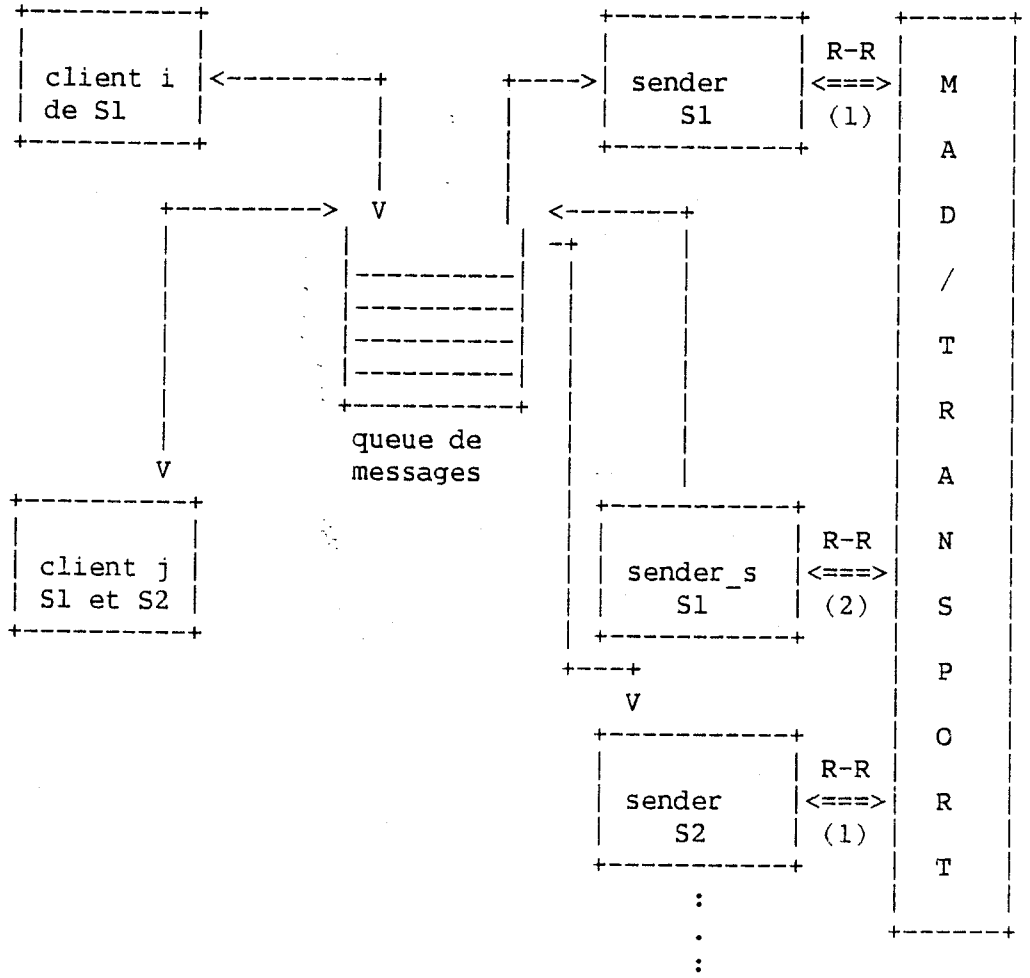
• "sender" est en attente de réception sur la queue de messages et émet sur le réseau.

• "sender_s" est en attente de réception sur le réseau et émet sur la queue de messages.

De même :

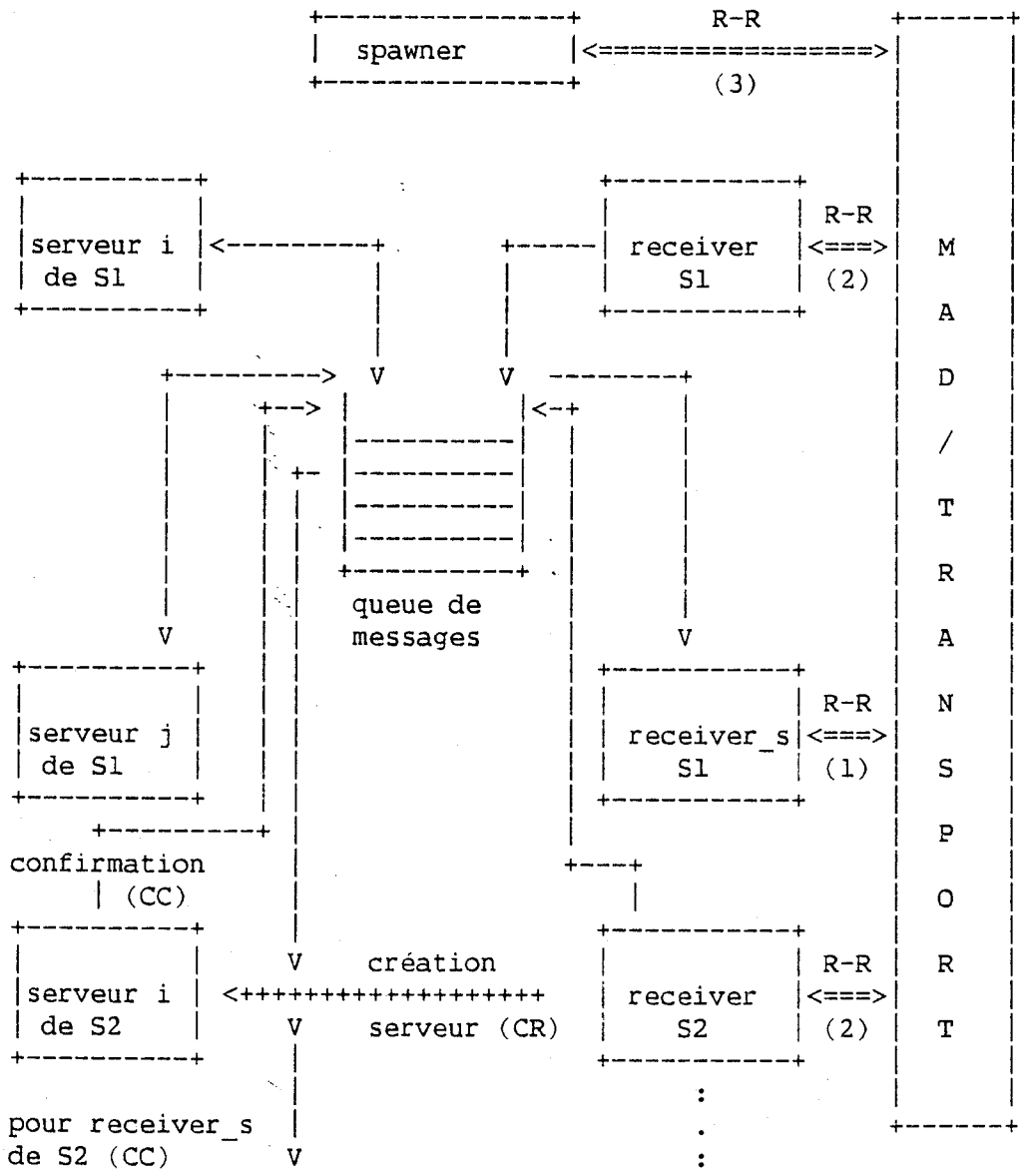
• "receiver" est en attente de réception sur le réseau et émet sur la queue de messages.

• "receiver_s" est en attente de réception sur la queue de messages et émet sur le réseau.



R-R : mécanisme Requête-Réponse vis-à-vis du transport
(1) : émision données normales
(2) : réception signaux et données normales

Figure V.4 : implantation côté client.



R-R : mécanisme Requête-Réponse vis-à-vis du transport
(1) : émision données normales
(2) : réception signaux et données normales
(3) : acceptation et confirmation (ou refus) de connexion

Figure V.5 : implantation côté serveur.

V.1.3.5. Accès au service session.

Le dernier choix concernant le développement a répondu au critère (4).

Pour pouvoir faire une première série de test sans tester l'accès à Ethernet, nous avons choisi d'accéder à la session qui permet de tester en rebouclage.

Ce choix ne remet pas en cause la programmation, mais seulement la réinitialisation des blocs de contrôle pour l'accès au transport.

V.1.4. Traitement de l'IPC local.

La même queue de messages est utilisée par plusieurs systèmes pour optimiser l'occupation mémoire. Par contre, si elle "casse", tous les systèmes deviennent inaccessibles.

Les processus qui émettent ou lisent sur la queue de messages utilisent la structure suivante :

```
struct ipc_msg {
        long type ;
        char texte [] ;
} ;
```

où :

- "type" est le numéro du processus destinataire.
- "texte" est un tampon contenant le texte du message proprement dit.

Ce mécanisme est expliqué dans le chapitre UNIX (utilisation pour DISIX).

Pour compléter ce mécanisme, nous avons simulé le système des "sockets" de BSD 4.2 qui fait que, si deux processus sont connectés et que l'un d'eux meurt, l'autre reçoit un message vide.

Ce système est utilisé par le "sender" pour signaler au client que son serveur est mort.

V.1.5. Le service session.

Le service session est situé sur un MT réseau qui fonctionne en environnement temps réel : SPART-UT. Le mécanisme de la méthode d'accès distribuée permet à l'application d'être localisée sur un MT différent de celui où sont implantés les services session ou transport.

Le service offert permet :

- l'ouverture de connexion de session.
- la fermeture normale ou anormale.
- l'envoi de messages normaux ou express.
- la réception de messages normaux ou express.
- la synchronisation.

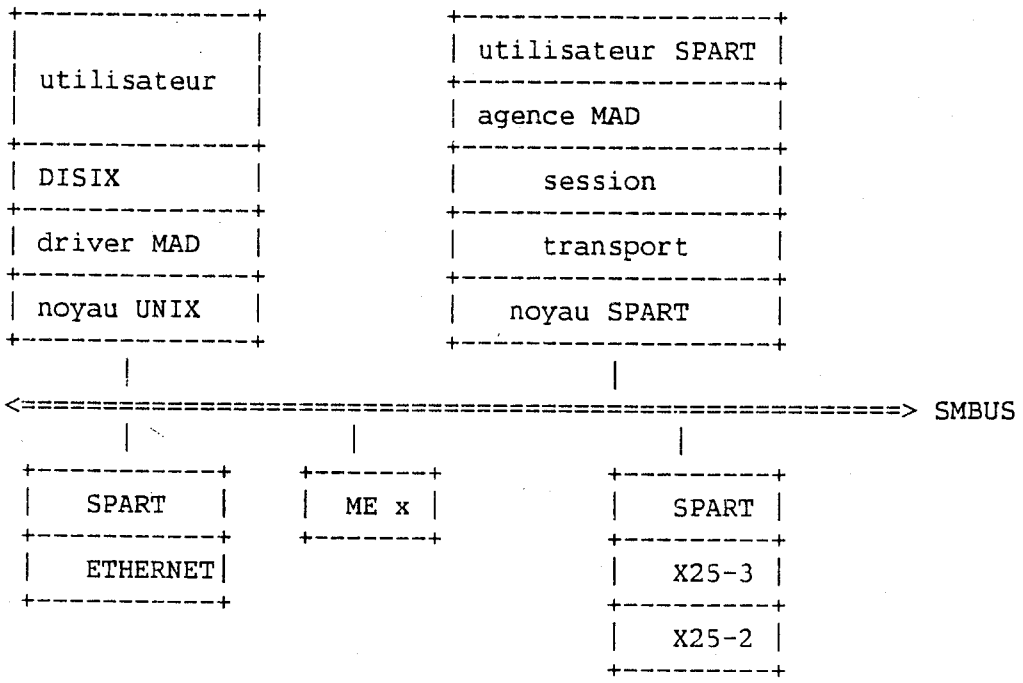


Figure V.6 : Configuration haut de gamme.

Le mécanisme requête-réponse est utilisé pour les communications entre des processus clients et les services de communication (session, transport). Le processus client pourra indiquer de manière explicite (UCB) ou implicite (le mineur du "driver" MAD indexe la table d'adressage) la localisation du serveur (session ou transport).

l'utilisateur du service session doit fournir dans la demande d'établissement d'une connexion de session, les éléments d'adressage nécessaires pour identifier les points d'accès locaux ou distants aux services session et à ceux inférieurs à la session.

Cette technique est applicable pour les accès aux services transport, réseau et liaison.

V.2. Les commandes DISIX.

Nous présentons les commandes qui ont été ajoutées et qui sont nécessaires pour pouvoir accéder aux fichiers distants (voir annexes).

V.2.1. Mounts.

C'est la commande qui permet le montage d'un système distant sur un répertoire du système local (voir III.2).

Les systèmes montés sont catalogués dans une table contenue dans le fichier "/etc/mtabs", accessible en lecture/écriture par le super-utilisateur et en lecture uniquement par les autres utilisateurs.

Cette table comporte des enregistrements constitués chacun de 3 champs.

Format de la structure de la table mtabs :

• pathname : nom du répertoire sur lequel le système distant est monté, écrit de manière unique et absolue (voir I).

• name : référence externe du système distant.

• sender_pid : numéro de processus du processus "sender" associé au système monté. Ce numéro servira d'identification du destinataire dans le mécanisme d'IPC, pour les émissions de la part des clients, et servira également au démontage du système distant.

De même, on a besoin d'une table contenant toutes les références des systèmes distants existants et accessibles. Cette table est stockée dans le fichier "/etc/site".

Dans le cas où la commande n'a pas d'argument, on affichera tous les systèmes montés et sur quel répertoire ils sont montés.

Avant de créer le processus "sender", il faut vérifier la validité des arguments :

- Le système distant existe-t-il ?

- Le système n'est-il pas déjà monté ?

- Le montage sur ce répertoire est-il autorisé ? La condition est qu'aucun répertoire ancêtre et aucun répertoire fils ne soient déjà utilisés pour le montage d'un autre système distant. De même, on interdit le montage sur la racine du système local.

Enfin, il faut négocier une connexion de session avec le "spawner" distant qui créera le processus "receiver".

Si toutes ces conditions sont vérifiées, alors "mounts" ajoute une entrée dans la table "mtabs" concernant le nouveau système monté, puis, il crée le processus "sender", en lui passant comme argument le nom du système distant et l'identification de la connexion de session.

Algorithme :

```
s'assurer de la validité de la commande ;
ouvrir un fichier spécial d'accès au "driver" MAD ;
initialiser l'SCB d'ouverture de la session ;
initialiser l'UCB d'ouverture de la session ;
requête d'ouverture de connexion session ;
réception ;
lire le compte rendu ;
créer un autre processus par la primitive "fork" ;
si
processus père
alors
    conserver le pid du fils ;
    mettre à jour mtabs ;
    envoyer message à l'opérateur ;
    fin;
sinon
    remplacer l'image mémoire par celle du "sender" avec en argument
    l'identification de la connexion session ;
fsi
```

V.2.2. Umounts.

Cette commande, exécutable seulement par le super-utilisateur, permet le démontage d'un système distant monté.

Le système ne sera démonté que si aucun processus local n'a de serveur sur le système distant concerné. Ce contrôle sera effectué en émettant un

message special au "receiver" par l'intermédiare de l'IPC local et du "sender". Le "receiver" compte le nombre de serveurs avec lesquels il communique : il les crée et il est avertit de leur mort. Le "receiver" repondra à "umounts" si le démontage peut avoir lieu ou non.

La commande "umounts" négociera l'abandon de la connexion session. La table "mtabs" permettra de récupérer l'identificateur de processus "pid" du "sender" concerné et le signal SIGINT sera envoyé au "sender" (kill (pid, SIGINT)), pour qu'il se tue.

Ensuite, on met à jour la table "mtabs" en enlevant l'entrée concernant ce système distant.

Cette mise à jour se fait par une copie du fichier "/etc/mtabs" sur un fichier temporaire "/etc/mtabsbis" en supprimant la structure associé au système distant. Après, le fichier "/etc/mtabs" est détruit, et on renomme le fichier "/etc/mtabsbis" par "/etc/mtabs".

V.2.3. Rpasswd.

Cette commande permet au processus client de pouvoir accéder à un système distant monté en demandant à l'utilisateur sous quel nom il désire travailler sur ce système.

Pour cela il devra fournir le mot de passe associé à ce nom. S'il est correct, le programme met dans le fichier ".rpasswd", qui est dans le répertoire de connexion (login) de l'utilisateur, le nom et le mot de passe crypté associé.

Il est à remarquer que si ce fichier est lisible par tous, seul le super-utilisateur peut le modifier pour des raisons évidente de sécurité car sinon l'utilisateur pourrait mettre par exemple le nom "root" et le mot de passe associé dans ce fichier et pourrait accéder aux fichiers distants en tant que super-utilisateur sans connaître en réalité le mot de passe en clair de "root".

Afin de pouvoir modifier le fichier ".rpasswd", la commande "rpasswd" sera exécutée avec le mode "set on user id". Ainsi, étant donné que le propriétaire du fichier est "root", tous les utilisateurs exécuteront cette commande en mode super-utilisateur.

Si cette commande n'a pas d'argument, elle affichera les noms des systèmes et les noms sous lesquels il pourra travailler sur les systèmes distants.

Le fichier ".rpasswd" est composé de structures de 3 champs.

Format de la structure associée à chaque système distant :

• system : référence externe du système.

• name : nom de l'utilisateur sur ce système.

• passwd : mot de passe crypté associé au nom de l'utilisateur sur le système distant.

Tout d'abord, la commande demande à l'utilisateur le nom d'utilisateur sur le système distant. Ensuite, le processus fait une demande de service pour avoir le mot de passe crypté associé au nom donné. Ce sera le seul service du processus.

Sur la machine distante, un service spécial nommé "PASSWORD" est implanté; le serveur correspondant au processus client "rpasswd" délivrera le mot de passe crypté demandé sans avoir envoyé les demandes de service "OPEN" et "READ" sur le fichier distant "/etc/passwd". Ce procédé réduit le nombre de messages et fait gagner du temps (pour le format voir le service PASSWORD).

Du fait qu'à ce moment-là, l'utilisateur n'a ni nom, ni mot de passe sur le système distant, stockés dans le fichier ".rpasswd", il ne pourra obtenir de serveur, car le "receiver" le refusera faute de mot de passe.

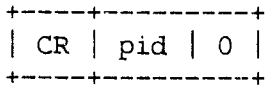
Comme le processus ne demande qu'un seul service, le "receiver" créera néanmoins le serveur en ajoutant un argument "-p" signifiant au serveur que son client est la commande "rpasswd".

Le "receiver" a le numéro d'utilisateur 0 (il est lancé par le système). N'ayant pas de numéro d'identificateur particulier à donner au serveur, il va lui attribuer son numéro ; le serveur sera donc en mode super-utilisateur.

Par l'argument "-p", le serveur initialisera dans le programme la variable "pflag" à 1. Avec la mise à la valeur 1 de cette variable, le serveur ne pourra exécuter qu'un type de service : le service PASSWORD, et aucun autre.

Il faut formater le message de demande de création du serveur d'une autre manière que celle décrite au chapitre III puisqu'il n'y a pas de nom, ni de mot de passe.

Format du message qui sera envoyé par la commande



A la place du nom et du mot de passe, on met le caractère nul.

Pour éviter d'avoir un code spécial pour la commande "rpasswd", on a ajouté une variable globale "rpasswdreq" qui normalement est nulle mais qui sera mise à 1 dans le programme de cette commande. Ainsi on enverra le message de demande de création du serveur, formaté comme ci-dessus et non comme habituellement.

Recevant le mot de passe crypté du serveur, le processus va le comparer avec le mot de passe entré par l'utilisateur au clavier et crypté avec la même clé que le mot de passe reçu du serveur. Si cela ne coïncide pas alors il répond à l'utilisateur "désolé" sinon il met à jour le fichier ".rpasswd" en ajoutant une nouvelle structure concernant le nouveau nom et le mot de passe associé.

Comme on ne peut avoir qu'un seul nom d'utilisateur par système distant, on détruira éventuellement l'ancienne structure contenant le même nom de système distant.

Il est à remarquer que cette commande ne sera effective que si le système distant concerné est monté à ce moment-là car elle nécessite une demande de service.

V.3. Les processus DISIX.

V.3.1. Spawner.

Créé à l'initialisation du système par le processus "init", le processus "spawner" a un travail très simple à effectuer : lire les demandes de connexion de la part des processus "senders" et créer en conséquence les processus "receiver".

Algorithme :

```
initialiser les UCB et SCB d'acceptation de connexion session ;
initialiser les UCB et SCB de confirmation de connexion session ;
cycle
    requête d'acceptation de connexion session ;
    réception ;
    lecture du compte rendu ;
    requête de confirmation (ou refus) de connexion session ;
    réception ;
    créer un autre processus par la primitive "fork" ;
    si processus père
    alors
```

```
            continuer ;
        sinon
            remplacer l'image mémoire par celle du "receiver"
            avec en argument l'identification de connexion session ;
        fsi
    fcycle
```

V.3.2. Client.

Le processus client lui-même n'est qu'utilisateur de la couche DISIX. Ce paragraphe décrit les services qui apparaissent au niveau du client. Ces services prennent les anciens noms des appels systèmes standards, ces derniers étant renommés avec le préfixe "d_".

Après édition de liens avec les appels systèmes DISIX, le processus utilisateur pourra référencer les machines distantes montées.

Par la suite, nous nommerons SYSCALL un appel DISIX, et d_SYSCALL l'ancien appel système standard.

V.3.2.1. Implantation des tables.

Ce paragraphe décrit les tables implantées et destinées à traiter les appels système.

La stratégie adoptée pour traiter un appel système repose sur l'implantation d'une table "systable" par processus, qui indique pour le processus client les connexions aux systèmes distants déjà établies.

Cette table contient 8 structures, chacune étant composée de 3 champs.

Format de la table des systèmes distants connectés au processus

- un champ sender-pid indique le numéro de processus du "sender", c'est-à-dire que ce champ identifie d'une manière unique le système distant.

- un champ server pid indique de manière unique le processus correspondant du client. Il est mis à jour lors de la réception de confirmation de connexion DISIX. Ce champ est initialisé à -1.

- un champ nb-linked indique le nombre de fichiers ouverts momentanément sur le système distant, pour le processus client.

V.3.2.2. Traitement de l'appel système SYSCALL.

Ce paragraphe traite de la manière dont DISIX, reconnaît qu'une ressource est distante ou non, et les paramètres qu'elle envoie dans le message au "sender".

i) Cas où la ressource est exprimée sous forme de "path".

Ecrivons le principe de l'algorithme de la procédure SYSCALL de DISIX.

Algorithme :

```
SYSCALL (..., path, ...)  /* "path" est une chaîne de caractères */
début
     rendre "path" unique et absolu (par rapport à la racine) ;
     parcourir "mtabs" afin de voir si un des répertoires sur lesquels
     sont montés les systèmes distants préfixe "path" ;
     si non trouvé
     alors
          appeler l'appel système standard d_SYSCALL et retourner sa
              valeur au processus client ;
     sinon
          regarder si un serveur est déjà créé sur la machine
              distante pour le processus client ;
          si il n'existe pas un tel serveur
          alors
               essayer de le créer; si impossible, message d'erreur
          fsi
          envoyer message de service au "sender" associé à la
              machine distante ;
          prendre en compte les résultats retournés par le serveur ;
          retourner le résultat ;
     fsi
fin
```

ii) Cas où la ressource est exprimée sous forme d'un descripteur de fichier.

Le système autorise simultanément un nombre maximal de 64 descripteurs de fichiers ouverts. Hors, ceux-ci sont codés sur 16, voire 32 bits (entiers) suivant les machines.

Nous pouvons donc utiliser certains bits afin de déterminer si le descripteur référence un fichier local ou distant.

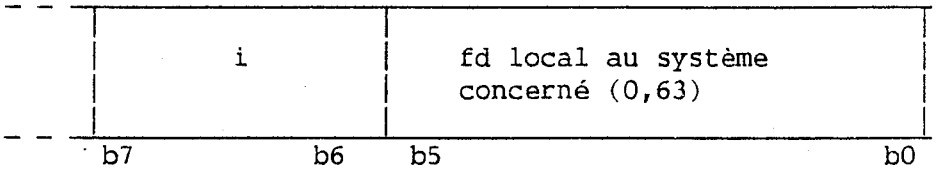


Figure V.7 : Codage d'un descripteur de fichier.

• si i = 0, le système possédant le fichier est le système local, et nous reprenons l'entier tel quel.

• si i <> 0, le système possédant le fichier est distant, et il représente l'index de la structure concernant ce fichier dans "systable". Nous remarquons ici que la ligne 0 de "systable" ne sera pas utilisée. On envoie alors un message au "sender" possédant pour arguments notamment le numéro de serveur (obtenu grâce à i) ainsi que les 5 bits de droite de l'entier qui représentent le descripteur de fichier sur le système distant.

Donc, moyennant une interception et un traitement sur les descripteurs de fichier, on peut directement accéder au système concerné.

V.3.2.3. Implantation des procédures.

Pour indiquer la démarche adoptée, nous allons introduire quelques procédures qui vont nous permettre de moduler le traitement. Nous indiquons leurs caractéristiques en langage C.

Soit un "path" : char *path0,

char *absolu (path0)

Retourne un "path" rendu unique et absolu à partir de path0. Ce "path" est calculé à partir du "path" du répertoire courant, que nous conservons dans un champ "CWD" de l'environnement.

struct remote-path *parcours mtabs (path0)

où struct remote-path {
local-path ;
site ;
sender-pid ;
} ;

Cette procédure parcourt la table "mtabs" des systèmes montés afin de trouver si il existe un répertoire lié à un système distant qui est un

préfixe de "path0". Dans le cas d'une recherche infructueuse, nous considèrons que la ressource est locale, et le pointeur retourne NIL. Dans le cas contraire, nous renvoyons :

- dans "local-path", le "path" rendu local pour le système distant ,sans le préfixe,
- dans "name-sys", le nom du système distant,
- dans "sender-pid", le numéro de processus du "sender" qui lui est associé.

int alloc-systb (dp)

où struct remote-path *dp ;

Cette procédure retourne l'index sur "systable" associé au système distant pointé par "dp". la procédure retourne -1 si la connexion au système distant ne peut s'établir.

La démarche théorique de cette procédure est donc la suivante :

```
début
     parcours de systable à la recherche d'un champ tel
         que sender-pid = dp->sender_pid ;
     si non trouvé
     alors
          parcours de systable à la recherche d'un enregistrement
              tel que nb_linked = -1 ;
          si non trouvé
          alors
               parcours de systable à la recherche d'un enregistrement
                   tel que nb_linked >= 0 ;
               si non trouvé
               alors
                    erreur (" table saturée ") ;
                    avorter ;
               sinon
                    "déconnecter" le système distant occupant cet
                        enregistrement ;
               fsi
          fsi
          "connecter" le système distant ;
          remplir le champ server_pid ;
          remplir le champ sender_pid par dp->sender_pid ;
     fsi
     retourner (index) ;
fin
```

L'écriture d'une telle procédure est bien sûr sujette à de nombreuses améliorations, l'algorithme précédent n'étant que théorique.

V.3.2.4. Traitement des sous-programmes.

Les sous-programmes utilisent les appels-systèmes, et pour la plupart simplement en utilisant les appels systèmes DISIX, les sous-programmes peuvent référencer des ressources distantes (même dans le cas des "streams", voir le chapitre VI pour avoir la liste des sous-programmes modifiés).

V.3.3. Sender.

Ce processus est créé par la commande "mounts" au moment de l'opération de montage. Il hérite de l'identification de la connexion session.

Son travail est de lire dans la queue de messages, les messages qui lui sont destinés et de les envoyer au "receiver" via la session.

Algorithme :

```
se protéger contre les signaux, sauf pour la correspondance
   avec "sender_s" et "umounts" ;
créer la queue de messages ;
ouvrir un fichier spécial d'accès au "driver" MAD ;
créer le "sender_s" avec pour arguments :
   le site distant,
   l'identification de la connexion session,
   l'identification de la queue de messages,
   le numéro de processus du "sender" ;
initialiser le SCB et l'UCB d'émission vers la session ;
cycle
   recevoir message sur la queue de messages ;
   si
   message vide
   alors
        avorter ;
   sinon
        requête émission d'UCB ;
        réception ;
   fsi
fcycle
```

Le "sender" est à l'écoute des erreurs graves sur réception au niveau du "sender_s", ou éventuellement de la mort de ce dernier.

V.3.4. Sender-s.

Ce processus est créé par la commande "mounts" par l'intermédiaire du "sender". Il est chargé de recevoir les réponses des services venant du réseau et de les délivrer au processus client concerné.

Il tient à jour une table de correspondance entre les numéros des processus clients et serveurs associés. Périodiquement (toutes les 2 minutes), il parcourt la table (polling) et envoi un signal :

- (kill (pid_cli, SIGKILL))

à destination du client pour s'assurer qu'il est toujours vivant et éviter ainsi une prolifération d'orphelins du coté serveur.

Dans TeamNET [Kavaler84], c'est le processus serveur lui-même qui fait ce travail.

Notre système apparait plus performant car il ne nécessite l'envoi d'un message réseau (et un seul), qu'uniquement dans le cas où le client

est mort.

TeamNET nécessite l'envoi d'un message pour s'assurer de la vie du client et d'un message pour la réponse.

Algorithme :

```
se protéger contre les signaux , sauf venant du "sender" ;
ouvrir la queue de messages ;
ouvrir un fichier spécial d'accès au "driver" MAD ;
initialiser le SCB et l'UCB de réception session ;
initialiser le SCB et l'UCB de lecture signal session ;
cycle
   requête lecture signal ;
   réception ;
      choix
      code retour

      NORMAL : requête lecture message normal;
               réception ;

      CLOSE ou ABANDON : avorter ;

      défaut : message d'erreur ;
               avorter ;
      fchoix
fcycle
```

Dans le cas de réception d'un message de déconnexion d'un serveur (DISC), "sender_s" retrouve le processus client et lui envoie un message vide.

Dans le cas de données normales (DATA), le texte du message est retiré de l'UCB pour être émis dans la queue de message.

En cas de confirmation de connexion (CC), la table de "polling" est mise à jour.

Enfin, pour les refus de connexion (NC), le message est émis tel quel dans la queue de message.

Le "sender" est à l'écoute des erreurs graves sur réception au niveau du "sender_s", ou éventuellement de la mort de ce dernier.

V.3.5. Receiver.

Ce processus est créé par le "spawner" si une demande de connexion de la part d'un système distant est acceptée. Il hérite de l'identification de la connexion session.

Il crée la queue de messages pour l'IPC local, et crée le processus "receiver_s" en lui fournissant l'identification de la connexion session, le nom du système distant et l'identification de la queue de messages.

Il est chargé de recevoir les messages venant du réseau / par l'intermédiaire de la session, et de les transmettre au serveur destinataire en émettant sur la queue de message.

S'il s'agit d'une demande de connexion de la part d'un client (CR), le "receiver" "fork". Le processus fils extrait du message le nom et le mot de passe. Il vérifie qu'ils concordent avec ceux du fichier "/etc/passwd". Si c'est positif, il crée le processus serveur en remplaçant son image mémoire par celle du serveur, en lui indiquant le numéro de processus du receiver_s, car c'est à ce dernier que le serveur enverra ses messages. Sinon il renvoie au client un message de refus de connexion (NC).

S'il apprend la mort d'un serveur, il enverra au "sender" un message (DISC) contenant le numéro du processus du serveur. C'est le "sender" qui retrouvera le client et l'avertira. Le "receiver" tient la comptabilité des processus serveurs qu'il crée.

Algorithme :

```
se protéger contre les signaux , sauf venant des fils ;
créer la queue de messages ;
ouvrir un fichier spécial d'accès au "driver" MAD ;
créer le "receiver_s" avec pour arguments :
       le site distant,
       l'identification de la connexion session,
       l'identification de la queue de messages,
       le numéro de processus du "receiver" ;
initialiser le SCB et l'UCB de réception session ;
initialiser le SCB et l'UCB de lecture signal session ;
nombre de serveurs := 0 ;
cycle
   requête lecture signal ;
   réception ;
      choix
      code retour

      NORMAL : requête lecture message normal;
```

```
                réception ;

        CLOSE ou ABANDON : avorter ;

        défaut : message d'erreur ;
                 avorter ;
        fchoix
    fcycle
```

Le "receiver" va traiter 5 types de message qui sont destinés aux serveurs :

CR, DISC, CF, DATA, SIGNAL,

et 1 message qui lui est destiné : KILL

Dans le cas de la demande de connexion (CR), il crée un serveur en lui indiquant toutes les informations necessaires à sa correspondance avec le client (numéro de processus du client) et avec le "receiver_s" (identification de la queue de message et numéro de processus du "receiver_s").

Dans le cas de données (DATA) et de "fork" du client (CF), il se contente de transmettre le message au serveur.

Dans le cas de deconnexion du client (DISC) il envoie un signal au serveur : kill (pid_ser, SIGKILL). C'est imparable, et la mort certaine du serveur.

Dans le cas d'un message (SIGNAL) indiquant à un processus s'exécutant localement, mais qui, en fait, est distant pour l'utilisateur, qu'une interruption a été générée sur l'entrée standard, le "receiver" exécute :

- kill (processus_distant, signal).

Le processus "distant" reçoit le signal et le traite comme il l'entend.

Enfin, le message "KILL" lui est destiné uniquement et lui indique qu'il va falloir abandonner la connexion suivant un protocole établi.

Le "receiver" est à l'écoute des erreurs graves sur émission au niveau du "receiver_s", ou éventuellement de la mort de ce dernier.

V.3.6. Receiver-s.

Ce processus est chargé de lire des messages sur la queue de messages dont il a hérité l'identification de la part du "receiver", et de les envoyer à la session; l'identification de la connexion de session ayant été

reçue de la même façon.

Algorithme :

```
se protéger contre les signaux, sauf du "receiver" ;
ouvrir la queue de messages ;
ouvrir un fichier spécial d'accès au "driver" MAD ;
initialiser le SCB et l'UCB d'émission vers la session ;
cycle
   recevoir message sur la queue de messages ;
   si
   message vide
   alors
        avorter ;
   sinon
        requête émission d'UCB ;
        réception ;
   fsi
fcycle
```

Si des problèmes surviennent au niveau de l'IPC ou de la session, le "receiver_s" avertit le "receiver" par signaux UNIX.

V.3.7. Serveur.

L'algorithme du serveur est très simple : il se contente d'attendre une demande de service, exécute le service distant associé, et renvoie ensuite la réponse de ce service au client.

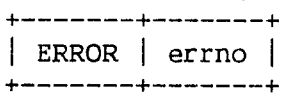
Algorithme :

```
recevoir le masque de création du client ;
recevoir la longueur des types de données de la
      machine locale à l'utilisateur (service "TYPES");
cycle
   attendre une demande de service ;
   vérifier que le service demandé existe ;
   l'exécuter ;
   renvoyer la réponse au client ;
fcycle
```

Tous les signaux attrapables sont associés à une fonction de traitement. Ainsi, le serveur enverra un message au client lui signifiant qu'il a reçu un signal (généralement, il y a une "trap").

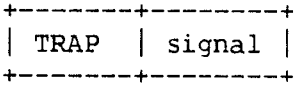
Par précaution supplémentaire pour éviter d'avoir des processus serveurs sans clients, si un serveur n'a pas reçu de demande de service au bout d'un certain temps limite (la valeur est relativement grande), le serveur se tue.

Format des messages dans le cas d'une erreur



Rappelons que la variable globale "errno" contient un numéro d'erreur. Les nouveaux "errno", ajoutés pour DISIX, sont donnés en annexe.

Format du message dans le cas d'un déroutement (trap)

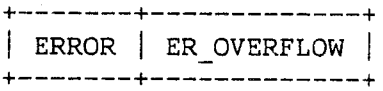


En cas de succès de l'exécution du service, le message est formaté selon le service demandé. Les formats seront décrits plus loin.

Rappelons que le serveur est chargé de convertir les données (en entrée ou en sortie) avec la taille correspondant au système du processus client.

Par exemple, si un entier est représenté sur 16 bits du côté client et 32 bits du côté serveur, le serveur travaillera bien sûr sur 32 bits, mais le nombre sera représenté sur 16 bits dans le message associé à la réponse.

Il se peut, toutefois, qu'il y ait débordement dans la conversion : "overflow". Dans ce cas, le message associé à la réponse aura le format :



La primitive retournera la valeur -1 au client.

V.3.8. In-out.

Le processus "in_out" aura la même structure qu'un serveur, mais il exécute moins de services qu'un serveur.

En outre c'est lui qui est chargé d'envoyer un message signifiant au processus s'exécutant à distance qu'il est prêt à recevoir les demandes de services.

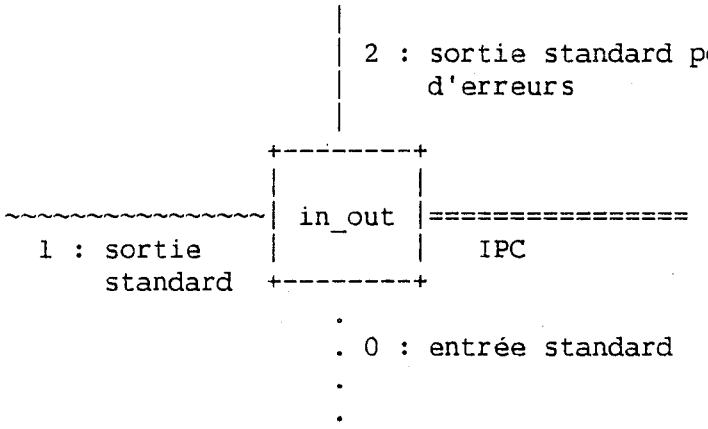


Figure V.8 : descripteurs du processus in-out.

Seules les entrées-sorties locales peuvent être redirigées. Les "redirections" distantes posent des problèmes plus importants qui nécessitent des extensions de DISIX.

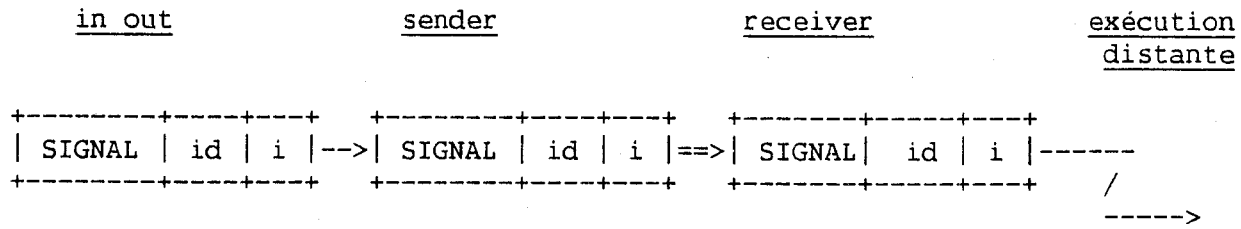
Un utilisateur, après avoir lancé une exécution distante, peut décider de l'arrêter prématurément en envoyant un "break" par exemple. Ce sera le processus "in_out" qui recevra le signal et non le processus distant.

Du fait que ces deux processus sont sur des systèmes différents, le processus "in_out" ne peut envoyer un signal au processus distant ("appel système "kill"). Comme la communication se fait uniquement à l'aide de messages, il va alors envoyer un message au processus distant lui signifiant la réception d'un signal.

Le "receiver" enverra lui-même le signal au processus distant, puisque c'est lui qui reçoit tous les messages et donc en particulier, le message signifiant que le processus "in_out" a reçu un signal de son entée standard.

Pour cela on va créer un autre type de message (SIGNAL) qui sera traité par le "receiver".

Représentation :



id : numéro du processus s'exécutant à distance
i : numéro du signal

Quand le "receiver" reçoit ce type de message, il extrait le signal "i" et le numéro du processus s'exécutant à distance "id".

Il pourra envoyer le signal par :

- kill (id, i).

Les services du processus "in_out" concernent les appels systèmes ayant pour paramètre un descripteur de fichier. Ils sont exécutés de la même façon que dans le serveur (cf Processus serveur dans la même partie).

Comme pour le serveur, nous avons mis un temps limite d'attente de demande de service (cette valeur est assez grande). Si au bout du temps limite il n'a rien reçu, alors le processus "in_out" se tue.

Liste des services : CLOSE, FCNTL, FSTAT, IOCTL, LSEEK, READ et WRITE.

Les appels systèmes OPEN, CREAT, DUP et DUP2 ne seront pas traités par "in_out" dans le projet. Seuls les descripteurs 0, 1 et 2 peuvent rester actifs.

V.4. Les appels systèmes et les services.

Nous allons décrire les formats des services correspondant au champ de données référencé par la lettre 'T'.

Toute chaîne de caractères se terminera par le caractère nul. Les descripteurs de fichier seront représentés sur un octet. Les autres paramètres seront représentés sur le même nombre d'octets que leur représentation en machine. Par exemple, sur le SPS7 ou le Vax 11/780 : un entier court est représenté sur 2 octets, un entier et un entier long sur 4 octets. Nous garderons le nom anglais des arguments des appels systèmes.

Si un appel système a comme argument une structure, pour des raisons d'implantation différente des structures d'une machine à une autre (tailles différentes pour les types, cadrage sur "mots" ou "demi-mots"), nous mettrons les champs un par un dans le message et non la structure entière.

Cependant, ceci ne résoudra quand même pas les problèmes liés à l'utilisation de la taille des structures dans des opérations d'entrées/sorties. En effet, la valeur de "sizeof (my_structure)" peut être différente d'une machine à l'autre.

La plupart des appels systèmes ne font l'objet d'aucun traitement particulier , pour d'autres nous détaillerons l'appel système.

Nous ne présenterons que les réponses à une demande de service où l'appel système distant s'est bien passé. Les autres cas ont été détaillés dans les paragraphes Processus concernant les clients et les serveurs.

V.4.1. ACCESS.

- numéro de service : 0
- format du service :

ACCESS	path	mode

- format de la réponse :

OK	accessible

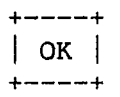
V.4.2. CHDIR.

On met à jour la variable d'environnement "CWD" si l'appel système s'est bien passé.

- numéro de service : 1
- format du service :

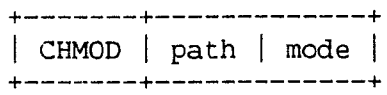
CHDIR	path

- format de la réponse :

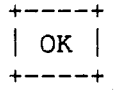


V.4.3. CHMOD.

- numéro de service : 2
- format du service :

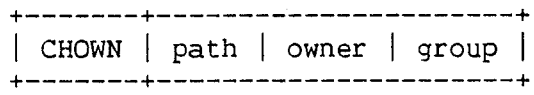


- format de la réponse :



V.4.4. CHOWN.

- numéro de service : 3
- format du service :

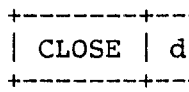


- format de la réponse :

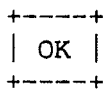
V.4.5. CLOSE.

- numéro de service : 4

- format du service :

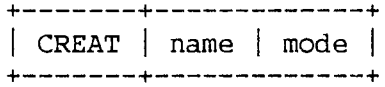


- format de la réponse :

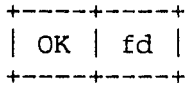


V.4.6. CREAT.

- numéro de service : 5
- format du service :



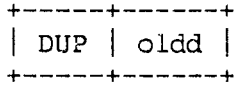
- format de la réponse :



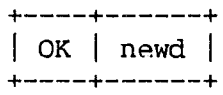
V.4.7. DUP.

Le nouveau descripteur aura toujours le même numéro d'index de système que l'ancien.

- numéro de service : 6
- format du service :



- format de la réponse :



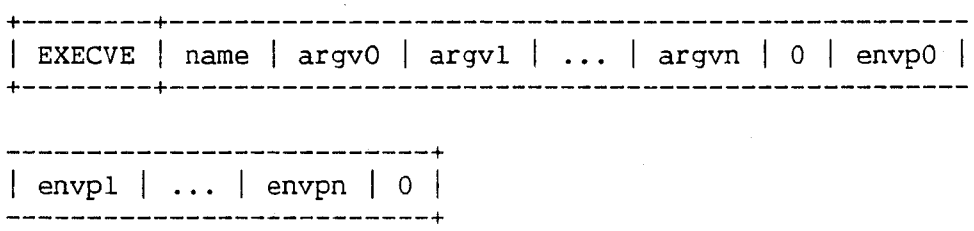
V.4.8. EXECVE.

Le processus client envoie le message formaté comme ci-dessus au serveur. Il attend la confirmation par l'exécution distante que l'appel système distant s'est bien passé. Ensuite il remplace son image mémoire par celle du serveur d'entrées-sorties "in_out".

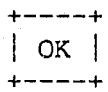
Si l'appel systéme distant s'est mal passé, la réponse sera envoyé par le serveur et l'appel système aura pour retour la valeur -1.

- numéro de service : 7

- format du service :



- format de la réponse :



V.4.9. FCNTL.

Le retour de l'appel système est soit un descripteur de fichier, soit un "flag". Il faudra faire un traitement selon l'argument "cmd" sur la réponse "res".

- numéro de service : 8

- format du service :

```
+--------+-----------------+
| FCNTL | fd | cmd | arg |
+--------+-----------------+
```

- format de la réponse :

```
+----+-----+
| OK | res |
+----+-----+
```

V.4.10. FORK.

Quand un processus client "fork", il faut aussi que tous ses serveurs "forks".

Le message contenant le service FORK est émis par le fils et parvient au processus serveur du père par l'intermédiaire du "sender" et du "receiver".

Pour réaliser cette opération un autre type de message, appelé CF, est créé. Il servira uniquement pour le service FORK.

Le serveur reçoit cette demande de service "fork", il "fork" à son tour et ce sera le serveur fils qui répondra à la demande de service.

- numéro de service : 9

- format du message envoyé par le client fils au sender :

```
+----+--------------------+
| CF | spp | FORK | cpf  |
+----+--------------------+
```

spp : identificateur de processus du serveur père
cpf : identificateur de processus du client fils

- format du message envoyé par le "sender" au "receiver" :

```
+----+--------------------+
| CF | spp | FORK | cpf  |
+----+--------------------+
```

spp : identificateur de processus du serveur père
cpf : identificateur de processus du client fils

- format du message envoyé par le "receiver" au serveur père :

```
+----+-------------+
| CF | FORK | cpf  |
+----+-------------+
```

cpf : identificateur de processus du client fils

- format du message envoyé par le serveur fils au "receiver" :

```
+----+------------+
| CC | cpf | spf  |
+----+------------+
```

cpf : identificateur de processus du client fils
spf : identificateur de processus du serveur fils

- format du message envoyé par le "receiver" au "sender" :

```
+----+------------+
| CC | cpf | spf  |
+----+------------+
```

cpf : identificateur de processus du client fils
spf : identificateur de processus du serveur fils

- format du message envoyé par le "sender" au client fils :

```
+----+-----+
| CC | spf |
+----+-----+
```

spf : identificateur de processus du serveur fils

V.4.11. FSTAT.

- numéro de service : 10
- format du service :

```
+-------+----+
| FSTAT | fd |
+-------+----+
```

- format de la réponse :

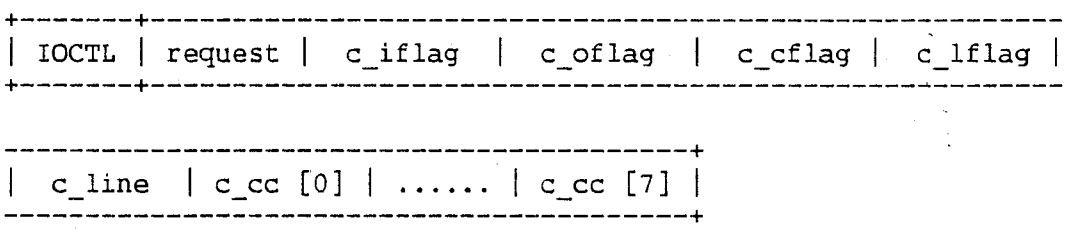
```
+----+-------------------------------------------------------------+
| OK | st_mode | st_ino | st_dev | st_rdev | st_nlink | st_uid |
+----+-------------------------------------------------------------+

-----------------------------------------------------+
| st_gid | st_size | st_atime | st_mtime | st_ctime |
-----------------------------------------------------+
```

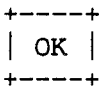
V.4.12. IOCTL.

Le format du message dépendra de la requête "request". Selon la requête, les champs des structures : termio, termcb, varterm, seront ou non dans le message de demande de service ou de réponse.

- numéro de service : 11
- format du service :

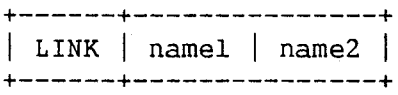


- format de la réponse :



V.4.13. LINK.

- numéro de service : 12
- format du service :

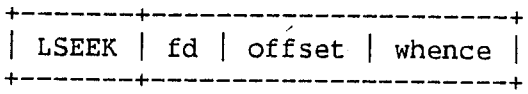


- format de la réponse :

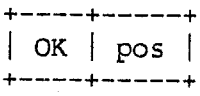
V.4.14. LSEEK.

- numéro de service : 13

- format du service :

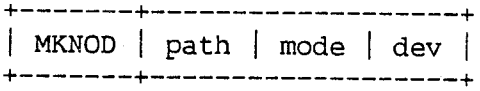


- format de la réponse :

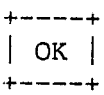


V.4.15. MKNOD.

- numéro de service : 14
- format du service :

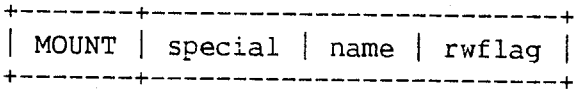


- format de la réponse :

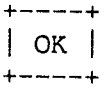


V.4.16. MOUNT.

- numéro de service : 15
- format du service :



- format de la réponse :



V.4.17. OPEN.

- numéro de service : 16
- format du service :

```
+------+--------------------+
| OPEN | path | flags | mode |
+------+--------------------+
```

- format de la réponse :

```
+----+----+
| OK | fd |
+----+----+
```

V.4.18. PASSWORD.

Ce service n'est pas associé à un appel système, il a été ajouté pour éviter de faire plusieurs demandes de services au lieu d'un.

Normalement seul le programme "/etc/rpasswd" utilisera le service PASSWORD.

- numéro de service : 17
- format du service :

```
+----------+------+
| PASSWORD | name |
+----------+------+
```

- format de la réponse :

```
+----+--------+
| OK | passwd |
+----+--------+
```

V.4.19. READ.

- numéro de service : 18

- format du service :

```
+-----------+--------+
| READ | fd | nbytes |
+-----------+--------+
```

- format de la réponse :

```
+----+-----------------+
| OK | cc | characters |
+----+-----------------+
```

V.4.20. STAT.

- numéro de service : 19
- format du service :

```
+------+------+
| STAT | path |
+------+------+
```

- format de la réponse :

```
+----+-------------------------------------------------------------
| OK | st_mode | st_ino | st_dev | st_rdev | st_nlink | st_uid |
+----+-------------------------------------------------------------

-----------------------------------------------------+
| st_gid | st_size | st_atime | st_mtime | st_ctime |
-----------------------------------------------------+
```

V.4.21. TYPES.

Ce service n'est pas associé à un appel système. Il a été ajouté pour des raisons d'hétérogénéité des machines. Les types ne sont pas représentés de la même façon sur les différents machines. Au moment de la connexion sur le système distant le processus client envoie les tailles des types : entier court, entier et entier long au processus serveur.

Ce sera toujours le serveur qui s'adaptera au client, c'est-à-dire que même si la taille d'un type de sa machine n'est pas la même que celle de la machine du client, il devra s'arranger pour représenter le nombre sur la bonne taille. Bien sûr, il se peut que cela soit impossible, alors le retour de l'appel système sera -1 (le message de la réponse du serveur sera

ERROR).

- numéro de service : 20
- format du service :

TYPES	short	int	long

- format de la réponse :

OK

V.4.22. UMASK.

- numéro de service : 21
- format du service :

UMASK	numask

- format de la réponse :

OK	oumask

V.4.23. UMOUNT.

- numéro de service : 22
- format du service :

UMOUNT	special

- format de la réponse :

```
+----+
| OK |
+----+
```

V.4.24. UNLINK.

- numéro de service : 23
- format du service :

```
+--------+------+
| UNLINK | path |
+--------+------+
```

- format de la réponse :

```
+----+
| OK |
+----+
```

V.4.25. ULIMIT.

Le retour de l'appel système est soit la limite en taille du processus, soit la valeur maximum d'augmentation dynamique du segment de données. Il faudra faire un traitement particulier suivant l'argument "cmd".

- numéro de service : 24
- format du service :

```
+--------+-----------------+
| ULIMIT | cmd | newlimit  |
+--------+-----------------+
```

- format de la réponse :

```
+----+-----+
| OK | res |
+----+-----+
```

V.4.26. UTIME.

- numéro de service : 25
- format du service :

UTIME	actime	modtime

- format de la réponse :

OK

V.4.27. WRITE.

- numéro de service : 26
- format du service :

WRITE	fd	nbytes	characters

- format de la réponse :

OK	cc

VI. INSTALLATION.

VI.1. Hiérarchie du système de fichiers.

```
/etc/
     server
     mtabs
     mounts
     umounts
     site
/usr/
     bin/
          ccd
          rpasswd
     lib/
          libc_d.a
          libc_dk.a
     ipc/
          clients/
          servers/
     src/
          disix/
               client/
                    c/
                         Makefile
                         access.c
                         chdir.c
                         chmod.c
                         chown.c
                         close.c
                         common_part.c
                         creat.c
                         dup.c
                         dup2.c
                         execve.c
                         fcntl.c
                         fork.c
                         fstat.c
                         ioctl.c
                         link.c
                         lseek.c
                         mknod.c
                         mount.c
                         open.c
                         perror.c
                         read.c
```

```
                stat.c
                umask.c
                umount.c
                unionlib.c
                unlink.c
                utime.c
                write.c
        cmd/
                Makefile
                ccd.c
                sh/
                        Makefile
                        sh.c
                        *.c
        h/
                client.h
                disix.h
                disix_config.h
                disix_err.h
                disix_mad.h
                in_out.h
                server.h
                var.h
        libc/
                sys/
                        makefile
                stdio/
                        Makefile
                gen/
                        Makefile
                        getlogin.c
                        getpass.c
                        popen.c
                        psignal.c
                        syslog.c
                        ttyslot.c
                        ttyname.c
        mounts/
                Makefile
                conn_mad.c
                mounts.c
                receiver.c
                receiver_s.c
                recv_mad.c
                rpasswd.c
                send_mad.c
                sender.c
                sender_s.c
                sendrecv.c
```

```
        spawner.c
        umounts.c
    server/
        c/
            Makefile
            in_out.c
            s_access.c
            s_chdir.c
            s_chmod.c
            s_chown.c
            s_close.c
            s_creat.c
            s_dup.c
            s_execve.c
            s_fcntl.c
            s_fork.c
            s_fstat.c
            s_link.c
            s_lseek.c
            s_mknod.c
            s_mount.c
            s_open.c
            s_password.c
            s_read.c
            s_stat.c
            s_types.c
            s_umask.c
            s_umount.c
            s_unlink.c
            s_utime.c
            s_write.c
            server.c
```

VI.2. Implantation des bibliothèques.

Les différents services correspondants aux appels système sur la machine locale, doivent être archivés, de manière à ce qu'on puisse faire l'édition de liens entre un programme utilisateur et cette bibliothèque.

La reprogrammation des appels systèmes impose le renommage des appels systèmes standards.

VI.2.1. Anciens appels systèmes.

Ils sont renommés en rajoutant le préfixe "d_" au nom standard (par exemple, "open.o" devient "d_open.o"), et sont archivés dans la bibliothèque "libc_dk.a". Ces appels systèmes ne doivent pas être utilisés directement dans les programmes utilisateurs.

La commande permettant d'archiver les anciens appels systèmes est "make". Le fichier "makefile" est dans le répertoire "/usr/src/disix/libc/sys". La bibliothèque "/usr/lib/libc_dk.a" est créée automatiquement.

VI.2.2. Nouveaux appels systèmes.

Ils seront archivés dans la bibliothèque "libc_d.a".

En outre, dans la bibliothèque "libc_d.a", se trouve le binaire associé au fichier source "common_part.c" qui contient toutes les fonctions communes aux appels DISIX.

Toutes ces procédures ont été compilées avec l'option "-c" (édition de liens séparée), leur édition de liens ne se faisant qu'au moment de l'édition avec le programme utilisateur. Les tables symboliques associées aux programmes objets contiennent donc avec le statut "inconnu" (U) :

- les appels standards (préfixés par "d_").

- les appels de procédures du fichier "common_part", sauf pour le fichier lui-même.

Pour éviter au programmeur d'utiliser deux bibliothèques ("libc_dk.a" et "libc_d.a"), tous les anciens appels systèmes ont été enregistrés dans le fichier "unionlib.c" dont le binaire sera archivé dans la bibliothèque "libc_d.a".

Ce fichier est édité partiellement (option "-r") avec la bibliothèque "libc_dk.a" des anciens appels système. Il y aura donc dans la bibliothèque "libc_d.a" une table symbolique et une seule qui connaisse les appels préfixé par "d_". L'édition de lien finale résoudra donc toutes les références inconnues.

VI.2.3. Sous-programmes modifiés de la section 3 du manuel UNIX.

Nous avons préféré, pour des raisons d'efficacité et de cohérence, garder les appels systémes standards plutôt que les appels systèmes DISIX.

Cela concerne les sous-programmes : getlogin, getpass, perror, popen,

psignal, syslog, ttyslot et ttyname.

Pour BSD 4.2 uniquement, le code du sous-programme "fdopen" qui utilise le sous-programme "getdtablesize" a été modifié. Compte tenu du fait qu'il peut y avoir des descripteurs de fichier supérieur à 19 et que "getdtablesize" retourne la valeur : 20. Le test comporte, en plus, une comparaison sur les "bits" correspondants aux systèmes distants.

VI.3. Compilation ccd.

Du fait de la présence d'une nouvelle bibliothèque, il faut modifier le source du programme cc.c au niveau du paramètre de l'édition de liens. Au lieu de mettre l'argument "-lc", on remplace par "-lc_d".

Exemple : ld /lib/crt0.o -o pgm pgm.o -lc_d

Le nouveau programme de compilation s'appelle "ccd.c".

VI.4. Commandes et programmes utilisateurs.

Pour pouvoir manipuler des fichiers distants, il est nécessaire de refaire l'édition de liens avec la bibliothèque "libc_d.a".

VII. CONCLUSION.

Ce projet m'a permis de faire une synthèse des problèmes d'accès à des fichiers distribués, ainsi que de mettre en oeuvre les mécanismes de communication, (à la fois communication interprocessus et communication réseau), dans les versions les plus récentes d'UNIX, le sytème V et BSD 4.2.

Il resterait à faire des mesures de performances, afin d'évaluer si l'accès à un fichier distant via le réseau Ethernet est, ou n'est pas, sensible pour l'utilisateur par rapport au temps d'accès à un fichier local. Ces mesures n'ont pû être faites à cause de la disponibilité tardive des coupleurs Ethernet sur SPS7. Ces mesures de performance permettraient aussi de comparer notre système DISIX au système le plus proche existant commercialement: la Newcastle Connection.

l'originalité de notre projet par rapport à la Newcastle Connection est d'utiliser un mode connecté et d'être conforme au modèle ISO (utilisation des protocoles ISO du niveau physique au niveau session). L'utilisation d'un mode connecté permet également de simplifier le niveau d'appel système à distance (pas de problèmes d'orphelins en particulier). L'utilisation des communications standards ne devraient pas être pénalisante, sauf dans la phase de montage. Ceci devrait être confirmé par les mesures de performances.

Certaines améliorations ou extensions peuvent être apportées à notre système. Principalement, afin de l'adapter à un environnement bureautique :

- montage au niveau de l'utilisateur, c'est-à-dire par poste de travail avec la possibilité de ne monter qu'une partie de système de fichiers distant,

- et surtout prise en compte d'environnement hétérogène (client MS/DOS et serveur UNIX par exemple).

Il serait également intéressant de comparer en détail notre système avec le système de fichiers distribué proposé par SUN. Ce système qui vient d'être proposé par cette compagnie, est par certains côtés trés proche du notre. Notamment, il lui est comparable par l'utilisation d'interfaces standards pour la communication entre systèmes, et par la tentative d'utilisation de protocoles (bien que les protocoles utilisés par SUN ne soient pas ceux proposés par l'ISO). Il lui est aussi comparable par le choix du mécanisme de montage et par l'objectif (réalisé par SUN et en projet pour DISIX), de communication entre systèmes hétérogènes.

Par contre, le système SUN diffère de DISIX du fait qu'il est, comme la Newcastle Connection, sans connexion. Mais sa particularité essentielle est de proposer un mécanisme de serveur sans état; c'est-à-dire qu'aucune information n'est conservée par le serveur sur le contexte du client. Ceci a l'avantage de rendre totalement indépendant le client du serveur entre

chaque opération atomique (lecture ou écriture). Mais cela risque de poser des problèmes en cas d'accès concurrent à un fichier, et nécessite de stocker, chez le client, tout un ensemble d'information sur la situation du client entre deux opérations atomiques. La documentation disponible sur SUN ne nous a d'ailleurs pas permis d'éclaircir tous les points.

Le SUN NFS (Network File System) deviendra fort probablement un standard de système de gestion de fichiers distribué pour systèmes d'exploitation et réseaux hétérogènes, au sein d'UNIX. Les accords actuel entre SUN et AT&T, laissent penser que les prochaines versions d'UNIX d'AT&T (1986) seront commercialisées avec le système de SUN.

Je terminerai ma conclusion en soulignant le fait que la période d'étude et de réalisation qui a abouti à ce mémoire, m'a apporté beaucoup sur le plan personnel et professionnel, en me permettant de me former dans des domaines de l'informatique qui m'étaient peu connus (systèmes d'exploitation, réseaux), et cela sur les systèmes les plus récents et appelés à être fortement développés.

A N N E X E 1

+--------------------+
| LISTE DES ERRNOS |
+--------------------+

100 ER_NOT_SERVED Pas de service
De manière générale, signifie que le service demandé n'est pas fourni par DISIX. Pour ioctl , ce message d'erreur correspond à une requête d'entrées-sorties distantes sur les "sockets" (spécifique BSD 4.2).

101 ER_NO_CWD Champ "CWD" inexistant dans environ
Ce message d'erreur correspond à l'inexistence du champ "CWD" dans l'environnement. Ce champ contient le chemin du "directory" courant, et est utilisé pour rendre un chemin quelconque absolu.

102 ER_NO_PLACE Pas de place mémoire
Ce message apparaît si il y a un manque de place mémoire après exécution d'un malloc , ou d'une primitive d'allocation de mémoire analogue.

103 ER_TAB_SAT Table des systèmes distants du processus saturée
Ce message d'erreur est rendu lorsque la table "systable" est saturée, c'est-à-dire lorsque le nombre maximal de système référencé simultanément par le processus (fichiers ouverts sur chacun ...) est atteint. Ce nombre est pour l'instant égal à 7.

104 ER_FORB_COM Appel interdit sur la machine distante
Ce message d'erreur est rendu par dup2 pour indiquer que cet appel système ne peut pas être exécuté à distance.

105 ER_FORBID_OP Opération interdite entre deux systèmes
Contrairement au message 104, l'appel est accepté entre systèmes distants ; cette erreur apparaît si les opérandes sont incompatibles (opération interdite entre deux ressources éloignées).

106 ER_OVERFLOW Débordement à la conversion
Cette erreur indique que lors de l'opération de conversion des types de deux machines distantes, il y a eu débordement.

107 ER_UNKNOWN_SERV Numéro de service inconnu
Le serveur a reçu un appel de service inexistant.

108 ER_SOCK_USED Fermeture de descriteur interdite
Tentative de fermeture sur un socket connecté à une machine dont le serveur associé a des fichiers ouverts (spécifique BSD 4.2).

A N N E X E 2

MANUEL DES COMMANDES

MODIFIEES

NAME

rpasswd

SYNOPSIS

rpasswd [system]

DESCRIPTION

This command changes the name and his password to access to the remote system system.

Without any argument, rpasswd prints all the remote systems accessible by the user.

The remote system system must exist and be mounted.

Rpasswd asks the user name on the remote system, then the user types the password associated with this name.

FILES

/etc/passwd
$HOME/.rpasswd

SEE ALSO

crypt(3C), mounts(1MD)

NAME

mounts - mount remote system

SYNOPSIS

/etc/mounts [system name]

DESCRIPTION

Mounts announces to the DISIX layer that a remote system is present. The remote system system must exist and not to be already mounted.

The file name must be a directory different of the root. If an ancestor or a descendant directory is already used for a remote mounted system, the command refuses the mounting.

The command creates the process sender associated to the remote system.

Mounts maintains a table of mounted systems in /etc/mtabs. If invoked without an argument, mounts prints the table.

FILES

/etc/mtabs	mounted systems table
/etc/system_table	systems table

SEE ALSO

umounts(1MD)

NAME

umounts - dismount remote system

SYNOPSIS

/etc/umounts system

DESCRIPTION

Umounts dismounts the remote system system if there is no client who has a server on this remote system.

The commande updates the table in /etc/mtabs and the process sender terminates.

FILES

/etc/mtabs	mounted systems table

SEE ALSO

mounts(1MD)

NAME

perror - system and Disix error messages

SYNOPSIS

```
void perror (s);
char *s;
```

DESCRIPTION

perror operates identically to the standard perror(3c) , except that it also investigates the Disix error variable errno. It prints the character string given, followed by a colon, followed by a message ending in a newline. The message may indicate either a normal UNIX error or a network error.

BUGS

there is no facility corresponding to the variables sys nerr and sys errlist for the Disix error messages.

BIBLIOGRAPHIE

References

[Barak80]
A.B. Barak, A. Shapir, UNIX with Satellite Processors, Software Practice and Experience, vol. 10, 80. (The Hebrew University Jerusalem).

[Beyls85]
P. Beyls, R. Sacco, SPIX System User Reference Manual Release 20, 29 323 705 REF 03/EN, Bull Sems, Echirolles, June 85.

[Beyls85a]
P. Beyls, R. Sacco, SPIX System Administrator Reference Manual Release 20, 20 893 157 REF 02/EN, Bull Sems, Echirolles, June 85.

[Beyls85]
P. Beyls, SPIX System Error Reference Manual, 20 893 891 REF 00/EN, Bull Sems, Echirolles, June 85.

[Blair83]
G.S. Blair, J.A. Mariani, W.D. Shepperd, A Pratical Extension to UNIX for Interprocess Communication, Software Practice and Experience, vol. 13, 83. (University of Strathclyde, Glasgow).

[Bourne]
S.R. Bourne, The Unix System, Addison Wesley.

[Brownbridge82]
D.R. Brownbridge, L.F Marshall, B. Randell, The Newcastle Connection or UNIXes of the World Unite!, Software Practice and Experience, vol. 12, Wiley & Sons, July 82.

[Cornafion81]
Cornafion, systèmes informatiques répartis: concepts et techniques, p. 367, Dunod, 81.

[Habert84]
M. Habert, Architecture de logiciel pour système distribué, De nouvelles architectures pour les communications, Eyrolles, Versailles, 84.

References

[Hwang82]
K. Hwang and Al., A UNIX based Local Computer Network with Load Balancing, Computer, Apr. 82. (Purdue University).

[Joy83]
W. Joy, E. Cooper, R.Fabry, S. Leffler, K. McKusik, D. Mosher, 4.2BSD System Manual, University of Berkeley, July 83.

[Joy83a]
W. Joy, 4.2BSD UNIX Programmers Manual, University of Berkeley, Aug. 83.

[Kavaler84]
P. Kavaler, TEAMNET A UNIX Based Distributed File System network, Altos Computer Systems, San Jose, Sep. 84.

[Kernigham]
B.W. Kernigham, D.M. Ritchie, The C Programming Language, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632.

[Leffler83]
S. Leffler, W. Joy, R. Fabry, 4.2BSD Networking Implementation Notes, University of Berkeley, July 83.

[Leffler83a]
S. Leffler, W. Joy, R. Fabry, 4.2BSD Interprocess Communication Primer, University of Berkeley, July 83.

[Lucas84]
H. Lucas, B. Martin, G. de Sablet, UNIX Mécanismes de base, Langage de commande, utilisation, Eyrolles, 84.

[Martin84]
B. Martin, H. Lucas, V. Merrien, Problèmes posés par le developpement de logiciel réseau sous Unix, De nouvelles architectures pour les communications, Eyrolles, Versailles, 84.

[Maughan84]
G.M. Maughan, Proposal for Newcastle Connection under EIES, Sep. 84.

[Panzieri83]
F. Panzieri, B. Randell, Interfacing UNIX to Data Communications Networks, University of Newcastle upon tyne, Dec 83.

References

[Ritchie78]
D.M. Ritchie, The UNIX I/O System, Bell Laboratories, 78.

[SPIX85]
P. Beyls, SPIX System Programmer Reference manual, 20 893 879 REF 00/EN, Bull Sems, Echirolles, June 85.

[Sems83]
Sems, SM90 Manuel de référence, 20 888 439 01/FR, Bull Sems, Echirolles, 83.

[Sems85]
Sems, Methode d'accès distribué Manuel de référence, Bull Sems, Echirolles, 85.

[Sems85a]
Sems, Accès Session et Transport ISO Manuel de référence, 20 893 470 REF, Bull Sems, Echirolles, 85.

[Sun84]
Sun Microsystems, Inc., Remote Procedure Call Protocol Specification, Oct. 84.

[Sun84a]
Sun Microsystems, Inc., Remote Procedure Call Reference Manual, Oct. 84.

[UNIX]
UNIX, SYSTEM INTERNAL, UC-1012, The Process Subsystem, UNITS 0-8, Western Electric Company.

[UNIXa]
UNIX, SYSTEM INTERNAL, UC-1012, The File Subsystem, UNITS 9-16, Western Electric Company.

[Wambecq83]
A.J. Wambecq, NETIX A Distributed Operating System Based on Unix Software and Local Networking, Bell Telephone Manufacturing Company, Antwerp, Belguim, 83.

[Rashid81]
R.F. Rashid, An Inter-Process Communication Facility for Unix, 81.