Systèmes d'exploitation
Module : Mécanismes avancés

G. Padiou, Ph. Quéinnec
Département Informatique et Mathématiques Appliquées
ENSEEIHT
2 rue Camichel, BP 7122, F-31071 Toulouse Cedex 7

21 octobre 2007

Table des matières

1 La notion d'activité **2**
1.1 Définition d'une activité . . . 2
1.2 Transitions d'états d'une activité . . . 3
1.3 Propriétés des activités . . . 4
1.4 Utilisation des activités . . . 5
1.5 Visibilité des données partagées . . . 5
1.6 La synchronisation des activités . . . 6
1.7 Réalisation d'un noyau d'exécution d'activités (ou threads) . . . 6
1.8 L'exemple d'interface POSIX . . . 7
1.8.1 Primitives de base . . . 7
1.8.2 Initialisation . . . 8
1.8.3 Données spécifiques (*thread-specific data*) . . . 8
1.8.4 L'activité initiale . . . 9
1.8.5 Synchronisation : verrous et variables conditions . . . 10
1.8.6 Règles de visibilité mémoire entre activités . . . 12
1.9 Conclusion . . . 12

2 La communication entre processus distants **14**
2.1 La communication par flots . . . 14
2.1.1 La notion de port . . . 15
2.1.2 Les ports standards . . . 16
2.1.3 La notion de socket . . . 16
2.1.4 L'interface de programmation par socket . . . 16
2.1.5 Conclusion . . . 17
2.2 La communication par appel procédural à distance . . . 18
2.2.1 La sémantique d'un appel procédural à distance . . . 19
2.2.2 L'hétérogénéité . . . 20
2.2.3 Les outils de mise en œuvre . . . 20
2.3 Conclusion . . . 22

Préambule

Ce module a pour objectif de compléter l'étude des noyaux de systèmes d'exploitation en abordant différents mécanismes avancés mais aujourd'hui bien définis et couramment exploités dans de tels systèmes. Ces mécanismes apportent de nouveaux services aux applications dans les domaines d'intérêt suivants :

Le parallélisme : nous avons vu que la notion de processus permettait de gérer les traitements de chaque usagers d'un système. Cependant, le parallélisme obtenu est de granularité grossière puisqu'elle se situe au niveau d'un programme exécutable. La notion de processus léger ou activité (en anglais thread) permet de gérer et mettre en œuvre un parallélisme de granularité plus fine se situant au niveau de la procédure (dans un programme). Tout processus, qualifié de lourd par opposition, constitue alors le support d'exécution (machine virtuelle) à un ensemble d'activités issues d'un même programme et représente une unité d'allocation de ressources pour les traitements soumis au système par un usager.

La communication entre processus distants : il s'agit là de l'extension majeure de la dernière décennie qui s'est imposée avec les progrès rapides des réseaux de communication. Dans ce domaine, le schéma protocolaire du client-serveur s'est imposé. Nous aborderons tout d'abord l'interface de communication de niveau le plus élémentaire par échange de messages via le mécanisme de socket. Puis, nous développerons une approche plus abstraite fondée sur la notion d'appel de procédure à distance (RPC : Remote Procedure Call).

La gestion mémoire : la protection des programmes en cours d'exécution les uns vis-à-vis des autres constitue une nécessité pour obtenir un système fiable. Par ailleurs, les instructions d'un programme ne peuvent être exécutées par un processeur que lorsqu'elles sont en mémoire centrale. Par conséquent, la mémoire centrale est une ressource critique dont la gestion doit être optimisée de façon à pouvoir exécuter le maximum de programmes dans un espace minimal. La notion de mémoire virtuelle permet d'une part, d'assurer une protection sûre des programmes en cours d'exécution et du noyau d'exécution lui-même et d'autre part, d'optimiser l'utilisation de la mémoire centrale.

1 La notion d'activité

Dans les systèmes d'exploitation, la granularité du parallélisme apporté par le concept de processus est liée aux programmes applicatifs des usagers. Un processus gère l'exécution d'un programme binaire objet exécutable. Il contrôle l'allocation des ressources nécessaires à cette exécution.

La notion d'activité permet d'obtenir une granularité plus fine. Un activité reste un processus dans le sens où il s'agit une nouvelle fois de contrôler et gérer l'exécution séquentielle d'un flot d'instructions. Cependant, l'unité à exécuter n'est plus un programme, mais une procédure dans un programme. On obtient ainsi une granularité de parallélisme plus fine.

La figure (1) illustre le parallélisme obtenu grâce à la notion d'activité. Un processus devient le support d'exécution non plus d'un traitement séquentiel mais d'un ensemble de traitements séquentiels de granularité plus fine.

1.1 Définition d'une activité

Un activité (en anglais thread) est le suivi et la gestion de l'exécution séquentielle d'une procédure dans l'environnement support d'un processus lourd. Ce dernier apparaît alors comme unité d'allocation de ressources et comme "machine virtuelle" support.

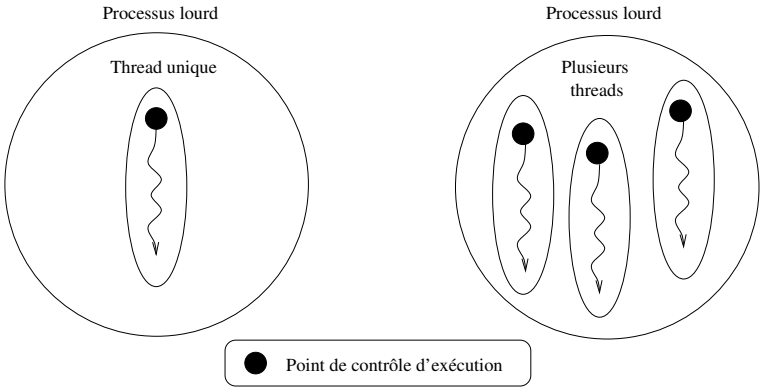

FIG. 1 – Granularité du parallélisme

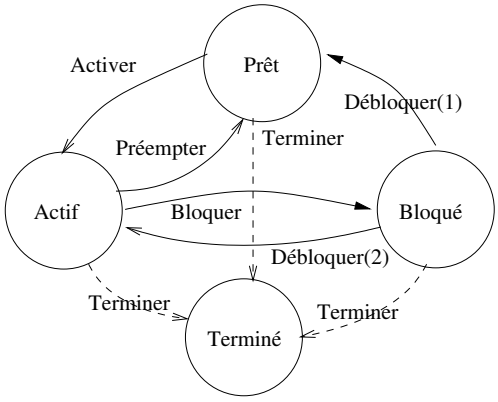

FIG. 2 – Graphe de transition d'états d'une activité

Les activités engendrées par un même processus peuvent s'exécuter en parallèle. Ce parallélisme peut être réel ou simulé par simple partage du temps processeur, alloué au processus lourd, entre les différentes activités.

1.2 Transitions d'états d'une activité

À un certain niveau d'abstraction, une activité peut être représentée par un automate d'états. La figure (2) donne le graphe de transitions d'états d'une activité. On remarquera la similitude, naturelle, avec le graphe de transitions d'états d'un processus lourd (mono-activité).

La création La création d'une activité consiste à créer un objet activité (descripteur) comportant en données la procédure à exécuter et les paramètres éventuels d'appel de cette procédure. Une activité doit disposer d'une pile d'exécution privée qui permettra de gérer les blocs d'activation procéduraux au cours de son exécution.

Le déroulement de l'exécution Une fois créée, l'exécution de l'activité sera effective lorsqu'elle sera dans l'état actif. Le blocage d'une activité sur une condition logique quelconque (comme pour les processus lourds) entraînera son passage dans l'état bloqué. Le déblocage replacera celle-ci soit dans l'état actif, soit dans l'état prêt.

La terminaison La terminaison d'une activité est une opération délicate qui peut avoir plusieurs causes différentes :

- l'activité se termine par la fin de l'exécution de la procédure : c'est le cas le plus classique. La terminaison de la procédure associée à l'activité induit la terminaison de l'activité elle-même et par conséquent entraîne implicitement le choix d'une autre activité à exécuter ;
- l'activité se termine par un appel explicite d'une primitive de terminaison (*thread_exit*). Ce cas est similaire au précédent. Il ne faut toutefois pas confondre cette primitive avec la primitive de terminaison du processus lourd (*exit*) qui entrainerait la terminaison globale du processus et donc de toutes les activités ;
- l'activité provoque une erreur (déroutement). Elle ne peut continuer son exécution et doit être avortée ;
- l'activité est détruite par un autre activité via une primitive explicite de destruction ;

La terminaison d'une activité doit s'accompagner de la libération des ressources utilisées par cette dernière (pile d'exécution par exemple). Cependant des contraintes de synchronisation entre activités sur l'événement de terminaison peuvent conduire à conserver un minimum d'informations sur cet objet devenu "zombi".

Si possible, on doit pouvoir contrôler les effets de bord d'une activité et exécuter éventuellement un ensemble d'actions de terminaison visant à assurer la transparence de la terminaison normale ou anormale de l'activité vis-à-vis des autres activités.

La terminaison d'une activité n'est évidemment plus synonyme de la terminaison du processus lourd support. La terminaison de ce processus lourd aura lieu lorsque TOUTES les activités créées seront terminées ou par l'appel explicite de la primitive de terminaison globale d'un processus lourd (*exit*).

1.3 Propriétés des activités

Contrairement au concept de processus qui avait entre autre objectif de bien contrôler et cloisonner l'exécution de différents programmes indépendants entre eux, le concept d'activité se concentre exclusivement sur l'aspect parallèle en conservant un couplage fort entre activités. Les activités partagent le même environnement d'exécution fourni par le processus lourd. Il en découle un ensemble de propriétés qui les différencient des processus lourds mono-activité :

1. les activités partagent le même espace mémoire. Elles peuvent donc communiquer par variables partagées ;
2. les activités partagent les canaux de communication offerts par le processus support. Par conséquent, toute liaison à une ressource telle qu'un fichier, ouverte par une activité, est valide implicitement pour toutes les autres activités ;
3. les activités partagent le système de gestion des exceptions du processus lourd d'accueil. Dans le cas d'un déroutement, l'activité active fautive peut être repérée sans ambiguïté. Par contre, lorsqu'il s'agit d'une interruption, le choix de l'activité à interrompre peut être délicat.

Ce partage a deux conséquences importantes :

- un avantage : la commutation d'activité est beaucoup moins coûteuse que la commutation de processus. Elle se réduit pratiquement à une commutation de pile d'exécution. Elle est donc très rapide.
- un inconvénient : le couplage plus étroit implique un moins bon cloisonnement des activités entre elles. Il faudra donc d'autant plus prendre soin au contrôle des interactions entre activités.

Lorsqu'on conçoit un programme dont l'exécution engendrera plusieurs activités dynamiquement, il est en particulier important de se poser le problème de la réentrance des procédures appelées. En effet, toute procédure est implicitement partagée (au sens où elle est appelable) par toute activité. L'exécution en parallèle (pseudo-parallélisme y compris) de deux appels de la même procédure par deux activités différentes donnera un résultat a priori correct à condition que la procédure soit réentrante. Or, toute procédure modifiant des variables partagées communes aux deux activités n'est pas réentrante. Il faudra alors mettre en œuvre une synchronisation assurant le bon usage des variables partagées (par exemple en garantissant un accès en exclusion mutuelle).

Dans un tel contexte d'exécution, une condition suffisante de réentrance est qu'une procédure possède les propriétés d'une fonction (au sens mathématique). En effet, une telle procédure n'accède alors qu'aux

variables locales allouées dans la pile privée de l'activité appelante. Par conséquent, il n'y a pas de conflit d'accès à des variables partagées durant son exécution.

Lorsque des activités font appel à des procédures d'une bibliothèque, il faudra donc prendre soin de contrôler si celle-ci sont réentrantes.

1.4 Utilisation des activités

On peut évidemment se poser la question de savoir si ce niveau de granularité de parallélisme est utile. De ce qui précède, on peut souligner que la gestion des activités peut augmenter le parallélisme des applications à un coût raisonnable. Or, il existe de nombreuses applications pour lesquelles cette granularité de parallélisme et ce partage de ressources est intéressant. Nous en donnons une illustration sur un exemple très courant, en l'occurrence une application obéissant au paradigme client-serveur. Dans ce modèle générique d'applications, des processus clients adressent des requêtes à un processus serveur auquel il revient de satisfaire ces requêtes. Les requêtes des clients sont évidemment totalement asynchrones et peuvent donc être parallèles. Il est alors intéressant de pouvoir structurer les traitements du serveur en associant une activité particulière à chaque client en mode connecté, voire à chaque requête client en mode déconnecté (exemple d'un serveur Web). L'activité "racine" conserve, pour sa part, la tâche d'écouter tout nouveau client (ou toute nouvelle requête) et de créer au fur et à mesure des activités "serveurs".

Un autre intérêt majeur du concept d'activité concerne le modèle de programmation qu'il offre au programmeur. En effet, la structuration d'une application en différentes activités constitue une solution alternative à la programmation dite événementielle. On parle de programmation événementielle lorsque l'on développe une application de type réactive qui doit accepter des événements issus d'un environnement et réagir en conséquence sur ce dernier. Très schématiquement, le comportement générique d'un tel programme peut être décrit comme un automate :

```
loop {
   wait(e) ;
   switch(e) {
      case E1 : Trans(E1,ctx) ; break ;
      ...
      case En : Trans(En,ctx) ; break ;
      }
   }
```

Si cette structure peut paraître simple, elle est très primitive et ne donne aucune idée de l'évolution du contrôle entre suites d'événements ayant une causalité entre eux. Par opposition, la notion d'activité permet de décrire le déroulement d'un traitement activé par une séquence d'événements asynchrones mais causalement liés qu'il captera au fur et à mesure de son exécution. Cette approche permet de rendre explicite une synchronisation d'un traitement sur une suite d'événements asynchrones apportant ainsi une forme de modèle de programmation "virtuellement synchrone".

1.5 Visibilité des données partagées

L'une des propriétés intéressantes des activités est le partage de données via un espace mémoire commun. Cependant, dans le cas d'architectures multiprocesseur, les activités peuvent s'exécuter en parallélisme réel. Or, sur ce genre d'architecture, l'accès à un mot mémoire partagé par deux activités parallèles peut conduire à des anomalies :

- Modification concurrente
  ⟨ `x := 0x01` ⟩ ∥ ⟨ `x := 0x100` ⟩
  La variable x est modifiée par les deux processeurs et selon la granularité de l'atomicité des écritures, le résultat final peut être : ⇒ `x = 0x01` ou `0x100` ou `0x101` ou `3.1415!`
- Exécution concurrente
  ⟨ `a := x ; x := a + 1` ⟩ ∥ ⟨ `b := x ; x := b - 1` ⟩

Dans ce cas, on n'est pas assuré d'obtenir un résultat équivalent à une exécution séquentiel des deux activités. Le résultat peut être issu d'un entrelacement quelconque des instructions des deux activités et donner par exemple : ⇒ `x = -1, 0` ou `1`.

- Causalité et cohérence mémoire
  ⟨ `x := 1 ; y := 2` ⟩ ∥ ⟨ `printf("%d %d",x,y) ;` ⟩
  Dans ce cas, la causalité des écritures de x puis de y par la première activité ne sera par forcément « observée » par l'autre activité qui pourra afficher ⇒ `0 2` ! Ce genre d'anomalie a pour origine la gestion des caches des processeurs.

On voit donc qu'il faudra prendre soin de synchroniser explicitement les accès aux données partagées par des activités si l'on ne veut pas avoir de résultats erronés.

1.6 La synchronisation des activités

Qui dit parallélisme, dit synchronisation. Pour synchroniser des activités, on va bien évidemment utiliser les mécanismes génériques utilisés avec les processus lourds. La norme POSIX offre, à titre d'exemple, les briques de base suivantes :

- la synchronisation directe sur la terminaison d'une activité : une activité peut se bloquer en attente de la fin d'un autre activité par une opération *join*. L'exécution de cette primitive permettra de libérer toutes les ressources de l'activité terminée. Si l'on ne souhaite pas contrôler la fin d'une activité, il faudra appeler une primitive *detach* indiquant au noyau que la destruction totale de l'objet activité est possible dès sa terminaison ;
- le mécanisme de sémaphore d'exclusion mutuelle, appelé encore verrou d'exclusion ;
- le mécanisme de variable condition tiré des moniteurs de Hoare. Cependant, ce mécanisme ne garantit pas une priorité de l'activité signalée (réveillée) sur l'activité signaleuse. Le mécanisme de moniteur de Hoare n'est donc pas directement utilisable. Seule reste la possibilité de contrôler l'accès en exclusion mutuelle à des données partagées. Une primitive de signalisation de toutes les activités bloquées sur une variable condition est aussi spécifiée.

1.7 Réalisation d'un noyau d'exécution d'activités (ou threads)

Pour mettre en œuvre la notion d'activité, il faut définir une interface de programmation. Elle apparaîtra comme la définition d'une classe d'objets "activités" définissant d'un ensemble de méthodes permettant d'utiliser des activités. Un effort de normalisation a été fait en la matière et il existe notamment une norme POSIX de threads. Cette interface doit bien entendu être finalement implantée avec deux contextes possibles.

Le premier consiste à réaliser une bibliothèque de primitives qui sera intégrée à tout programme, utilisant des activités, lors de l'édition des liens. Les activités ne sont alors pas connues du noyau du système d'exploitation. Leur gestion s'appuie sur celle du processus lourd seul connu du noyau et elles ne peuvent alors s'exécuter qu'en mode pseudo-parallèle. Dans ce contexte d'implantation, il n'est pas suffisant d'implanter les seules primitives de manipulation d'activités. En effet, l'appel de toute primitive bloquante du noyau du système d'exploitation sous-jacent provoque le blocage du processus lourd appelant (le noyau ne connaît pas l'existence des activités). Il faut donc implanter une bibliothèque de procédures ayant la même interface mais fonctionnant en mode non bloquant vis-à-vis du processus lourd. Ces procédures gèreront le blocage des activités et non pas le blocage du processus lourd support. Il y a donc un travail important de développement d'une bibliothèque spécifique aux activités, notamment pour les entrées/sorties.

Le second consiste à intervenir dans le noyau même du système d'exploitation et d'y intégrer directement la gestion des activités. Dans ce cas, le parallélisme entre activités peut être réel si le système est multiprocesseur. De plus, l'ordonnancement des activités peut être contrôlé et piloté avec une précision plus grande. Des priorités peuvent par exemple être fixées et respectées globalement quel que soit l'appartenance des activités aux processus.

1.8 L'exemple d'interface POSIX

L'interface décrite ici correspond à la norme POSIX1003.1c. Elle fournit des primitives pour créer des activités et les synchroniser. Ces primitives sont similaires à celles fournies par diverses autres bibliothèques (SunOS lightweight process library, Solaris LWP) ou langages (Modula-3, Java). Dans le domaine des processus légers, la norme POSIX s'est imposée, tout au moins dans le monde Unix.

Code d'erreur La majorité des routines de la bibliothèque `pthread` renvoient un entier. La valeur de retour est 0 si tout va bien, et un code d'erreur sinon (généralement > 0). Ce code d'erreur correspond aux valeurs possibles de errno (cf `/usr/include/errno.h`). On trouvera par exemple `EFAULT` (pointeur erroné), `EINVAL` (invalid argument), `EBUSY` (destruction d'un objet utilisé, ou test de verrouillage), `EPERM` (permission denied)...

1.8.1 Primitives de base

Création-Terminaison

```
int pthread_create (pthread_t *thread, const pthread_attr_t *attr,
                    void * (*start_routine)(void *), void *arg) ;
```

Crée un nouvelle activité pour exécuter la routine indiquée, appelée avec l'argument `arg`. On ignorera les attributs qui sont principalement utilisés pour définir la priorité et la politique d'ordonnancement (scheduling policy). `thread` contient l'identificateur de l'activité créée.

```
void pthread_exit (void *status) ;
```

Termine l'activité appelante en fournissant un code de retour (remarquer qu'il s'agit d'un pointeur). `pthread_exit(NULL)` est automatiquement exécuté en cas de terminaison du code de l'activité sans appel de `pthread_exit`.

```
int pthread_join (pthread_t thread, void **status) ;
```

Attend la terminaison de l'activité indiquée et récupère le code retour. L'activité ne doit pas être détachée (voir 1.8.1).

Identification

```
pthread_t pthread_self (void) ;
```

Renvoie l'identificateur de l'activité appelante.

```
int pthread_equal (pthread_t thread_1, pthread_t thread_2) ;
```

Renvoie vrai si les arguments désignent la même activité, faux sinon.

Libération des ressources

```
int pthread_detach (pthread_t thread) ;
```

Détache l'activité `thread`. Normalement, les ressources allouées pour l'exécution d'une activité (pile...) ne sont libérées que lorsque l'activité s'est terminée et que l'exécution d'un `join` s'est produite. Pour éviter de devoir se synchroniser sur la terminaison d'une activité dont on compte ignorer le code retour, on peut détacher cette activité, auquel cas les ressources sont libérées dès la terminaison de l'activité.

Exemple

```
#include <pthread.h>
#include <stdio.h>
void *routine (void *arg) {
    int *status = malloc (sizeof(int)); /* pour renvoyer le code de retour */
    printf ("Arg = %d\n", *(int *)arg); /* casting vers (int *) necessaire */
    *status = *(int *)arg * 2;
    pthread_exit (status);
}

int main() {
    pthread_t un_p;
    int erreur, argument = 3;
    int *resultat;
    erreur = pthread_create (&un_p, NULL, routine, &argument);
    if (erreur != 0) fprintf(stderr,"Echec creation de thread: %d\n",erreur);
    pthread_join (un_p, (void **)&resultat);
    printf ("Resultat: %d\n", *resultat);
    exit(0);
}
```

Contrôle des effets de bord Pour simplifier la terminaison d'une activité, il est possible d'enregistrer des procédures qui seront automatiquement appelées à la terminaison de l'activité de façon à assurer une terminaison correcte veillant à effacer les effets de bord éventuels de cette activité.

```
void pthread_cleanup_push (void (*routine)(void *), void
*routine_arg) ;
void pthread_cleanup_pop (int execute) ;
```

La première procédure enregistre, dans la pile de nettoyage de l'activité appelante, la procédure `routine` comme devant être appelée avec le paramètre `routine_arg`. À la terminaison, les procédures de nettoyage sont appelées dans l'ordre inverse de leur enregistrement.

La deuxième procédure dépile le sommet de la pile de nettoyage de l'activité appelante. Si `execute` est non nul, la procédure de nettoyage est en outre exécutée.

1.8.2 Initialisation

```
pthread_once_t once_controle = PTHREAD_ONCE_INIT;
int pthread_once (pthread_once_t *ctl, void (*init_routine)(void));
```

Permet d'assurer que la procédure `init_routine` est appelée une et une seule fois.

1.8.3 Données spécifiques (*thread-specific data*)

```
pthread_key_t clef;
int pthread_key_create (pthread_key_t *clef, void (*destructeur)(void *));
int pthread_key_delete (pthread_key_t clef);

int pthread_setspecific (pthread_key_t clef, const void *value);
void *pthread_getspecific (pthread_key_t clef);
```

Une *clef* est créée avec `pthread_key_create` et peut être détruite avec `pthread_key_delete` (rarement utile).

Données spécifiques : pour une clef donnée (partagée), chaque activité possède *sa propre valeur* associée à cette clef (ou `NULL` en absence de valeur positionnée).

Lors de la terminaison d'une activité, pour chaque clef ayant une valeur associée dans cette activité, le destructeur positionné lors de la création de la clef est exécuté, avec la valeur associée en paramètre.

Exemple : associer un *identifiant* (numérique) à chaque activité

```
#include <pthread.h>
#include <stdlib.h>
#include "t_ident.h"

static int numthread;
static pthread_mutex_t excl_tid;
static pthread_key_t key_tid;

static pthread_once_t once_init_tid = PTHREAD_ONCE_INIT;

static void free_tid (void *tid) {
    if (tid != NULL) free (tid);
}

/* Initialise la clef pour obtenir le nom et le mutex. */
static void init_tid (void) {
    numthread = 0;
    pthread_key_create (&key_tid, free_tid);
    pthread_mutex_init (&excl_tid, NULL);
}

/* Get thread id. If the thread has no id, set it now. */
int gettid (void) {
    int *pi;
    pthread_once (&once_init_tid, init_tid);
    pi = (int *) pthread_getspecific (key_tid);
    if (pi == NULL) {
        pi = malloc (sizeof (int));
        pthread_mutex_lock (&excl_tid);
        *pi = numthread;
        numthread++;
        pthread_mutex_unlock (&excl_tid);
        pthread_setspecific (key_tid, pi);
    }
    return *pi;
}
```

1.8.4 L'activité initiale

Lorsque le programme démarre, une activité est automatiquement créée pour exécuter la procédure `main`. Plus précisément, elle exécute une procédure de démarrage qui contient le code :

```
{ int r = main(argc,argv); exit(r); }
```

Cela signifie que si la procédure `main` se termine, le process lourd est ensuite terminé (par l'appel à `exit`), et non pas seulement l'activité initiale. Pour éviter que la procédure `main` ne se termine alors qu'il reste des activités en cours, on peut :

- bloquer l'activité initiale sur l'attente de la terminaison d'une ou plusieurs autres activités par l'utilisation de `pthread_join` ;

– terminer explicitement l'activité initiale avec `pthread_exit`, ce qui court-circuite l'appel de `exit`.

1.8.5 Synchronisation : verrous et variables conditions

Les activités ayant en commun des données, il est nécessaire d'assurer la cohérence des accès à ces données partagées en utilisant des mécanismes classiques de synchronisation. Les noyaux normalisés proposent comme mécanismes les verrous d'exclusion (sémaphores d'exclusion nutuelle) et les variables conditions issues des moniteurs de Hoare. Ainsi, la bibliothèque POSIX fournit des verrous et des variables conditions. Ceci permet d'implanter une notion de module partagé proche du concept de moniteur de Hoare, à ceci près que les règles de réveil des activités bloquées sur une variable condition ne donnent pas la priorité à l'activité réveillée par rapport à l'activité ayant provoqué ce réveil par la primitive signal. Il faudra donc tenir compte de ces règles d'ordonnancement lors de l'écriture de solutions pour éviter en particulier des phénomènes de famine.

Création dynamique ou statique d'un verrou

```
int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutex_attr *attr) ;
                                    ou
            pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER ;
```

Destruction d'un verrou

```
int pthread_mutex_destroy (pthread_mutex_t *mutex) ;
```

Verrouillage/déverrouillage

```
int pthread_mutex_lock (pthread_mutex_t *mutex) ;
```

Verrouillage, avec blocage en attente si déjà verrouillé. Renvoie 0 si ok.

```
int pthread_mutex_trylock (pthread_mutex_t *mutex) ;
```

Verrouillage si possible et renvoie 0, sinon renvoie `EBUSY` si le verrou est déjà verrouillé (ou, comme toujours, `EINVAL` ou `EFAULT` en cas d'erreur).

```
int pthread_mutex_unlock (pthread_mutex_t *mutex) ;
```

Déverrouillage. Seule l'activité qui a verrouillé `mutex` a le droit de le déverrouiller (en cas de tentative de déverrouiller un mutex verrouillé par un autre thread, le comportement est indéfini).

Si une ou plusieurs activités sont bloquées en attente du verrou, une d'entre elles obtient le verrou et est débloquée. Le choix de cette activité dépend des priorités et des politiques d'ordonnancement. Si les attributs (pour la création des activités et des mutex) ne sont pas utilisés, on considérera que l'ordre de déblocage est l'ordre de blocage (FIFO), *même si le système SOLARIS ne spécifie pas un tel ordre.*

Création dynamique ou statique d'une variable condition

```
int pthread_cond_init (pthread_cond_t *cond, const pthread_cond_attr *attr) ;
                                    ou
            pthread_cond_t cond = PTHREAD_COND_INITIALIZER ;
```

Destruction d'une variable condition

```
int pthread_cond_destroy (pthread_cond_t *cond) ;
```

Blocage sur une variable condition

```
int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex) ;
```

L'activité appelante doit posséder le verrou `mutex`. L'activité est alors bloquée sur la variable condition après avoir libéré le verrou. L'activité reste bloquée jusqu'à ce que la variable condition soit signalée et que l'activité ait réussi à réacquérir le verrou.

Réveil simple ou général sur une variable condition

```
int pthread_cond_signal (pthread_cond_t *cond) ;
```

Signale la variable condition : une activité bloquée sur la variable condition est réveillée. Cette activité tente alors de réacquérir le verrou correspondant à son appel de `cond_wait`. Elle sera effectivement débloquée quand elle réussira à réacquérir ce verrou. Il n'y a aucun ordre garanti pour le choix de l'activité réveillée. L'opération signal n'a aucun effet s'il n'y a aucune activité bloquée sur la variable condition (pas de mémorisation).

```
int pthread_cond_broadcast (pthread_cond_t *cond) ;
```

Toutes les activités en attente sont réveillées, et tentent d'obtenir le verrou correspondant à leur appel de `cond_wait`.

Propriétés des verrous et variables conditions

Les verrous sont par défaut des verrous d'exclusion mutuelle. En utilisant les attributs (non documentés ici), on peut créer un verrou dit *récursif* qui peut être verrouillé plusieurs fois par la même activité (le verrou devra alors être autant de fois déverrouillé avant qu'une autre activité puisse l'acquérir).

Par ailleurs, contrairement à la définition des moniteurs de Hoare, l'activité signalée n'a pas priorité sur la signaleuse : la signaleuse ne perd pas l'accès au moniteur si elle le possédait, et la signalée reste bloquée tant qu'elle n'obtient pas le verrou. C'est pourquoi il est nécessaire d'utiliser une boucle d'attente réévaluant la condition d'exécution. En effet, cette condition peut être invalidée entre le moment où l'activité est signalée et le moment où elle obtient effectivement le verrou, par exemple si une autre activité obtient ce verrou et pénètre dans le moniteur avant l'activité signalée.

Enfin, pour des raisons d'efficacité, il est courant de faire l'appel à `cond_signal` hors de la zone parenthésée par `lock` et `unlock`, de sorte que l'activité signalée puisse acquérir plus facilement le verrou. Attention cependant à garantir l'atomicité des opérations du moniteur !

Exemple de moniteur : producteur/consommateur à une case

```
/* Time-stamp: <04 Nov 2002 11:06 padiou@enseeiht.fr> */
#include <pthread.h>
#include <string.h>
#include "prodcon.h"
/* Variables conditions */
static pthread_cond_t est_libre, est_plein;
/* Verrou */
static pthread_mutex_t protect;
/* Variable tampon à protéger */
static char *buffer;
```

```
/* Depose le message msg (qui est dupliqué). Bloque tant que le tampon est plein. */
void deposer (char *msg) {
    pthread_mutex_lock (&protect);
      while (buffer != NULL) pthread_cond_wait (&est_libre, &protect);
      /* buffer = NULL */
      buffer = strdup (msg); /* duplication de msg */
      pthread_cond_signal (&est_plein);
    pthread_mutex_unlock (&protect);
}

/* Renvoie le message en tête du tampon. Bloque tant que le tampon est vide.
 * La libération de la mémoire contenant le message est à la charge de l'appelant. */
char *retirer (void) {
    char *result;
    pthread_mutex_lock (&protect);
      while (buffer == NULL) pthread_cond_wait (&est_plein, &protect);
      /* buffer != NULL */
      result = buffer;  buffer = NULL;
      pthread_cond_signal (&est_libre);
      pthread_mutex_unlock (&protect);
    return result;
}

/* Initialise le producteur/consommateur. */
void init_prodco (void) {
   /* Création des variables de synchronisation */
    pthread_mutex_init (&protect, NULL);
    pthread_cond_init (&est_libre, NULL);
    pthread_cond_init (&est_plein, NULL);
    buffer = NULL;
}
```

1.8.6 Règles de visibilité mémoire entre activités

- toute valeur mémoire visible par une activité avant un appel à `pthread_create` est aussi visible par la nouvelle activité à son démarrage ;
- toute valeur mémoire visible par une activité lorsqu'elle libère un mutex (explicitement ou par blocage sur une variable condition) sera aussi visible par toute activité qui verrouillera ce même mutex ;
- toute valeur mémoire visible par une activité quand elle signale (`cond_signal` ou `cond_broadcast`) est aussi visible par la ou les activités réveillées par ce signal ;
- toute valeur mémoire visible par une activité quand elle se termine (`pthread_exit` ou fin de la procédure de départ ou annulation) est aussi visible par l'activité qui la « joint » (`pthread_join`).

1.9 Conclusion

Nous terminerons par un bilan avantages/inconvénients auxquels il faut s'attendre lorsque l'on utilise des activités. Le tableau ci-dessous résume les principaux points clés.

Domaine	Avantages	Inconvénients
Parallélisme	Granularité fine Parallélisme réel	Surcoût d'implantation Difficulté de mise-au-point
Synchronisation	Couplage fort	Attention à la réentrance
Programmation	Modèle structuré	Dangers d'interférence

En résumé, la notion d'activité offre un modèle de programmation permettant de mieux structurer des programmes réactifs notamment ceux obéissant au modèle générique client-serveur. Néanmoins, le couplage fort existant entre les activités via les nombreuses ressources qu'elles partagent, nécessite d'analyser avec soin les interférences parfois complexes qui peuvent exister.

Pour un approndissement de ce thème, on retiendra les références suivantes :

- le livre publié des ingénieurs de Sun Microsystems et SunSoft présentant à la fois la norme POSIX et la norme Unix International [KSS96] ;
- le livre écrit par D. R. Butenhof, ingénieur chez Digital, artisan de la norme IEEE POSIX [But97] et contenant une bonne étude des avantages et inconvénients de la programmation par threads ;

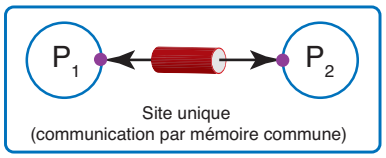

FIG. 3 – Communication entre processus locaux via un pipe

2 La communication entre processus distants

Les progrès technologiques des réseaux de communication ont permis de connecter les ordinateurs de façon à concevoir des applications dites réparties. Les usagers (clients) de telles applications peuvent être géographiquement localisés dans des sites différents, leur machine n'ayant pas forcément la même architecture ou n'étant pas obligatoirement exploitée sous le contrôle du même système d'exploitation. L'avantage majeur d'une telle approche est la possibilité offerte de partage de ressources. En effet, un usager va pouvoir accéder aux ressources de toutes les machines qui sont connectées au réseau à condition, bien sûr, qu'ils en aient le droit.

Il a donc fallu définir des protocoles pour établir une communication entre des processus distants. Ces protocoles définissent de facto une façon de structurer une application répartie composée de processus communicants via des abstractions réparties. Nous n'insisterons pas ici sur les principes généraux de conception d'abstractions, d'objets répartis. En la matière, on se limitera ici à l'exposé de deux modèles d'exécution répartie :

- le premier consiste à définir un protocole de communication par flots de données. Un tel protocole permet d'établir une liaison fiable et bidirectionnelle entre deux processus. Ce type de communication est fondée sur une extension répartie du mécanisme de pipe (ou tube) Unix. Au niveau réseau, ces flots seront évidemment implantés par des messages. Il existe d'ailleurs, en général, une possibilité d'utiliser directement un mode "message" de plus bas niveau que le mode par flot ;
- le second consiste à définir un mécanisme de plus haut niveau, en l'occurrence l'appel procédural à distance. L'appel procédural, mécanisme bien connu du programmeur, est étendu au contexte réparti en autorisant que l'exécution d'une procédure ait pour cause un appel distant. Contrairement au cas centralisé, l'appel et l'exécution proprement-dite d'une procédure ne se font donc plus dans le même espace d'exécution. La mise en oeuvre de ce mécanisme conduit à une relation obéissant au schéma générique du client-serveur. Le processus appelant est le client du processus serveur qui exécute réellement la procédure.

On trouvera dans [CS93c], [CS93b] et [CS93a] une description détaillée des protocoles de communication et en particulier des protocoles de l'Internet TCP/IP.

2.1 La communication par flots

Il s'agit de définir un protocole de communication bidirectionnelle par flots entre deux processus. Chacun des processus est localisé sur une machine et ces machines sont reliées par un médium de communication (réseau local de type Ethernet, ou global tel que le réseau Internet). Les concepteurs de cette interface ont en fait étendu le mécanisme de pipe bidirectionnel à un environnement réparti. La notion de pipe, proposée dans le système Unix, permet de faire communiquer deux processus localisés sur une machine. La figure (3) rappelle de mécanisme.

Le protocole implante un objet réparti de type pipe bidirectionnel dont chaque extrémité est sur une machine différente. La notion de communication par canaux entre processus est ainsi étendue en autorisant une liaison entre des canaux distants. La figure (4) illustre cette situation.

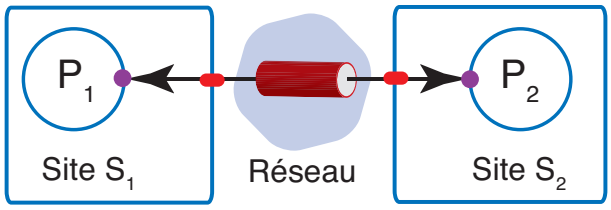

FIG. 4 – Communication par flots entre processus distants

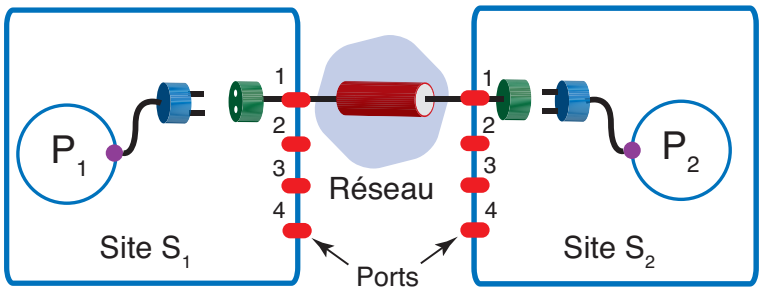

FIG. 5 – La notion de port

2.1.1 La notion de port

Le premier problème à résoudre dans un tel protocole point à point est celui de la désignation des processus distants. En effet, un processus ne possède qu'une désignation locale, par exemple un numéro. Il faudrait donc qu'un processus connaisse le nom local du processus distant pour pouvoir communiquer avec lui en le désignant par un couple (nom de machine, numéro local de processus). Cette approche est pratiquement impossible car le noyau du système d'exploitation attribue des numéros aux processus à l'aide d'un compteur croissant. Tout arrêt et redémarrage du système verraient une réattribution de numéros aux processus sans lien avec l'affectation précédente. Il faudrait alors réapprendre aux processus avec qui ils peuvent communiquer.

Face à cet obstacle, l'idée retenue est d'introduire un mécanisme de désignation **indirecte**. Le noyau (la machine) dispose d'un domaine de noms (numéros) pour définir des points de communication via un réseau. Ces points de communication sont appelés ports. Un processus désignera son partenaire non plus par le couple (nom de machine, nom de processus) mais par le couple $[M, p]$ où M est un nom de machine et p est un numéro de port. Le processus partenaire devra donc être maintenant relié à ce numéro de port pour que la communication puisse s'établir.

Par ailleurs, l'établissement d'une communication nécessite de définir un protocole obéissant à des règles bien définies. En la matière, la stratégie du client-serveur est la plus largement adoptée. On distingue deux comportements génériques de processus : d'une part, le rôle de serveur qui consiste à accepter de recevoir des flots de données qui entraîneront en général des flots de réponses et d'autre part, le rôle de client qui consiste à prendre l'initiative du dialogue en envoyant une demande d'ouverture de flots vers un processus ayant un comportement de serveur.

Pour repérer un processus serveur, un processus client utilise donc un couple (nom de machine, numéro de port). Un processus, pour acquérir le rôle de serveur, devra indiquer au noyau sa connexion à un port particulier. Après quoi, il pourra écouter les éventuelles requêtes d'ouverture de flots arrivant sur ce port. La figure (5) illustre cette approche.

Une remarque importante est à souligner. Pour qu'une communication par flot puisse s'établir, il faut que côté client, un port de communication soit affecté (c'est le seul moyen de communiquer avec l'extérieur via le réseau). C'est pourquoi, une communication par flot est caractérisée et ne peut exister que par la définition d'une paire de coordonnées $([M1, pcl], [M2, psv])$ repérant sans ambiguïté les DEUX extrémités

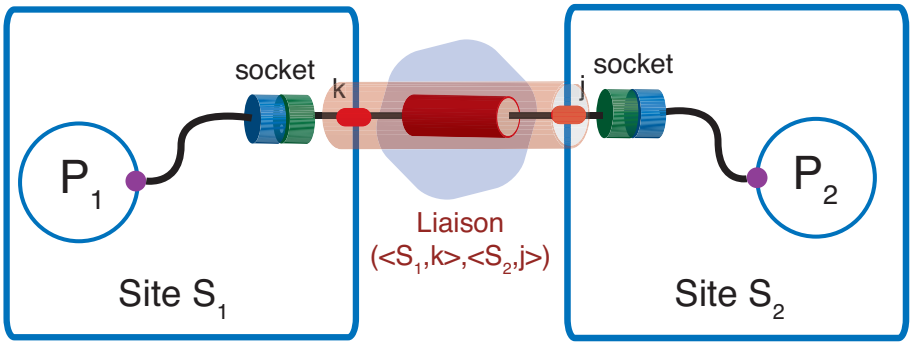

FIG. 6 – La notion de socket

du pipe réparti support.

2.1.2 Les ports standards

Un certain nombre de services répartis sont aujourd'hui disponibles telles que l'accès à distance à des fichiers (ftp), la messagerie électronique (smtp), la connexion à distance (telnet, rlogin), le service Web (http). À titre d'exemple, le tableau suivant donne quelques affectations de numéros de ports.

ftp	21/tcp	
telnet	23/tcp	
smtp	25/tcp	mail
http	80/tcp	web
#		
# UNIX	specific services	
#		
exec	512/tcp	
login	513/tcp	
printer	515/tcp	spooler
who	513/udp	whod
talk	517/udp	

2.1.3 La notion de socket

Le protocole de communication par socket a été introduit en 1982 sur le système Unix 4.1 BSD. Il est aujourd'hui mature, très largement utilisé, disponible sur différents types de plateformes (UNIX BSD, UNIX SystemV, VMS, DOS...), non propriétaire et a prouvé sa résistance aux évolutions de la technologie (propriétés et capacités des réseaux, types d'ordinateurs...). Il est souvent confondu avec TCP/IP (protocole de réseau sous-jacent le plus couramment utilisé).

La connexion entre un canal, qui est un objet local au processus, et un port, qui est un objet associé à la machine (au noyau), sera représentée par un objet "socket" (prise). Un tel objet constitue une ressource, qui une fois créée, permet à un processus de communiquer via le réseau. La figure (6) illustre ce principe.

2.1.4 L'interface de programmation par socket

La mise en œuvre de ces principes se traduit sous la forme d'une interface de programmation (API) comportant un ensemble de primitives spécifiques.

La figure (7) illustre l'utilisation de l'interface de programmation par socket pour établir une communication par flots entre un client et un serveur.

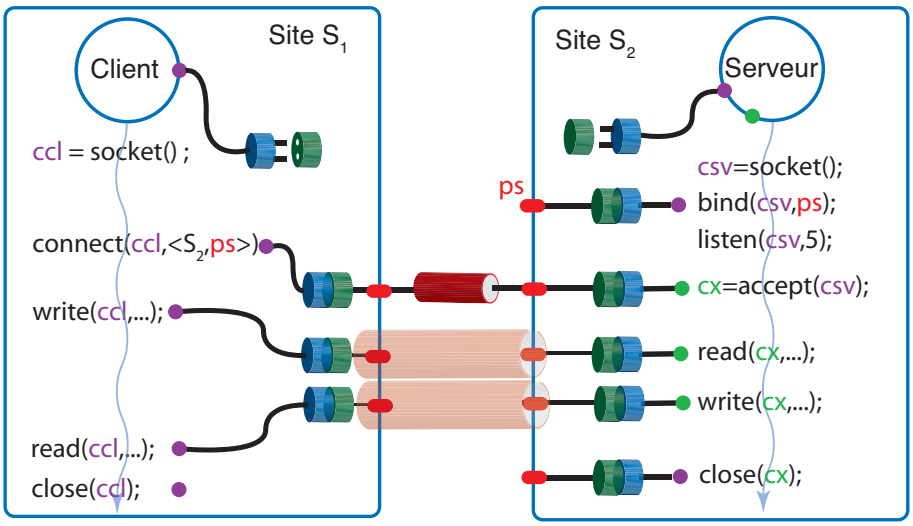

FIG. 7 – Utilisation de l'API par socket

Côté client, il faut naturellement créer un objet socket pour accéder au réseau (primitive *socket*). Si l'opération réussit, un canal est affecté à cette liaison dont le numéro constitue le résultat de l'opération. Désormais, toutes les opérations référenceront ce canal ouvert sur le socket. L'opération *connect* permet d'établir le contact avec le serveur cible. Il faut donc préciser en paramètre les coordonnées de ce processus sous la forme (nom de machine, numéro de port). Puis, la communication peut véritablement se faire. Les opérations classiques *read* et *write* peuvent être utilisées dans ce cas comme pour un canal connecté à une ressource locale classique (fichier, fenêtre,...). L'opération de fermeture du canal (*close*) entraînera la rupture de la liaison avec le socket.

Côté serveur, il faut créer un objet socket qui représente l'autre extrémité de l'objet réparti pipe. L'opération *bind* réalise ensuite la liaison à un port. La primitive *listen* fait passer le socket en mode écoute autorisant le serveur à accepter des demandes de connexions en fixant le nombre de telles demandes acceptables simultanément. Après quoi, le processus serveur peut accepter des demandes de connexion via la primitive *accept*. Cette primitive est bloquante si aucune requête n'est présente lors de l'appel. Dès qu'une requête arrive, une connexion est établie par allocation d'un nouveau canal obtenu en résultat de l'opération. On se trouve alors dans une situation où le processus serveur doit faire deux choses à la fois :

- d'une part, accepter de nouvelles requêtes de connexion provenant d'autres clients ;
- d'autre part, gérer le dialogue avec le client accepté.

Autrement dit, il faut gérer des activités parallèles. Pour ce faire, on peut recourir soit aux processus lourds soit aux processus légers (threads). En effet, il suffit que le nouveau processus (lourd ou léger) s'occupe par exemple du dialogue client ouvert via le canal dédié et que le processus racine se consacre à l'écoute des nouveaux clients.

Comme pour le côté client, une connexion au socket serveur pourra être rompue par la primitive de fermeture de canal.

2.1.5 Conclusion

L'interface de communication par socket permet d'établir un flot de communication bidirectionnel entre deux processus distants. Chaque processus doit créer une des extrémités sous forme d'un objet socket et un protocole de connexion permet de créer une sorte de pipe réparti support des échanges.

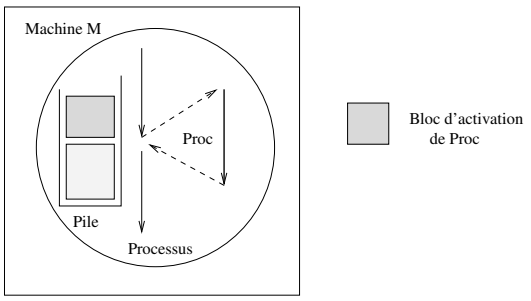

FIG. 8 – L'appel procédural centralisé

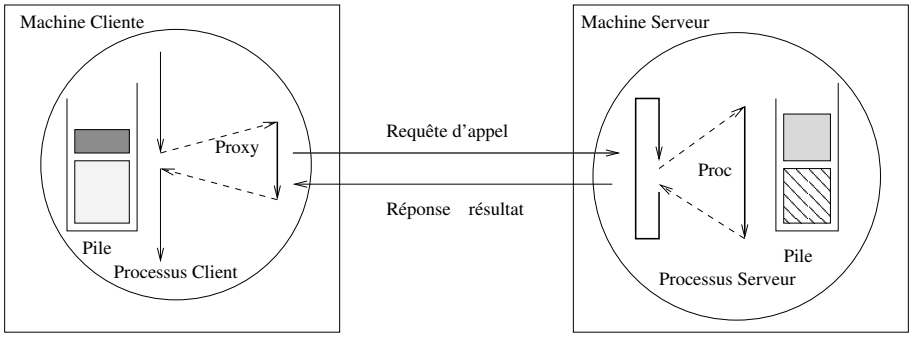

FIG. 9 – L'appel procédural réparti

2.2 La communication par appel procédural à distance

La communication par flots de données fournit un mécanisme de base pour les applications parallèles obéissant plus ou moins au schéma générique du producteur-consommateur éventuellement en cascade. Ce modèle n'est pas a priori le plus intéressant pour les applications réparties. En effet, la situation la plus fréquente obéit plutôt au modèle générique du client-serveur. Un processus sur une machine donnée souhaite bénéficier des ressources offertes par un autre processus sur une autre machine en sous-traitant une partie des traitements à exécuter. Or, du point de vue contrôle, ce schéma correspond au mécanisme procédural. L'idée de base est donc d'étendre la notion de procédure (et d'appel de) de façon à ce qu'une procédure distante puisse être invoquée par un processus. Bien entendu, cet appel à distance implique que le processus qui exécutera réellement la procédure ne sera pas le même que celui qui l'a appelé. La figure (8) illustre le cas centralisé alors que la figure suivante (9) montre le passage au cas réparti.

Ce mécanisme assure par ailleurs un bon niveau de masquage de la répartition des traitements. En effet, dans le contexte d'un schéma réparti de type client-serveur, l'utilisation de l'appel procédural à distance fournit un moyen d'implanter la notion de module distant. Autrement dit, l'unité de répartition est le module. Chaque module encapsule des données et définit un ensemble de procédures publiques accessibles à distance.

Cependant, cette approche soulève deux problèmes de mise en œuvre :

- d'une part, un problème de sémantique : est-il possible de conserver toutes les propriétés sémantiques de l'appel procédural centralisé ?
- d'autre part, un problème d'hétérogénéité : est-il possible de réaliser un tel mécanisme sur une architecture répartie hétérogène ?

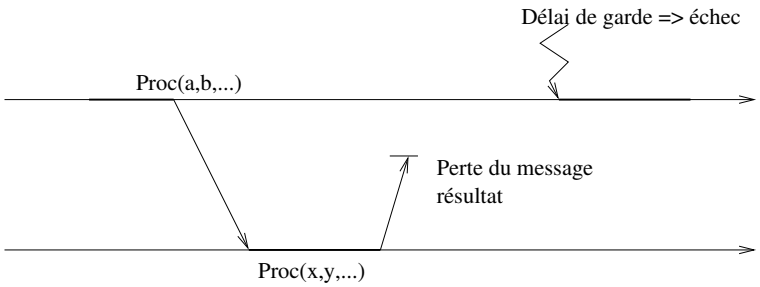

FIG. 10 – Problème de fiabilité du réseau

2.2.1 La sémantique d'un appel procédural à distance

L'appel procédural à distance implique la mise en œuvre de mécanismes très différents de ceux de l'appel centralisé. En particulier, tout appel va se traduire par des échanges de messages et par la participation de deux sites (et processus) distincts.

Un premier aspect sémantique concerne le passage des paramètres. Les paramètres seront passés "par message" ce qui signifie que l'on ne pourra implanter qu'un mode de passage par valeur et résultat. En effet, un passage de paramètre par référence (par adresse) est impossible : l'adresse échangée concerne l'espace d'exécution du processus origine et, en conséquence, celle-ci n'a aucune signification dans l'espace du processus récepteur. À titre d'exemple, si une liste doit être passée en paramètre, c'est une copie de cette liste qu'il faudra fournir au processus serveur et non un simple pointeur de tête de liste.

Dans les contraintes ainsi fixées, il faut aussi contrôler si l'appel de procédure garde sa propriété de base : tout appel se traduit pas une exécution et une seule de la procédure appelée.

Cette propriété qui ne soulève pas beaucoup de problèmes dans un contexte centralisé fiable est beaucoup plus délicate à maintenir dans le contexte réparti. En effet, des erreurs de communication ou une défaillance du processus serveur sont des événements qui peuvent perturber la réalisation d'un RPC et invalider la propriété naturelle de base d'un appel procédural.

L'exemple de la figure (10) montre que le processus client considère que l'appel a échoué puisqu'il n'a pas obtenu de résultat dans le délai prévu (problème d'évaluation de ce délai) alors que le processus serveur a bien satisfait l'appel. Il y a donc une incohérence de point de vue entre le client et le serveur.

Plus généralement, un ensemble de sémantiques possibles ont été définies. Plus on se rapproche d'une sémantique centralisée, plus le coût de mise en œuvre est lourd. On retiendra les sémantiques suivantes :

- Sémantique "sans garantie" (maybe) : il s'agit d'une forme dégénérée d'appel pour lesquels on ne se souci pas de la réponse ;
- Sémantique "au moins une fois" (at-least-once) : il s'agit d'assurer l'exécution de l'appel en tolérant des fautes de la part du réseau voire du serveur. Il faut dans ce cas traiter la perte du message d'appel et/ou du message réponse ou la défaillance temporaire du serveur. Le protocole sous-jacent est donc plus complexe (détection des rappels, des réplications de messages résultats) ;
- Sémantique "au plus une fois" (at-most-once) : il s'agit de garantir, qu'en **cas de réussite**, le client est sûr que la procédure s'est exécutée une seule fois et correctement. La difficulté majeure de mise en œuvre de cette sémantique est d'éviter bien entendu une exécution répliquée de l'appel. Or, la perte d'un message réponse peut conduire à la ré-émission du message d'appel. Dans ce cas, le serveur devra détecter cette ré-émission mais surtout renvoyer à nouveau les résultats de l'exécution réalisée précédemment. Par conséquent, il faudra **mémoriser** ces résultats au cas où l'on devrait les ré-émettre ;
- Sémantique "exactement une fois" (exactly-once) : dans le cas précédent, l'échec d'un appel peut être causé par une défaillance du serveur lors de l'exécution de la procédure. Cette procédure peut donc avoir commencé à modifier des données rémanentes sur le site serveur et les avoir laissé dans un état incohérent. Cette dernière sémantique vise à garantir dans ce cas une atomicité de la procédure : soit celle-ci aura été exécutée complètement, soit pas du tout. Ceci nécessite de mettre en œuvre la notion de transaction.

On constate donc, qu'il est possible d'obtenir une sémantique de type centralisé lorsque le client perçoit la réussite de l'appel. Par contre, l'éventualité de l'échec doit être toujours traitée et dans ce cas, l'interprétation de l'échec par le client peut éventuellement être erronée.

2.2.2 L'hétérogénéité

L'appel procédural à distance, pour être pleinement réparti, doit pouvoir être mis en œuvre sur des architecture hétérogènes. Plusieurs facteurs d'hétérogénéité peuvent exister : les machines peuvent être différentes (processeur Intel d'un côté, processeur Motorola d'un autre), les systèmes d'exploitation peuvent être différents (Windows NT d'un côté, Unix de l'autre), les langages peuvent enfin être différents (C d'un côté, C++ de l'autre).

Le traitement de l'hétérogénéité pose le problème essentiel de la représentation des paramètres dans les messages. La solution retenue est de passer par un format intermédiaire "standard". Chaque partenaire devra alors savoir aussi bien coder que décoder ces paramètres pour les insérer dans les messages et les extraire de ces mêmes messages. Les conventions de passage des paramètres procéduraux dans les langages utilisés imposeront en général de respecter des règles bien précises pour les paramètres des procédures accessibles à distance.

2.2.3 Les outils de mise en œuvre

Nous avons vu que l'implantation du mécanisme de RPC nécessitait l'échange de messages entre processus. Les routines d'appel côté client et d'acceptation côté serveur sont très fastidieuses à écrire. Leur schéma est fixe et seuls les paramètres d'appel diffèrent : module appelé, procédure appelée, paramètres d'appel, paramètres de retour. Il est donc tout à fait possible d'engendrer de telles routines de façon automatique à partir d'une description de l'interface du module dans un langage déclaratif appelé langage de description d'interface (IDL : Interface Definition Language).

L'utilisation du mécanisme de RPC comporte les phases suivantes :
- Décrire l'interface du module ;
- Utiliser un générateur de "talons" (stubs) ;
- Programmer le module serveur et le(s) programme(s) client(s) ;
- Engendrer par édition de liens les binaires exécutables serveur et client(s).

Cette approche devra en général être précédée d'une étape plus amont pour définir les modules susceptibles de devenir accessibles à distance. Lors de cette analyse, il faudra en particulier tenir compte de la répartition des données qui en découlera.

À titre d'exemple, la figure (11) décrit un module minimal comportant une seule procédure dans le langage IDL proposé par SUN.

```
struct Arg ... ;
program MS {
      version MSV {
         int PROC (Arg) = 1 ;
      } = 1 ;
} = 0x30000050 ;
```

Fig. 11 – Description d'interface en IDL

Un numéro unique est attribué au module (service), un numéro de version est fixé et l'interface de la procédure $PROC$ est décrite, celle-ci recevant elle-aussi un numéro.

À partir de cette description, un compilateur d'IDL peut engendrer les talons d'appel et d'acceptation de façon automatique ainsi que les routines de conversion des paramètres. Dans l'exemple, une telle routine sera nécessaire pour le paramètre Arg.

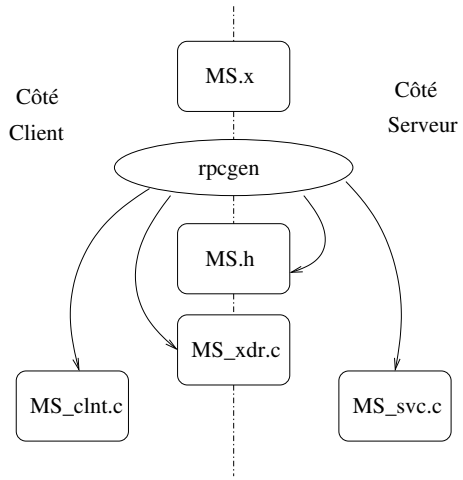

FIG. 12 – Exemple de générateur : rpcgen

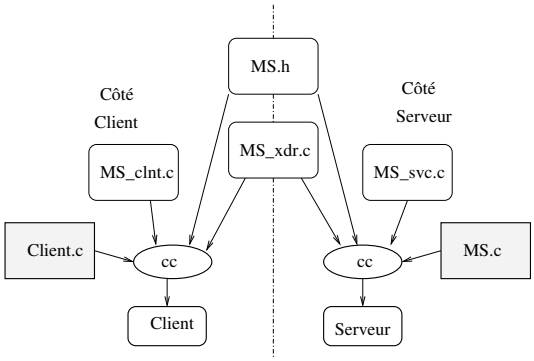

FIG. 13 – Génération des exécutables

Phase de génération

La figure (12) illustre la phase de génération de différents fichiers sources pour faciliter l'intégration des RPC's dans les applicatifs. Le compilateur de langage IDL (ici *rpcgen*) engendre à partir du texte IDL contenu dans le fichier $MS.x$ les talons d'appel pour le côté client dans un fichier $MS\_clnt.c$ et d'acceptation pour le côté serveur dans un $MS\_svc.c$. Les routines de conversion sont engendrées dans le fichier $MS\_xdr.c$. Un fichier $MS.h$ contient enfin des définitions globales.

Phase de production des exécutables

À partir des fichiers engendrés par la phase précédente d'une part et d'autre part des fichiers sources applicatifs (client et serveur), deux programmes exécutables vont pouvoir être engendrés selon le schéma de la figure (13). Le programme exécutable *client* sera compilé en utilisant le fichier $MS.h$ et édité avec le résultat de la compilation des routines $MS\_xdr.c$ et $MS\_clnt.c$. Le programme exécutable *serveur* sera lui-aussi compilé avec le fichier d'inclusion $MS.h$ et édité avec le résultat de la compilation des routines $MS\_xdr.c$ et $MS\_svc.c$.

2.3 Conclusion

La mise en œuvre d'applications réparties s'appuyant sur les services de communication de type flots de données et/ou de type procédures à distance est aujourd'hui largement répandue.

En particulier, des environnements orientés objets ont été proposés. L'OMG a défini la spécification CORBA (Common Object Request Broker Architecture) définissant l'interface et les fonctionnalités d'un environnement d'exécution répartie fondé sur la notion d'objets accessibles à distance. Dans cette approche, l'accent est mis sur l'hétérogénéité de l'architecture répartie support.

De façon similaire, l'environnement de développement Java propose la notion de RMI (Remote Method Call) permettant de faire communiquer des objets Java à distance. Dans cette approche, c'est la portabilité, la facilité de déploiement et la mobilité du code qui sont particulièrement mis en avant.

Références

[But97] David R. Butenhof. *Programming with POSIX threads*. Professional Computing. Addison-Wesley Publishing Company, 1997.

[CS93a] D.E. Comer and D. L. Stevens. *Internetworking with TCP/IP : Client-server programming and applications BSD socket version*, volume III. Prentice-Hall International Inc., 1993.

[CS93b] D.E. Comer and D. L. Stevens. *Internetworking with TCP/IP : Design, implementation and internals*, volume II. Prentice-Hall International Inc., 1993.

[CS93c] D.E. Comer and D. L. Stevens. *Internetworking with TCP/IP : Principes, protocols and architecture*, volume I. Prentice-Hall International Inc., 1993.

[KSS96] S. Kleiman, D. Shah, and B. Smaalders. *Programming with threads*. SunSoft Press. Prentice Hall, 1996.

[PS83] J-L. Peterson and A. Silberschatz. *Operating System Concepts*, pages 131–187. Addison-Wesley Publishing Company, 1983.

[Tan03] A. Tanenbaum. *Systèmes d'exploitation, 2˚Edition*, pages 201–279. Pearson Education France, 2003.