Systèmes et Applications Répartis

Michel Daydé
ENSEEIHT-IRIT
2 rue Camichel
31071 TOULOUSE CEDEX FRANCE
dayde@enseeiht.fr

January 26, 2005

Contents

1 Introduction au système réparti

1.1 Pourquoi l'informatique répartie et le calcul parallèle?

- Puissance croissante des stations de travail et des PC
- Apparition de processeurs dédiés : image, parole, ...
- Capacités des moyens de stokage de l'information croissantes
- Disponibilités d'outils facilitant l'accés à des ressources dispersées sur un réseau
- Développement d'applications et de services tirant profit de ces évolutions pour améliorer la gestion des données et la performance des traitements.
- Frontière entre multiprocesseurs, réseaux de machines, clusters de machines floue
- Description des concepts utilisés par les applications réparties : répartition ou distribution, interopérabilité, modèle client-serveur, ...

1.1.1 Pourquoi des traitements parallèles (exemples dans le domaine du calcul scientifique)

- Besoins de calcul non satisfaits dans beaucoup de disciplines
- Objectif actuel :
  supercalculateur 1 Terabytes / 1 Teraflops
- Performance uniprocesseur proche des limites physiques
  Cycle $\approx$ 1 ns $\leftrightarrow$ 2 GFlops (avec 2 flop/s.)
- Calculateur 1 TFlops $\rightarrow$ 50 processeurs
  $\rightarrow$*calculateurs massivement parallèles*

1.2 Architectures parallèles : Multiprocesseurs, clusters, réseaux de machines

1.2.1 Comment accroître la vitesse de calcul ?

- Technologies plus rapides
  TTL Schottky $\rightarrow$ ECL $\rightarrow$ AsGa
- Problèmes :
  - Conception des puces
  - Refroidissement
  - Reste insuffisant

  1 ns = temps pour qu'un signal parcoure 30 cm de cable
- Temps de cycle 1 ns $\leftrightarrow$ 2 Gigaflops (avec 2 flops)

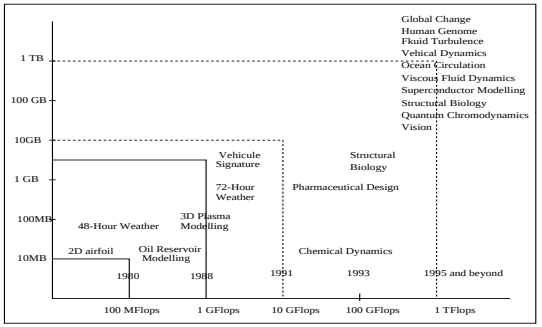


Figure 1: Grand challenge problems

Machine	Problème de petite taille	Problème Grand Challenge
TFlops computer	2 secondes	10 heures
CM2 64K	30 minutes	1 an
CRAY-YMP-8	4 heures	10 ans
ALLIANT FX/80	5 jours	250 ans
SUN 4/60	1 mois	1500 ans
VAX 11/780	9 mois	14,000 ans
IBM AT	9 ans	170,000 ans
APPLE MAC	23 ans	450,000 ans

Table 1: Vitesse de certains calculateurs sur un problème Grand Challenge (d'après J.J. Dongarra [9])

1.2.2 Parallélisme

- Exécution simultanée de d'instructions à l'intérieur d'un programme
- A l'intérieur d'un processeur :
  - micro-instructions
  - traitement pipeliné
  - recouvrement d'instructions exécutées par des unités distinctes

transparent pour l'utilisateur

(géré par le compilateur ou durant l'exécution)
Entre des processeurs distincts:

- suites d'instructions différentes exécutées

→ *synchronisations implicites (compilateur) ou explicites (utilisateur)*

1.2.3 Comment obtenir de hauts débits mémoire ?

- L'accés aux données est un problème crucial dans les calculateurs actuels
- Accroîssement de la vitesse de calcul sans accroître le débit mémoire → goulot d'étranglement

  MFlops plus faciles que MOctets pour débit mémoire

- Temps de cyle processeurs → 1 GHz (1 ns)
  Temps de cycle mémoire → < 20 ns SRAM
  < 60 ns SRAM
- Solutions :
  - Plusieurs chemins d'accés entre mémoire et processeurs
  - Plusieurs modules mémoire accédés simultanément (entrelaçage par exemple)
  - Accés mémoire pipelinés
  - Mémoire organisé hiérarchiquement
- La façon d'accéder aux données peut affecter la performance :
  - Minimiser les défauts de cache
  - Minimiser la pagination mémoire
  - Améliorer le rapport références à des mémoires locales/ références à des mémoires à distance

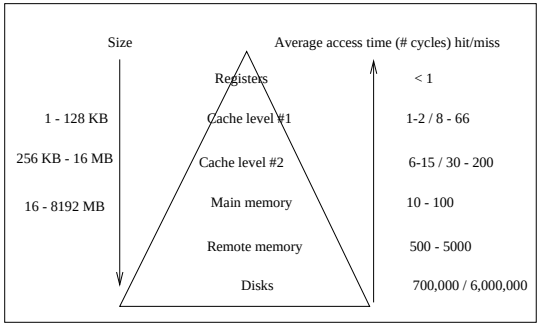


Figure 2: Exemple de hiérarchie mémoire

1.2.4 Conception mémoire pour grand nombre de processeurs ?

- Comment 100 processeurs rapides peuvent avoir accés rapide à données rangées dans mémoire partagée centrale (technologie, interconnexion, prix ?)

  → Solution à coût raisonnable :

  mémoire physiquement distribuée

  (chaque processeur a sa propre mémoire locale)

2 solutions :

- mémoires locales globalement adressables :

  Calulateurs à mémoire partagée virtuelle

- transferts explicites des données entre processeurs avec échange de messages

  Programmation en mode messages

- Scalabibité impose :
  - augmentation linéaire du débit de la mémoire locale avec la vitesse du processeur
  - augmentation du débit des communication inter-processeurs avec le nombre de processeurs
- Rapport coût/performance → mémoire distribuée et bon rapport coût/performance sur les processeurs

1.2.5 Architecture des multiprocesseurs

Nombre élevé de processeurs → mémoire physiquement distribuée

Org. logique	Org. physique	
	Partagée	Distribuée
Partagée	multiprocesseurs à mémoire partagée	espace d'adressage global au dessus de messages mémoire partagée virtuelle
Distribuée	émulationde messages (buffers)	échange de messages

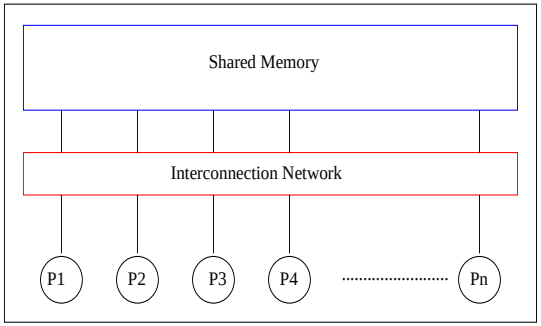


Figure 3: Exemple d'architecture à mémoire partagée

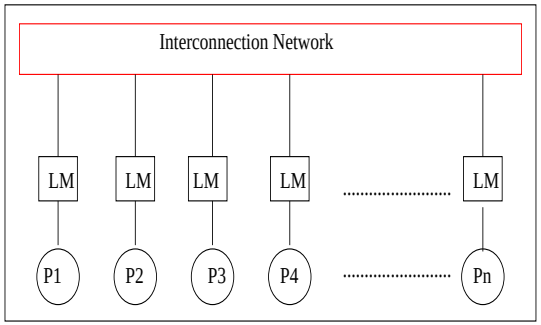


Figure 4: Exemple d'architecture à mémoire distribuée

1.2.6 Clusters de processeurs

- Plusieurs niveaux de mémoire et de réseaux d'interconnexion → temps d'accés non uniforme
- Mémoire commune partagée par un faible nombre de processeurs (SMP)
- Eventuellement des outils de programmation distincts (transferts de message entre les clusters, ...)
- Exemples : HP, CONVEX, SGI, SUN, Clusters de PC, ...CONVEX Exemplar

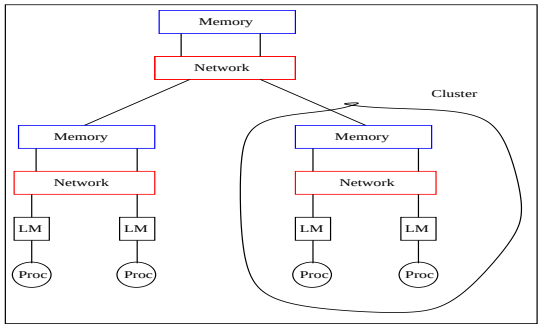


Figure 5: Exemple d'architecture "clusterisée"

1.2.7 Réseaux de Calculateurs

- Evolution du calcul centralisé vers un calcul distribué sur des réseaux de calculateurs
  - Puissance croissante des stations de travail
  - Intéressant du point de vue coût
  - Processeurs identiques sur stations de travail et MPP
- Calcul parallèle et calcul distribué convergent :
  - modèle de programmation
  - environnement logiciel : PVM, MPI, ...
- Performance effective peut varier énormément sur une application
- Performance très dépendante des communications (débit et latence)
- Réseaux :

  - Ethernet : 10 Mbits
  - SOCC : 220 Mbits
  - FDDI : 100 Mbits

- Hétérogène / homogène
- Plutôt orienté vers un parallélisme gros grain
- Charge du réseau et des calculateurs peut varier pendant l'exécution

Equilibrage des traitements ?

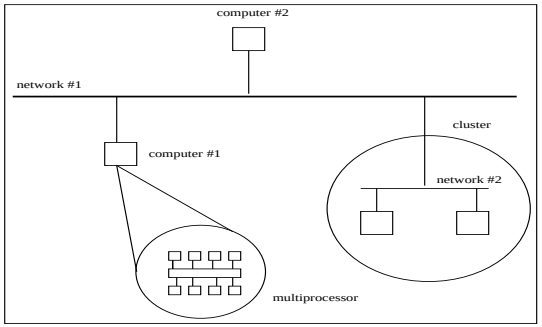


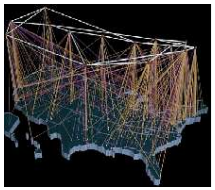
Figure 6: Exemple de réseau de calculateurs

1.2.8 Multiprocesseurs vs réseaux de machines

- Systèmes répartis (réseaux de machines) : communications relativement lentes et systèmes indépendants
- Systèmes parallèles (architectures multiprocesseur) : communications plus rapides (réseau d'interconnexion plus rapide) et systèmes homogènes en général

Il y a convergence entre ces deux classes d'architectures et la frontière est floue :

- clusters et clusters de clusters
- des systèmes d'exploitation répartis comme MACH et CHORUS savent gérer les deux
- versions de UNIX multiprocesseur
- souvent mêmes environnements de développement



- Accés transparent à des ressources sur Internet : capacités de traitement, logiciels d'expertise, bases de données, visualisation, instruments de mesure, ...
- Plus rarement traitements parallèles sur une grille
- Exemples : NetSolve, Globus, NEOS, Ninf, Legion, UNICORE, EUROGRID, DATAGRID, ...

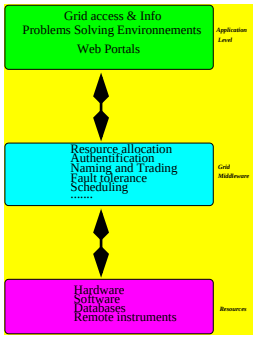


Figure 7: Grid software / hardware layers

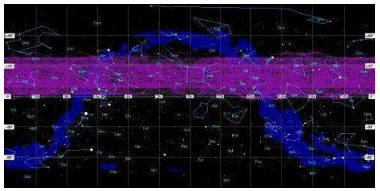


Figure 8: Peer-to-Peer : SETI@home

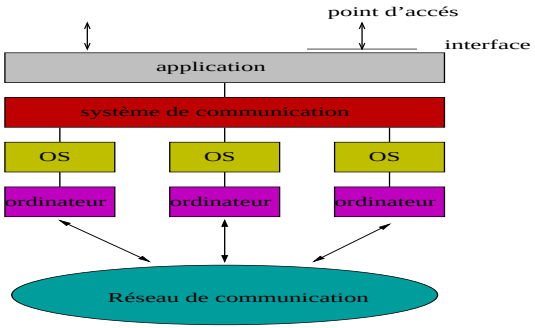


Figure 9: Composants d'un système informatique

1.2.9 Grid Computing: motivations

- Use 500,000 PCs to help searching for extraterrestrial intelligence
- Data and Signal processing analysis
- Computers download a MB dataset from Arecibo Radio Telescope when they are idle
- Results are sent back to SETI team
- In average *55 TFlops*
- Gives inspiration to a number of compagnies

Google (from J. Dongarra)

- 2000 queries per second (150×10^6 per day)
- 100 countries
- 3×10^9 documents in the index
- 15,000 Linux systems in 6 data centers
- Each query $\Leftrightarrow$ *eigenvalue problem* on a transition probability matrix (1 between page i and j means there is an hyperlink from i to j)

1.3 Systèmes informatiques

1.3.1 Notion d'interface

- Ensemble des fonctions accessibles aux utilisateurs d'un service
- Chaque fonction est définie par :

  - son format et sa syntaxe : mode d'emploi
  - sa spécification : son effet
- Principe de base : séparation réalisation et interface
  - Description de l'interface indépendante de réalisation
  - Facilite la portabilité (passage à une autre implantation du service)

1.3.2 Exemples de services

- Informations : bulletin météo, infos, ...
- Moteur de recherche sur le Web
- Courrier électronique
- Forums de discussion (news)
- Utilisation d'un calculateur distant (telnet)
- Commerce électronique
- ...

1.3.3 Rôle d'un système d'exploitation

Fournit une interface avec le matériel :

- Dissimule détails de mise en œuvre
- Dissimule limitations physiques (taille mémoire, #processeurs), partage les ressources

  → machine virtuelle
- Gestion des processus et de la mémoire
- Gestion des communications et des accés (protection, droits d'accés)

1.3.4 Interfaces d'un système d'exploitation

En général deux interfaces :

- **API** Application Programming Interface
  - utilisable à partir des programmes s'exécutant sous le système
  - ensemble d'appels systèmes
  - en C pour UNIX
- Interface utilisateur ou commande
  - utilisable par individu connecté (textuelle ou graphique)
  - ensemble de commandes :
    - textuelle : e.g. **rm ***
    - graphique : e.g. déplacer un fichier avec la souris vers la corbeille

1.4 Applications réparties

- Données ou traitements répartis ou distribués : la mise en œuvre d'une opération nécessite d'utiliser plusieurs machines
- Traitement coopératif : dialogue entre deux applications pour réaliser une tâche
- Interopérabilité : capacité des systèmes à partager des données ou des traitements via des interfaces standards (systèmes ouverts aptes à communiquer dans un environnement hétérogène).
- Evolution au cours du temps
  - *échange* : des applications sur des systèmes différents s'envoient des informations (e.g. fichiers)
    ↓
  - *partage* : les ressources sont accessibles directement par plusieurs machines (e.g. partage de fichiers)
    ↓
  - *coopération* : les machines coopèrent en vue de réaliser un traitement

1.4.1 Classes d'applications réparties

- Coordination d'activités
- Communication et partage d'information : bibliothèques virtuelles
- Travail coopératif :
  - Edition coopérative
  - Téléconférence
  - Ingénierie coopérative
- Applications Temps Réel :
  - Contrôle de procédés
  - Systèmes embarqués (avionique, ...)
  - Localisation de mobiles
- Services grand publics :
  - Presse électronique
  - Télévision interactive
  - Commerce électronique, ...

1.4.2 Objectifs des systèmes répartis

- Optimiser l'utilisation des ressources :
  - puissance de calcul
  - capacités de stockage (mémoire, disque)
  - capacités graphiques
  - périphériques (imprimantes, ...)
- Simplifier le travail de l'utilisateur :
  - amélioration des performances par répartition des données et des traitements
  - offre de noouveaux services
  - amélioration du confort d'utilisation
- Avantages :
  - partage et optimisation des ressources
  - nouvelles fonctionnalités et amélioration des performances
  - souplesse et disponibilité
- Inconvénients
  - dépendance aux performances et à la disponibilité du réseau
  - problèmes de sécurité (on devient souvent dépendant de la machine la moins sûre)

1.4.3 Répartition vs parallélisme

Répartir les traitements (ou les distribuer) sur les machines les plus adaptées n'implique pas que les traitements seront effectués en parallèle.

Mais répartir les traitements est aussi une façon de les paralléliser.

1.4.4 Transparence

- Possibilité d'accéder à des ressources ou à des services sans connaître leur localisation
- Pour le développeur d'applications : possibilité d'utiliser les mêmes primitives d'accés où que se situent le service ou la ressource désiré
  vision d'une seule interface et d'un seul ordinateur
- Transparence = vision unifiée d'un système au lieu d'une collection d'objets indépendants

Plusieurs types de transparence :

- transparence d'accès : des opérations identiques permettent l'accès à des objets locaux ou distants

- transparence de localistion : objets accessibles sans avoir à connaître leur localisation physique
- transparence sur la concurrence des accès : plusieurs utilisateurs doivent pouvoir accéder simultanément à des données sans effet indésirable
- transparence sur la duplication : des données ou des objets peuvent être dupliqués pour améliorer la performance ou la disponibilité sans que cela soit visible
- transparence – tolérance – aux pannes : possibilité de terminer un traitement même si un composant matériel ou logiciel tombe en panne
- transparence aux reconfigurations : le système peut modifier dynamiquement sa configuration (e.g. ajout de ressources) pour améliorer ses performances

1.4.5 Modèle Client-Serveur

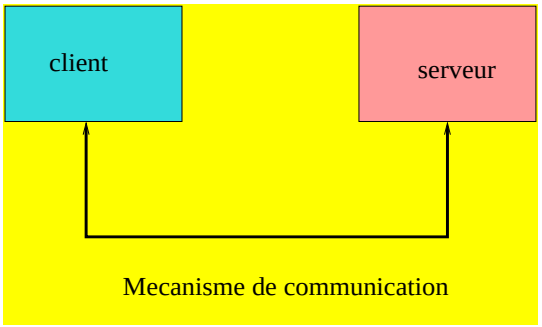


Figure 10: Modèle Client-Serveur

- *Client* : demande accès à un service ou à une ressource
- *Serveur* : entité qui rend le service ou attribue la ressource
- Peuvent être sur la même machine (communication locales) ou sur des machines distantes (mécanismes de communication réseau)
- Par exemple client et serveur sont deux processus UNIX communiquant par des IPC (interprocess communication) local ou réseau
- Client et serveur ne jouent pas des rôles symétriques
- *Serveur* : s'initialise et se met en attente de requêtes de clients éventuels

- *Client* : en général lancé interactivement, envoie des requêtes au serveur
- Exécution d'une requête par le serveur : peut impliquer un dialogue avec le client, ensuite le serveur se remet en attente d'autres requêtes
- Deux types de processus serveurs :
  - *serveurs itératifs* : le processus serveur traite lui-même la requête; viable si traitement rapide ou peu de clients
  - *serveurs parallèles* : le processus serveur invoque un autre processus pour traiter la requête du client (**fork()** par exemple), après création le processus serveur ne bloque pas sur la fin d'exécution du fils et se remet en attente.
- A chaque serveur on associe une adresse de service : requêtes émises vers cette adresse
- *serveurs sans état* : pas de conservation d'informations sur les clients au contraire des *serveur avec état*.
- En cas de rupture de communication : reprise plus simple avec des serveurs sans état mais fonctionnement aléatoire.

1.5 Outils disponibles sous UNIX

TCP/IP → ensemble d'outils :

- **Sockets** → Emmanuel Chaput
- Bibliothèque TLI (Transport Level Interface)
- **NFS** (Network File System)
- RFS (Remote File Sharing)
- X Window
- **XDR** (eXternal Data Representation)
- **RPC** (Remote Procedure Call) de SUN
- NCS (Network Computing System)

2 Rappels UNIX

2.1 Protocoles de transport

2.1.1 Notions de protocole et d'interface

Exemple de la requête sur le web → divers niveaux d'échange entre le client et le serveur

- Niveau application : client clique sur un lien, serveur renvoie une page Web
- Niveau des messages : client envoie un message contenant une URI, serveur renvoie un fichier HTML
- Niveau de la transmission des bits : envoi de paquets où chaque bit est transmis comme un signal électrique sur une ligne
- Chaque niveau s'appuit sur les niveaux inférieurs

Notions de "Protocole" et "Interface" sont une représentation de ce mode de fonctionnement

- Interface (d'un service) : fonctions logicielles ou matérielles et règles d'accès pour utiliser ce service.
- Protocole : conventions définissant les échanges entre les entités coopérant pour réaliser un service.
- Relations entre protocoles et interfaces :
  - Interface définit l'accès au service, protocole définit sa réalisation
  - Construction d'un protocole souvent basée sur des protocoles de niveau inférieur en accédant à leurs interfaces.

protocoles en couches

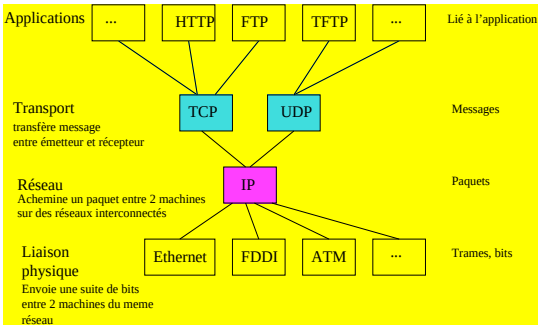


Figure 11: Protocoles de l'Internet

- HTTP : HyperTexte Transfer Protocol → Web
- TFTP, FTP : Trivial File Transfer Protocol
- TCP : Transmission Control Protocol (transport en mode connecté)

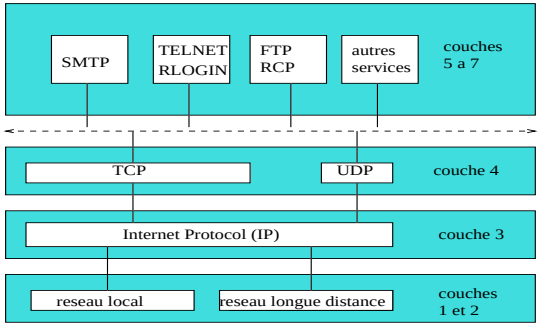


Figure 12: Les protocoles et services TCP/IP

- UDP : User Datagram Protocol (transport en mode non connecté)
- IP : Internet Protocol (interconnexion de réseaux, routage)
- FTP, RCP : transferts de fichiers
- TELNET, RLOGIN : terminal virtuel
- SMTP : messagerie

2.1.2 Protocoles de transport

- Fonctions
  - Assurer la communication entre processus
  - Protocoles de "bout en bout" (pas de vision des sites intermédiaires)
  - Les applications ne voient pas les protocoles de niveau inférieur
- Problèmes
  - Protocole de transport utilise protocole de réseau (IP) qui
    - perd des messages (ou les délivre 2 fois)
    - ne respecte pas l'ordre d'émission
    - limite la taille des messages
- Protocole de transport doit garantir aux applications :
  - Délivrance des messages
  - Respect de l'ordre d'émission
  - Pas de limitation de taille
  - Synchronisation et contrôle de flux
- Exemples : **UDP** minimal, **TCP** avec garanties, **RPC** intégré à un langage

2.1.3 Identification des processus

- Protocole de transport fait communiquer des processus sur des hôtes différents → les identifier
- Identification par numéro interne (**pid** UNIX) inadéquate :
  - Liée à un OS particulier
  - Identifie un processus individuel alors que l'on a besoin d'identifier une classe de processus équivalents (rendant un service) : processus peut disparaître et être remplacé par un autre

Solution

- Identification indirecte au moyen de portes
- Porte = point d'entrée prédéfini sur une machine, identifié par un numéro de prote codé sur 16 bits
- Messaga arrivant à une porte est reçu par le processus associé à la porte
- Conventions d'usage des portes pour les services standard (numéro < 1024) *processus identifié par (adresse IP hôte, numéro porte)*

2.1.4 Protocole UDP

- Protocole de transport "minimal"
  - Simple transposition de IP au niveau transport
  - Communication entre processus en mode non connecté (les messages sont indépendants les uns des autres)
  - Pas plus de garantie que IP
- Format :
  - En-tête : numéro porte d'origine (16 bits), numéro porte destination (16 bits), contrôle d'erreur (16 bits), taille (16 bits)
  - données
- Propriétés :
  - Simple
  - Mais garanties minimales
  - Réalisation d'applications peu exigeantes, construction de protocoles plus élaborés

2.1.5 Protocole TCP

- Permet de transmettre un flot d'octets bidirectionnel entre un processus émetteur et un processus récepteur
- Propriétés :
  - Fiabilité (garantie de livraison dès qu'une liaison physique existe)
  - Préserve l'ordre d'émission
- Contrôle de flux (récepteur peut demander à l'émetteur de réduire son débit)
- Contrôle de gestion (limitation du débit de l'émetteur pour éviter de saturer le réseau)
- NB : contrôle de flux → capacité du récepteur, contrôle de congestion → capacité du réseau
- Fonctionne en mode "connecté" organisé en 3 phases :
  - Phase de connexion : établir une liaison entre les processus
  - Phase de commnunication : échange de données sur la liaison établie
  - Phase de déconnexion : déconnecter les 2 processus en supprimant la liaison
- Connexion et déconnexion utilisent un mini-protocole : demande - accord - accord confirmé
- Format des données TCP :
  - En-tête : porte origine (16 bits), porte destination (16 bits), numéro séquence (32 bits), ...
  - données
- Utilisation d'un tampon d'émission pour l'émetteur et d'un tampon de réception pour le récepteur
- Aspects importants : **sécurité + compression de données**

2.2 Fichiers et commandes UNIX utiles

Fichiers de configuration :

- /etc/hosts : informations sur les machines du réseau local auquel appartient le système. Exemple :

  147.127.18.114 wanda.enseeiht.fr wanda.enseeiht.fr wanda

  adresse, nom officiel, liste d'alias

  Organisé hiérarchiquement si on dépasse le cadre du réseau local.

- /etc/networks : base de données des réseaux connus (nom officiel du réseau, adresse Internet, list d'alias)
- /etc/services : liste des ervices Internet connus (nom, numéro de port et protocole, liste d'alias)
- ...

2.2.1 Commandes d'administration

Permettent d'obtenir des informations générales sur l'état du réseau

- hostid, hostname : donnent respectivement l'adresse Internet et le nom officiel de la machine.
- ruptime : état des machines du réseau local
- ping : permet de tester si une machine est active

```
ping julia
```

- netstat : informations sur l'activité réseau du système
- traceroute <nom domaine> : détermination du chemin suivi dans le réseau

2.2.2 Processus démons

Certains services UNIX standards (**telnet** ou **ftp** par exemple) → nécessitent l'existence démons sur la machine distance

- Super-démon inetd : démon principal qui a la charge de créer automatiquement le serveur correspondant à un service requis (/etc/inetd), détruit après le service rendu.
- /etc/inetd.conf : utilisé par **inetd** à son lancement pour connaître les ports à écouter.

Lignes du fichier :

- nom du service,
- type de socket,
- protocole sous-jacent,
- option wait/nowait
- nom d'utilisateur qui sera propriétaire du démon associé au service
- référence absolue du fichier programme
- liste de paramètres


```
ftp stream tcp nowait root /usr/sbin/tcpd in.ftpd -l -a
telnet stream  tcp  nowait  root    /usr/sbin/tcpd in.telnetd
...
shell stream tcp nowait root /usr/sbin/tcpd in.rshd
...
```

Démons de services standard

- rwhod : dialogues avec ses homologues sur les autres systèmes et maintient la base de données utilisée par rwho et ruptime. Pas toujours lancé.
- telnetd, rlogind : permettent le "login" sur depuis un autre système.
- ftpd, tftpd : serveurs des protocoles ftp et tftp de transfert de fichiers.
- rshd : utilisé pour l'existence de commandes distantes par rsh ou rcmd) ou la copie de fichiers par rcp.

2.2.3 Commandes de services standard

- telnet [hote [port]] : connexion avec le port TCP (optionnel) d'une machine
- ftp [-v] [-i] [-n] [hote] : transfert de fichiers entre site local et distant. Nécessite un login et un password (exception notable : anonymous). Plusieurs commandes sont disponibles :
  - **get, mget** : fichier site distant → local
  - **put, mput** : fichier site local → distant
  - **cd** : changement répertoire site distant
  - **lcd** : changement répertoire site local
  - **ls** : ls sur site distant
  - ...

2.2.4 Commandes de services UNIX

- Configuration du réseau : en l'absence d'une distribution des fichiers systèmes (cf. NFS plus loin), pb d'identification des usagers entre les diverses machines du réseau :
  - Solution globale par administrateurs systèmes (/etc/hosts.equiv)
  - Au niveau de chaque utilisateur (.rhosts chez chaque utilisateur)
- /etc/hosts.equiv : liste de machines équivalentes à la machine locale i.e. login machine distante identique sur machine locale.
- .rhosts : Permet à un utilisateur d'autoriser des usagers d'autres machines à s'identifier sous son nom. Exemples :

julia dayde
+ dayde

- rwho : liste des utilisateurs logés sur les machines du réseau
- rcp : remote copie. Exemple :
  rcp file myhost:otherfile

Exécution de commandes distantes
rsh hote [-l user] commande . Exemple :
rsh julia ls

Ne marche que si sur julia on a + **dayde** ou **ma_machine dayde** dans **.rhosts**.

2.3 Communication entre utilisateurs sous UNIX

Popularité de UNIX : possibilité d'échanger des fichiers ou de dialoguer par messagerie ou forums.

Communication facilitée grâce à l'interconnexion des machines via Internet

- **Mécanismes d'adressage**
  - Démon **sendmail**
  - Adresses **uucp**
  - Domaine de noms Internet
- **Courrier**
  - Boîtes aux lettres
  - Envoi de courrier
  - Lecture du courrier
  - Autres commandes

3 Communication entre processus sous UNIX

3.1 Introduction

- Communications entre processus sur une même machine *intra-UNIX* :
  - Tubes (pipes) – nommés –
  - Files de messages
  - Mémoire partagée et sémaphores
  - Sockets et interface TLI
  - ...
- Entre processus sur des systèmes distants *inter-UNIX* :
  - Sockets
  - TLI et streams
  - RPC
  - ...

3.2 Exemple introductif : client - serveur ([14] et [13])

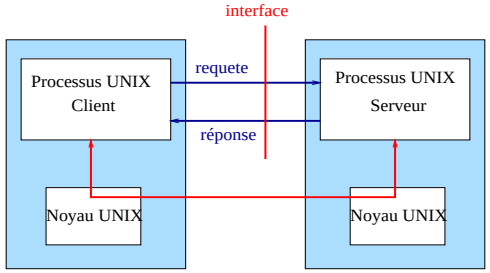


Figure 13: Client/Serveur avec communications inter-UNIX.

- *Client* (processus) demande l'exécution d'un service (spécifié par interface)
- *Serveur* (processus) réalise le service
- En général client et serveur sur deux machines distinctes

Intérêt du schéma client-serveur

- Bien structuré :
  - Fonctions bien identifiées
  - Séparation entre interface du service et réalisation (client ne connaît que l'interface)
  - Client et serveur peuvent être modifiés (remplacés) indépendamment
- Sécurité
  - Client et serveur s'exécutent dans des domaines différents
- Gestion des ressources :
  - Serveur peut être partagé entre plusieurs clients

Implantation du modèle client-serveur - 1

- Client et serveur = processus
- Communiquant par messages
  - Requête = paramètres d'appel, spécification du service requis
  - Réponse = résultats, flags d'exécution ou d'erreur
- Identification des procssus client et serveur

- Numéro de porte
- Identification symbolique → RPC

Implantation du modèle client-serveur - 2

- Réalisation en utilisant les protocoles de transport **bas niveau** ou **haut niveau**
- Avec protocoles de bas niveau :
  - Utilisation de fonctions de communications fournies par l'OS et directement construites sur le protocole de transport
  - Exemple : **sockets** UNIX
    - Mode non connecté (UDP)
    - Mode connecté (TCP)
- Avec protocoles de haut niveau :
  - Utilisation d'un logiciel spécialisé (interface entre systèmes de communication et applications)
  - Exemples :
    - *Librairies de transferts de messages* → **PVM, MPI, ...**
    - *Langage de programmation* → **appel de procédure à distance (RPC)**
    - *Objets répartis* → **appel de méthodes, création d'objets à distance**

4 Sockets → Emmanuel Chaput

5 eXternal Data Representation (XDR)

5.1 Introduction

- Norme IEEE très répandue mais pas universelle
- Echange de données binaires entre systèmes pose souvent problème
- Alignement, stockage ("big endian", "small endian") peuvent être différents selon les architectures et les compilateurs
- XDR (introduit par SUN en même temps que NFS) → représentation standard des données pour les échanges entre systèmes hétérogènes

Exemple :

- Lors de la programmation des sockets dans le domaine Internet
- Utilisation d'une représentation standard des entiers courts ou longs pour dsigner les ports UDP ou TCP ou les adresses IP
- Passage d'une représentation réseau à une représentation interne avec **ntohl** ou **ntohd** ou transformation inverse avec **htonl** ou **htons**.

5.2 Fonctionnalités

Services offerts :

- Permet de décrire et de représenter des données indépendamment de la machine
- Alternatives à un format commun tel XDR :
  - Transmettre des données en ASCII : lourd, accroissement taille des données, perte de précision éventuelles due aux conversions
  - Convertir au cas par cas → autant de programmes de conversion que de formats (IEEE universel sur les stations de travail)

Conventions :

- 1 format pour la représentation des entiers : 32 bits "big-endian" (octet de poids fort dans + petite adresse)
- codage IEEE pour les réels
- Longueur des données toujours multiple de 4 octets (ajoût éventuek de '0')
- Données non typées → émetteur et récepteur doivent connaître le type des données à échanger (évite un codage du type)
- Inconvénient : transcodages qui ne sont pas toujours nécessaires.

- Mais coût négligeable par rapport au temps de transmission et traitement systématique.

Encodage et décodage

- Emetteur encode les données – sérialisation – grâce aux primitives de la bibliothèque XDR.
- Cette opération crée un flot d'informations - **flot XDR** - constitué des représentations XDR des valeurs.
- Récepteur décode les données.
- XDR utilisable en mémoire, sur des fichiers, ou à travers le réseau.
- Bibliothèque XDR = ensemble de fonctions C.

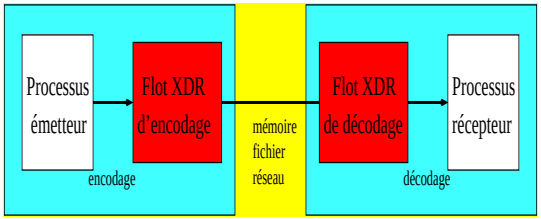


Figure 14: Utilisation de XDR.

5.3 Flot et filtre XDR

- *Flot XDR* : suite de données au format XDR
- *Filtre XDR* : procédure encodant ou décodant un certain type de données (entier, réels, ...)
- Les filtres XDR lisent ou écrivent des données dans les flots accédés par l'émetteur et le récepteur.
- Deux groupes de primitives XDR :
  - Création et manipulation de flots XDR
  - Conversion de données et transfert dans ces flots

Flots XDR

- Trois types de flots XDR :
  - Flots standard d'entrée-sortie (écriture / lecture de données sur un fichier)

  - Flots en mémoire (codage de données en mémoire)
  - Flots d'enregistrement (permet de délimiter les données en enregistrements)

- Pointeur sur une structure XDR ("handle"). Défini dans <rpc/xdr.h> : donne des informations sur les opérations effectuées sur le flot XDR
- **XDR_ENCODE** : encodage
- **XDR_DECODE** : décodage
- **XDR_FREE** : libération de l'espace mémoire alloué par une opération de décodage (il y a plus simple).
- Avec XDR_ENCODE, les données codées en XDR par le filtre sont écrites dans le flot associé.
- Avec XDR_DECODE, les données décodées au format machine sont lues dans le flot associé.

5.3.1 Flot standard d'entrée-sortie

- Flot permettant de lire ou d'écrire des données XDR sur un fichier.
- Création de ce flot avec xdrstdio_create()

```
void xdrstdio_create ( xdr_handle, file, op )
  xdr *xdr_handle    ;   /*handle*/
  FILE *file         ;   /*Pointeur sur un fichier ouvert*/
  enum xdr_op op     ;   /*XDR_DECODE ou XDR_ENCODE*/
```

- Allouer de la mémoire pour le handle XDR.
- Flot unidirectionnel, pas de retour d'erreur
- Utilisable pour lire / écrire des données binaires via NFS.

Exemple (tiré de [14])
Utilisation d'un fichier /tmp/fixdr pour échanger un entier et un flottant entre deux processus : client → serveur.

```
/* Client.c ****************************/
/* Encode 1 entier et 1 flottant        */
#include <stdio.h>
#include <rpc/rpc.h>
#define FIC "/tmp/fixdr"
main()
{
```

```
   FILE *fp ;           /* FILE pointer */
   XDR  xdrs ;          /* handle XDR   */
   long i=10 ;          /* entier       */
   float x=4.5          /* flottant     */

/* ouverture fichier en ecriture        */
   fp = fopen(FIC, "w") ;
/* Creation flot XDR d'encodage         */
   xdrstdio_create( &xdrs, fp, XDR_ENCODE ) ;
/* Ecriture d'un entier                 */
   xdr_long( &xdrs, &i ) ;
/* Ecriture d'un flottant               */
   xdr_float( &xdrs, &x ) ;
   close( fp ) ;
   exit(0)     ;
}

/* Serveur.c ****************************/
/* Decode 1 entier et 1 flottant        */
#include <stdio.h>
#include <rpc/rpc.h>
#define FIC "/tmp/fixdr"
main()
{
   FILE *fp ;           /* FILE pointer */
   XDR  xdrs ;          /* handle XDR   */
   long i ;             /* entier       */
   float x ;            /* flottant     */
/* ouverture fichier en lecture         */
   fp = fopen(FIC, "r") ;
/* Creation flot XDR de decodage        */
   xdrstdio_create( &xdrs, fp, XDR_DECODE ) ;

/* Lecture d'un entier                  */
   xdr_long( &xdrs, &i ) ;
/* Lecture d'un flottant                */
   xdr_float( &xdrs, &x ) ;
   close( fp ) ;
   exit(0)     ;
}
```

5.3.2 Flot mémoire

- Flot pour codage de données en mémoire :

```
void xdrmem_create( xdr_handle, addr, size, op )
   XDR  *xdr_handle    ;     /*handle                */
   char *addr          ;     /*adresse memoire       */
   int   size          ;     /*taille memoire        */
   enum wdr_op op      ;     /*XDR_ENCODE ou XDR_DECODE */
```


- Données XDR lues ou écrites en mémoire à partir de addr pour une taille de size. Espace suffisamment grand pour données XDR (et multiple de 4) → utilisation macro RNDUP.
- Taille mémoire insuffisante → échec
- Solutions :
  - Taille mémoire avec marge de sécurité
  - Connaître le codage XDR et calculer l'espace nécessaire
  - Augmenter la taille mémoire en cas d'erreur

5.3.3 Flot d'enregistrements

- Rangement dans des mémoires tampons des données échangées et encodées entre un processus émetteur et un processus récepteur.

```
void xdrrec_create( xdr_handle, sendsize, recvsize,
  iohandle, readproc, writeproc )
   XDR  *xdr_handle    ;     /*handle                */
   int  sendsize, recvsize ; /*taille des tampons    */
   char *iohandle ;          /*identificateur        */
   int (*readproc)() ;       /*procedure de lecture  */
   int (*writeproc)() ;      /*procedure d'ecriture  */
```

- sendsize, recvsize : taille de tampons en émission et en réception.
- iohandle : identifie la ressource permettant de lire ou d'écrire les données XDR c-à-d pointeur sur un fichier, socket TCP, ou tout objet permettant de ranger des donnes dans des mémoires tampons.
- readproc, writeproc : adresse de 2 procédures à définir. Tampon de réception vide → readproc() appelée par le filtre XDR pour lire les données. Quand le tampon d'émission est plein, appel de writeproc() par le filtre pour écrire les données.
- Par exemple

```
int readproc( iohandle, buf, nbytes )
   char *iohandle ;     /*identificateur ressource*/
   char *buf      ;     /*adresse buffer          */
   int nbytes     ;     /*taille buffer           */
```

iohandle peut être un pointeur sur un fichier, une socket TCP, où tout objet permettant de ranger des données dans des tampons.

- Flot d'enregistrements fonctionne en écriture ou en lecture en positionnant x_op du handle XDR.
- Il existe 3 procédures supplémentaires :
  - xdrrec_endofrecord() pour spécifier la fin d'un enregistrement (force l'écriture – flush – du tampon).
  - xdrrec_skiprecord() : à utiliser par le récepteur pour lire l'enregistrement suivant (en particulier avant première lecture).
  - xdrrec_eof() : le récepteur peut ainsi savoir si il reste des données à lire dans le tampon.

Macros relatives aux flots

- Obtention de la position courante :

```
int xdr_getpos( xdr_handle )
   XDR *xdr_handle ;    /*handle*/
```

- Positionnement dans le flot : renvoie vrai si le positionnement est possible

```
bool_t xdr_setpos( xdr_handle, pos )
   XDR *xdr_handle ;    /*handle*/
   int pos ;            /*position dans le flot*/
```

Exemple : calcul du nombre d'octets du codage XDR ([14])

```
#include <stdio.h>
#include <rpc/rpc.h>

main()
{
   XDR xdrs ;            /*handle XDR*/
   int pos1, pos2 ;      /*positions */

/*Position avant codage XDR*/
   pos1 = xdr_getpos( &xdrs ) ;
/*Codage XDR*/
   ..................
/*Position apres codage XDR*/
   pos2 = xdr_getpos( &xdrs ) ;
   printf("Nombre octets dans le flot XDR %d\n",pos2-pos1) ;
}
```

Filtres XDR

- Filtres : procédures ralisant les opérations de transcodage retounent TRUE si opération OK, FALSE sinon.

- Trois types de filtre : de base, composites,complexes.
- Filtres de base : dans la bibliothèque XDR associés aux types de base de C : char, int, long, float, double, void, enum
- Exemple :

```
bool_t xdr_int( xdr_handle, pobj )
   XDR *xdr_handle
   type *pobj
```

Filtres composites

- Disponibles dans la bibliothèque XDR
- Traitent les types de données composées (chaîne, tableau, ...)
- 2 premiers arguments idem ci-dessus, les autres arguments dépendent de la nature du filtre
- Types de données : string, opaque, bytes, vector, arrayu, union, reference, pointer

Filtres complexes

- Construits par l'utilisateur, combinaison des filtres précédents (e.g. filtre pour une structure)
- En pratique utiliser compilateur **RPCGEN** qui engendre un filtre avec seulement 2 paramètres

5.4 Gestion de la mémoire

- Pb : volume mémoire à réserver par un processus pour contenir des données décodées (exemple : longueur d'une chaîne de caractères)
- Solutions : réserver un buffer assez grand ou laisser XDR allouer la taille correcte → donner au filtre XDR un pointer NULL sur l'objet à décoder.
- Après décodage : libérer la mémoire par xdr_free()

```
void xdr_free( proc, objp )
   xdrproc_t proc ;    /* procedure qui a effectuee le filtre */
   char *objp     ;    /* pointeur sur l'objet decode         */
```

Exemple de gestion mémoire

```
#include <stdio.h>
#include<rpc/rpc.h>
"define FICHIER "/tmp/filexdr"
#define LGMAX 1024 /* Taille max de la chaine */

main()
{
   FILE *fp        /* FILE pointer */
   XDR  wdrs       /* handle XDR   */
   char *objp      /* pointeur sur chaine decodee */

/* Ouverture du fichier */
   fp = fopen( FICHIER, "r" ) ;

/* Creation du flot de decodage */
   xdrstdio_create( &xdrs, fp, XDR_DECODE ) ;
/* Lecture chaine, pointeur NULL car on ne connait pas
   la taille de la chaine decodee */
   objp = NULL ;
   xdr_string( &xdrs, & objp, LGMAX ) ;
/* Utilisation de la chaine obtenue */
   ......................
/* On libere la memoire alloueee par XDR */
   xdr_free( xdr_string, &objp ) ;
   close( fp ) ;
   exit(0) ;
}
```

5.5 Utilisation de XDR avec les sockets

- Combinaison de sockets et XDR nécessaires si Client et Serveur n'utilisent pas la même représentation des données
  - Flot mémoire avec les sockets UDP
  - Flot d'enregistrement avec les sockets TCP
- Exemple Echo d'une chaîne de caractères :
  - Client envoie au serveur un certain nombre de buffers de caractères et le serveur renvoie chacun des buffers en écho
  - La taille des buffers est envoyée au serveur avant la première émission.

Echo avec flot mémoire et sockets UDP [14]

```
/* Fichier gen.x : Description RPCGEN des donnees echangees */
typedef string st<16384>  /* Longueur de chaine maxi */

/* Fichier commun.h */
```



```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <errno.h>
#include <netdb.h>
/* Taille maxi d'echange entre client et serveur */
#define TAILLEMAXI 16384

/* Fichier soct.h */
#include <commun.h>
#include <sys/socket.h>
#include <netinet/in.h>
/* Numero de port utilise par les sockets */
#define PORTS 6258
/* Fichier client.c */
#include "soct.h"
clientipc()
{
   int sock ;                  /* Descripteur socket */
   struct sockaddr_in server ; /* Adresse serveur */
   struct sockaddr_in sclient ; /* Adresse client */
/* Creation socket */
   sock = socket( AF_INET, SOCK_DGRAM, 0 ) ;
/* Binding obligatoire en mode datagramme */
   bzero = ( &client, sizeof( sclient ) );
   sclient.sin_familly = AF_INET ;
   sclient.sin_addrs.s_addr = INADDR_ANY ;
   sclient.sin_port = htons(0) ;
   bind( sock, (struc sockaddr *) &sclient, sizeof( sclient ) ) ;
/* Affectation de la structure d'adresse du serveur */
   bzero = ( &server, sizeof( server ) );
   bcopy( (char *) hp->h_addr, (char *) &server.sin_addr, hp->h_lenght );
   server.sin_port = htons(PORTS) ;
   server.sin_familly = AF_INET ;
   len = sizeof( server ) ;
/* Appel du service echo */
   client(sock, &server, len ) ;
/* Fermeture connexion */
   close (sock) ;
}



/* Fonction emission - reception */
client( sock, pserver, len )
int sock ;                    /* Descripteur socket */
struct sockaddr_in *pserver ; /* adresse serveur */
```

```
int len ;                    /* longueur adresse */
{
   XDR xdr_handle1 ;         /* handle encodage */
   XDR xdr_handle2 ;         /* handle decodage */
   char *mem ;               /* buffer */
   char *pbuf ;              /* pointer */
   unsigned int size ;       /* taille multiple de 4 */
   unsigned int pos ;        /* position */
   int serverlen ;           /* longueur adresse */
/* Initialisation variable contenant longueur structure
   adresse du serveur */
   serverlen = len ;
/* buffer de taille TAILLEMAXI + 4 pour le codage XDR avec RNDUP */
   size = RNDUP( TAILLEMAXI + 4 ) ;
   mem = malloc( size ) ;
/* adresse d'un pointeur a xdr_st */
   pbuf = buf ;
/* Allocation flots XDR memoire pour encodage / decodage */
   xdrmem_create( &xdr_handle1, mem, size, XDR_ENCODE ) ;
   xdrmem_create( &xdr_handle2, mem, size, XDR_DECODE ) ;
/* Envoi de la taille du buffer traite, on fait un transcodage pour
   connaitre la taille de ce qui sera transmis */
   xdr_st( &xrd_handle1, &pbuf ) ;
   lbuf = xdr_getpos( &xdr_handle1 ) ;
/* On se repositionne en debut de buffer */
   xdr_setpos( &xdr_handle1, 0 ) ;
/* Encodage */
   xdr_int( &xdr_handle1, &lbuf ) ;
/* Longueur de chaine encodee */
   pos = xdr_getpos( &xdr_handle1 ) ;
/* transmission au serveur */
   retour = sendto( sock, mem, pos, 0, pserver, len ) ;
/* Boucle envoi et reception de buffers */
   for (i=0, i < nbuf; i++ )
   {
/*    Repositionnement en debut de buffer */
      xdr_setpos( &xdr_handle1, 0 ) ;
/*    Encodage */
      xdr_st ( &wdr_handle1, &pbuf ) ;
/*    Transmission */
      retour = sendto( sock, mem, lbuf, 0, pserver, len ) ;
/*    Reception sur l'adresse du serveur (connue) */
      retour = recvfrom( sock, mem, lbuf, 0, pserver, &serverlen ) ;
/*    Repositionnement debut du buffer */
      xdr_setpos( &xdr_handle2, 0 ) ;
/*    Decodage */
      xdr_st( &xdr_handle2, &pbuf ) ;
```



```
   }
/* Liberation memoire */
   free( mem ) ;
}

/* Fichier serveur.c */
#include "soct.h"
serveuripc()
{
   int sock ;         /* descripteur socket */
   struct sockaddr_in server ; /* adresse serveur */
   struct sockaddr_in sclient ; /* adresse client */
   int len ;                    /* longueur adresse */
/* creation socket */
   sock = socket( AF_INET, SOCK_DGRAM, 0 ) ;
/* assignation adresse a la socket */
   bzero( &server, sizeof(server) ) ;
   server.sin_family = AF_INET ;
   server.sin_addr.s_addr = INADDR_ANY ;
   server.sin_port = htons(PORTS) ;
   len = sizeof( server ) ;
   bind( sock, (struct sockaddr *) &server, len ) ;
/* Appel de la boucle ecriture */
   for (;;)
   {
      serveur( sock, &sclient, sizeof(sclient) ) ;
   }
}
/* Fonction reception - emission */
serveur ( sock, psclient, len )
int sock ;                    /* descripteur socket */
struct sockaddr_in *psclient ; /* adresse client */
int len ,                     /* longueur adresse */
{
/* Traitement symetrique par rapport au client */

}
```

Echo avec flot d'enregistrement et sockets TCP [14]

```
/* Fichier gen.x : Description RPCGEN des donnees echangees */
typedef string st<16384>  /* Longueur de chaine maxi */
/* Fichier soct.h */
#include <commun.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define PORT 6368   /* Numero de port TCP */
```

```
/* Fichier d'inclusion pour les procedures XDR */
#include <rpc/rpc.h>
/* Fichier d'inclusion engendre par RPCGEN */
#include "gen.h"
readp () ;      /* Procedure de lecture sur socket */
writep() ;      /* Procedure d'ecriture sur socket */
/* Fichier gen.h engendre par RPCGEN */
#include <rpc/types.h>
typedef char *st ;
bool_t xdr_st() ;

/* Fichier client.c */
#include "soct.h"
clientipc()
{
   int sock ;                  /* Descripteur socket */
   struct sockaddr_in server ; /* Adresse serveur */
/* Creation socket */
   sock = socket( AF_INET, SOCK_STREAM, 0 ) ;
/* Connexion au serveur */
   bzero( &server, sizeof(server) ) ;
   server.sin_familly = AF_INET ;
   bcopy( (char *) hp->h_addr, (char *) &server.sin_addr, hp->h_lenght );
   server.sin_port = htons(PORT) ;
   connect( sock, (struct sockaddr *) &server, sizeof(server)) ;

/* Appel du service */
   client(sock) ;
/* Fermeture connexion */
   close (sock) ;
}

/* Fonction emission - reception */
client( sock )
int sock ;                    /* Descripteur socket */
{
   char *pbuf ;               /* pointer */
   XDR xdrs ;                 /* handle XDR */
/* Pointer sur buffer */
   pbuf = buf ;
/* Mode ecriture */
   xdr.x_op = XDR_ENCODE ;
*/ Creation handle */
   xdrrec_create( &xdrs, 0, 0, &sock, readp, writep ) ;
/* Envoi de la taille du buffer traite */
   xdr_int( &xrd, &lbuf ) ;
/* Flush du buffer d'ecriture */
```



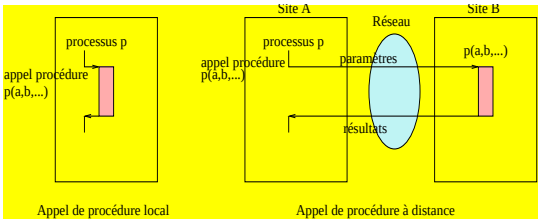
```
   xdrrec_endofrecord( &xdrs, TRUE ) ;
/* Boucle envoi et reception de buffers */
   for (i=0, i < nbuf; i++ )
   {
/*    Ecriture et encodage */
      xdr.x_op = XDR_ENCODE ;
      xdr_st ( &xdrs, &pbuf ) ;
/*    Flush du buffer */
      xdrrec_endofrecord( &xdrs, TRUE ) ;
/*    Lecture et decodage */
      xdrs.x_op = XDR_DECODE ;
/*    Positionnement sur l'enregistrement */
      xdrrec_skiprecord( &xdrs ) ;
      xdr_st( &xdrs, &pbuf );
    }
}

/* Fichier serveur.c */
#include "soct.h"
/* Variables globales positionnees dans les procedures readp() et writep() */
extern int nbcarlu ;   /* Nombre d'octets lus sur socket */
extern int nbcar ecrit ; /* Nombre d'octets ecrits sur socket */

serveuripc()
{
   int sock ;         /* descripteur socket */
   int nsock ;        /* descripteur socket */
   int retour ;       /* variable retour */
   struct sockaddr_in server ; /* adresse serveur */
/* creation socket */
   sock = socket( AF_INET, SOCK_STREAM, 0 ) ;
/* assignation adresse a la socket */
   bzero( &server, sizeof(server) ) ;
   server.sin_family = AF_INET ;
   server.sin_addr.s_addr = INADDR_ANY ;
   server.sin_port = htons(PORT) ;
   bind( sock, (struct sockaddr *) &server, sizeof(server) ) ;
/* mise a l'ecoute des connexions entrantes */
   listen( sock, 5 ) ;
/* boucle sur les demandes de connexion */
   for (;;;)
   {
      nsock = accept( sock, (struct sockaddr *) 0, (int *) 0 );
/* appel de la boucle lecture-ecriture */
      serveur( nsock ) ;
/* fermeture connexion courante */
      close ( nsock ) ;
```

```
   }
}

/* Fonction emission - reception */
serveur ( nsock )
int nsock ;   /* descripteur socket */
{
   Code symetrique par rapport au client */
   ...........
/* On sort de la boucle ,reception-emission lorsque
nbcarlu = 0 (client a fait un close) */
   if ( nbcarlu == 0 ) return;

/* Fichier soc.c : contient readp() et writep() */
#include <stdio.h>
/* On memorise nb octets lus et ecrits  afin de les exploiter evt */
int nbcarlu ;   /* Nombre d'octets lus */
int nbcarecrits ; /* Nombre d'octets ecrits */

/* Procedure de lecture sur socket */
readp( sock, buf, n )
int *sock ; /* descripteur socket */
char *buf ; /* buffer */
unsigned int n ; /* Nombre d'octets a lire */
{
   int nlu ;
   nlu = read( *sock, buf, n ) .
   nbcarlu = nlu ;
/* Positionnement erreur si si pas de car. lu */
   if ( nlu == 0 ) nlu = -1 ;
   return nlu ;
}
/* Procedure d'ecriture sur socket */
writep( sock, buf, n )
int *sock ; /* Descripteur socket */
char *buf ; /* buffer */
unsigned int n ; /* Nombre d'octets a ecrire */
{
   int necr ;
   necr = write( *sock, buf, n ) ;
   nbcarecrit =  necr ;
/* Positionnement erreur sir si pas car ecrit */
   if ( necr == 0 ) necr = -1 ;
   return necr ;
}
```


6 Appels de procédure à distance (RPC)

6.1 Introduction

- Introduit par SUN pour implanter NFS
- Outil de haut niveau pour la réalisation du schéma client-serveur
- Principe



  Effet vu du processus doit être identique (pb erreurs, pannes, pertes de messages, ...)

- Avantages :
  - Forme et effet identique à un appel local
    - Pas de modif des applications en passant à l'appel distant
    - Mise au point en local
    - Simplicité d'utilisation
  - Niveau d'abstraction
    - Indépendance par rapport aux protocles de communication (pas besoin d'apprendre un protocole de bas niveau)
    - Réutilisation possible du code y compris dans un environnement hétérogène
- Difficultés :
  - Situations complexes en cas de panne :
    - Processus client et serveur peuvent tomber en panne indépendamment
    - Incertitude introduite par le réseau (pertes, retard,...)
  - Restrictions sur les paramètres : pb de passage de structures complexes
- Sous UNIX protocole sous-jacent UDP pour la grande majorité des cas (rarement TCP).

Partage du serveur entre plusieurs clients

- Un serveur peut servir plusieurs clients

- Pour le serveur
  - Gestion des requêtes : file de requêtes, priorité
  - Exécution du service : séquentiel, concurrent
  - Mémorisation ou non de l'état du client

Gestion de processus du côté serveur

- Sur le serveur, la procédure distante est exécutée par un processus
- Plusieurs possibilités :
  - Dans tous les cas un processus de veille attend derrière une porte spécifiée (numéro de porte fonction du service)
  - Appel → message envoyé au veilleur avec nom de procédure et paramètres
    * le veilleur exécute lui-même la procédure et renvoie les résulats au client
    * ou le veilleur crée un processus (lourd ou léger) pour exécuter la procédure, le processus renvoie les résultats
    * Possibilité d'avoir un "pool" d'exécutants qui vont chercher le travail à effectuer donné par le veilleur

Réalisation de l'appel

- Problèmes d'identification et de désignation
  - Le client doit pouvoir désigner le serveur (envoi requête)
  - Le serveur doit pouvoir désigner le client (envoi résultats)
- Solutions possibles
  - Désignation du serveur par le client
    * Le client connait le site et le numéro de porte du serveur (convention, ...) → problème résolu
    * Le client ne connait qu'un nom symbolique de service :
      · Utilisation d'un service de désignation qui fournit (site, numéro de porte)
      · Connaissance du site et numéro de porte du service de désignation
  - Désignation du client par le serveur : dans sa requête le client indique (site, numéro de porte) vers où renvoyer le réponse
- En pratique : noms symboliques et opérations de désignation cachées aux utilisateurs

6.2 Principes du protocole ([19])

Le protocole doit permettre :

- Identification des procédures
- Authentification de la demande

1. Authentification des procédures :
   - Procédures regroupées en un programme réalisant un service (e.g. NFS)
   - Programme identifié par un entier ainsi que chaque procédure (numéro de NFS 100003 et lecture procédure 6). Chaque programme possède de plus un numéro de version.
   - Appel à fonction distante :
     → requête à un démon de la machine distante en lui transmettant numéro de programme, de version, et de procédure
     → démon lancera dialogue avec un processus de service exécutant la procédure demandée.
   - Echange d'informations avec XDR
   - Tout service contient la procédure 0 qui ne fait rien rien mais permet de tester sa disponibilité.
2. Numéros de programme :
   - Entiers longs allant de 0x00000000 à 0xffffffff.

Intervalle hexadécimal	Usage
0x00000000 - 0x1fffffff	Réservé
0x20000000 - 0x3fffffff	Non réservé
0x40000000 - 0x5fffffff	Réservé
0x60000000 - 0xffffffff	Réservé

Table 2: Numéros de programme RCP.

3. Authentification
   - Possibilité pour un client de s'identifier auprès du serveur → sécurité des accès
   - Messages échangés au cours des appels de procédures distantes incluent cette identification
   - Protocole idépendant du système sous-jacent → plusieurs styles d'authentification possibles (abscence, authentification UNIX, ..., définition nouveaux styles)

6.3 Implantation sous UNIX

- Met en jeu un certain nombre de fichier et de processus particuliers.
- Développement de service possible à plusieurs niveaux

1. **Services, ports Internet et processus**
   - Service → numéro de programme
   - Appel à procédure d'un service → réalisation de l'appel par un processus chargé de l'exécution du service.
   - Communication entre processus de service avec module appelant via socket Internet sur UDP ou TCP (pluis rarement).
   - Socket associée au numéro de port du protocole correspondant.
   - Deux possibilités pour le processus de service :
     - Processus créé une fois pour toutes et en écoute sur le port associé
     - Port associé fait partie d'un ensemble de ports sur lequel un processus particulier est en écoute (e.g. **inetd**). Ce processus crée le processus de service si nécessaire
2. **Processus portmap**
   - Processus correspondant à un service RPC particulier : associer un numéro de port à un numéro de service PRC donné
   - Doit être actif pour accéder au mécanisme RPC sur une machine donnée
   - Tout nouveau service doit être signalé à **portmap** avec des fonctions de la bilbiothèque standard (mécanisme d'enregistrement de service)
3. **Fichier /etc/rpc**
   - Contient la liste des services RPC
   - Ligne = informations relatives à un service
     - Nom officiel
     - Numéro
     - Liste d'alias
   - **rpcent** prédfinie dans <**netdb.h**>

```
struct rpcent {
    char *r_name ;          /* nom de programme RPC */
    char **r_aliases ;      /* liste d'alias */
    int r_munber ;          /* numero de programme RPC */
} ;
```

   utilisée par les différentes fonctions permettant de consulter **/etc/rpc** : **getrpcbyname** et **getrpcbynumber**.

4. **Commande rpcinfo**
   - Obtention d'informations sur les divers services disponibles sur une machine
   - Exemple

```
% rpc -p julia
   program no_version protocole  no_port
    100000    4   tcp    111  rpcbind
    ......
    100000    4   udp    111  rpcbind
    ......
    100005    1   udp  32998  mountd
    ......
    100003    3   tcp   2049  nfs
    100227    2   tcp   2049  nfs_acl
    ......
 805306368    1   udp  33001
 805306368    1   tcp  32800
    100249    1   udp  33002
    100249    1   tcp  32801
```

   - Tester disponibilité d'un service

```
% rpcinfo -u julia nfs
Le programme 100003 de version 2 est pret et en attente.
Le programme 100003 de version 3 est pret et en attente.
```

5. **Différents niveaux d'utilisation**
   - Couche haute :
     - Cache un maximum de détails à l'utilisateur
     - Uniquement appel de fonction dans une bibliothèque
     - Pas possible de développer de nouveaux services
   - Couche intermédiaire
     - La plus intéressante pour le développeur
     - Connaissance minimale de XDR et RPC et suffit pour la majorité des applications
     - Pas de manipulation explicite des sockets
   - Couche basse :
     - Nécessite une bonne connaissance des sockets
     - Nécessaire si les options choisies dans couhe intermédiaire (protocole UDP par exemple) sont inadaptées

6.4 Couche haute

- Fonctions standards disponibles dans **librpcsva.c**
- Simple édition de liens avec cette bibliothèque
- Exemples de fonctions :
  - **getrpcport** : fournit avec nom de machine + numéro de programme, de version et de protocole, le numéro de port si il est connu (0 sinon)
  - **rusers, rnusers** :
    - **rnusers(machine)** → nombre d'utilisateurs connectés
    - **rusers(machine, p)** → initialise la zone **p** avec les informations disponibles sur les utilisateurs.
  - **rwall(machine, message)** : envoi du message donné à tous les utilisateurs de la machine spécifiée.

6.5 Couche intermédiaire

- Création relativement simple de nouveaux services RPC
- S'appuie sur le protocole UDP → taille des messages limitée (et donc taille des paramètres et des résultats)
- Si insuffisant → couche basse

Suite des opérations à réaliser pour définir un nouveau service :

- Ecrire les différentes fonctions sur le serveur
- Après choix des numéros de programme et de version, demander l'enregistrement par démon **portmap** avec **registerrpc**.
- Appel à distance avec **callrpc**

Exemple de service : 3 procédures permettant de calculer

- $x + y$ pour la procédure 1
- $(x \times y, \frac{x}{y})$ pour la procédure 2
- $(\sqrt{x}, \sqrt{y})$ pour la procédure 3.

Exemple du côté du serveur

1. **Ecriture des fonctions**
   - Toute fonction existante peut être intégrée avec quelques modifications (paramètres et résultats) à un service RPC.
   - Un seul paramètre pointant sur une zone mémoire contenant les divers paramètres (structure) : lié à la nécessité d'utiliser XDR pour coder ces paramètres.

   - Idem pour les résultats : pointeur de type *char sur une zone contenant le résultat de la fonction (adresse en zone statique)

```
/* exemple.h */
#include <rpc/types.h>
#include <rpc/xdr.h>
#define ARITH_PROG 0X33333333   /* Numero de programme */
#define ARITH_VERS1 1           /* Numero de la version 1 */
#define ADD_PROC 1              /* Numero de la procedure add */
#define MULT_PROC 2             /* Numero de la procedure mult */
#define SQRT_PROC 3             /* Numero de la procedure rac */
struct couple {
   float e1, e2 ;
              } ;
int xdr_couple () ;

/* xdr_couple.c */
#include "exemple.h"
xdr_couple( xdrp, p )
XDR *xdrp ;
struct couple *p ;
{
   return( xdr_float( xdrp, &p ->e1 ) && xdr_float( xdrp, &p->e2)) ;
}

/* Fonction 1 : add.c */
#include "exemple.h"
char *add( p )
struct couple *p ;
{
   static float couple res ;
   res.e1 = p->e1 + p->e2 ;
   return ((char *) &res );
}

/* Fonction 2 : mult.c */
#include "exemple.h"
char *mult( p ) ;
struct couple *p ;
{
   static struct couple res ;
   res.e1 = p->e1 * p->e2 ;
   res.e2 = p->e1 / p->e2 ;
   return ((char *) &res );
}

/* Fonction 3 : rac.c */
```

```
#include "exemple.h"
char *rac( p )
struct couple *p ;
{
   static struct couple res ;
   res.e1 = sqrt(p->e1) ;
   res.e2 = sqrt(p->e2) ;
   return ((char *) &res );
}
```

2. **Enregistrement du service**

   - Signaler l'existence du service au démon **portmap**
   - Chaque fonction est enregistrée individuellement avec

     ```
     int registerrpc( prog, version, proc, f, xdr_arg, xdr_res )

        unsigned long prog, version, proc ;
        char *(*f) ( ) ;
        bool_t (*xdr_agr)(), (*xdr_res)() ;
     ```

     la fonction***f** est enregistrée sous le numéro **proc** dans la version **vers** du programme de numéro **prog**. **xdr_arg, xdr_res** définissent les traitements XDR à appliquer aux paramètres (décodage) et aux résultat (encodage). En retour 0 ou -1 en cas d'erreur.
   - Enregiostrement suppose ensuite choix d'un numéro de service, d'un numéro de version et de numéros de procédures. Définition aussi contenue dans exemple.h.
   - Enregistrement des 3 procédures dans un programme principal par appels à **registerrpc** :

     ```
     ...
     rep = registerrpc( ARITH_PROG, ARITH_VERS1, ADD_PROG,
                        add, xdr_couple, xdr_float ) ;
     if ( rep == -1 ) {
        fprintf ( sterr, "erreur registerrpc (add)\n") ;
        exit(2) ; }
     rep = registerrpc( ARITH_PROG, ARITH_VERS1, MULT_PROG,
                        mult, xdr_couple, xdr_couple ) ;
     if ( rep == -1 ) {
        fprintf ( sterr, "erreur registerrpc (mult)\n") ;
        exit(2) ; }
     rep = registerrpc( ARITH_PROG, ARITH_VERS1, SQRT_PROG,
                        rac, xdr_couple, xdr_couple ) ;
     if ( rep == -1 ) {
        fprintf ( sterr, "erreur registerrpc (rac)\n") ;
        exit(2) ; }
     ```


     ...

   - NB : enregistrer le service ne veut pas dire qu'il est disponible. Il faut aussi avoir l'existence d'un démon pour réaliser ce service (endormi et réveillé à la demande ou créé lorsque le service est appelé).

3. **Fonction svc_run**

   - Solution la plus simple pour rendre un service disponible : le processus qui demande l'enregistrement (avec **resgisterrpc** est aussi le processus démon du service.
   - Après enregistrement, il se met en attente de demandes avec **svc_run**.
   - Réalise une attente par appel à **select** : liste des descripteurs susceptibles de réveiller le processus en vue d'une lecture contient le descripteur de la socket associée au service RPC (accessible via la variable externe **svc_fds**
   - Quand aucun message n'est disponible sur cette socket, le processus se met en attente
   - **svc_run** n'a pas de retour sauf en cas d'erreur

4. Effacement du service :

   - Disparition service devrait être signalée au démon **portmap**

     ```
     pmap_unset( prog, vers )

        unsigned long prog ; /* Numero de programme */
        unsigned long vers ; /* Numero de version */
     ```

   - Le programme suivant efface le service ariht :

     ```
     main( n , v )
     int n ;
     char *v[] ;
     {
        unsigned long prog, vers ;

        sscanf( v[1], "%1", &prog ) ;
        sscanf( v[2], "%1", &vers ) ;
        pmap_unset( prog, vers ) ;
     }
     ```

     avec les paramètres 858993459 (0x33333333) et 1, le service arith est effacé des tables de portmap.

Code complet du serveur

```
/* Enregistrement des procedures du programme arithmetique et
mise en oeuvre du demon du service */
#include<stdio.h>
```

```
#include "exemple.h"
char *add() ;
char *mult() ;
char *rac() ;
main()
{ int rep ;
   rep = registerrpc( ARITH_PROG, ARITH_VERS1, ADD_PROG,
                      add, xdr_couple, xdr_float ) ;
   if ( rep == -1 ) {
      fprintf ( sterr, "erreur registerrpc (add)\n") ;
      exit(2) ; }
   rep = registerrpc( ARITH_PROG, ARITH_VERS1, MULT_PROG,
                      mult, xdr_couple, xdr_couple ) ;
   if ( rep == -1 ) {
      fprintf ( sterr, "erreur registerrpc (mult)\n") ;
      exit(2) ; }
   rep = registerrpc( ARITH_PROG, ARITH_VERS1, SQRT_PROG,
                      rac, xdr_couple, xdr_couple ) ;
   if ( rep == -1 ) {
      fprintf ( sterr, "erreur registerrpc (rac)\n") ;
      exit(2) ; }
   svc_run() ;
   fprintf( stderr, "erreur sur svc_run\n") ;
   exit(3) ;
}
```

Exemple du côté des clients

1. **Fonction callrcp**

```
callrpc( machine, prog, vers, proc, xdr_arg, arg, xdr_res, res )

   char *machine ;
   unsigned long prog, vers, proc ;
   char *arg, *res ;
   bool_t (*xdr_arg)(), (*xdr_res)() ;
```

- Appel sur **machine**, de la fonction **proc** de la version **vers** du programme **prog**.
- **arg** → paramètres de la procédure et ***xdr_arg** spécifie le traitement XDR correspondant à leur type.
- **res** → résulats de la procédure et ***xdr_res** spécifie le traitement XDR correspondant à leur type.
- Appel de la fonction bloquant. Retour de 0 en cas de réussite et autre valeur en cas d'échec.

```
/* Exemple d'appel au service arith sur une machine distante */
#include<stio.h>
#include "exemple.h"
```



```
main( n, v )
char *v[] ;
int n ;
{
   float x ;
   struct couple don, res ;
   int op, m ;

   don.e1 = 13.4 ;
   don.e2 = 17.1 ;
   m = callrpc( v[1], ARITH_PROC, ARITH_VERS1,
                ADD_PROC, xdr_couple, &don, xdr_float, &x ) ;
   if ( m == 0 )
      printf( "%f + %f = %f\n",don.e1,don.e2,x) ;
   else
      fprintf(stderr, "erreur : %d\n",m ) ;
}
```

2. **Erreurs**

- Valeur de retour de **call rpc** fournit en cas d'échec sa cause. <**rpc/clnt.h**> fournit la liste des erreurs possibles.
- Fonction **clnt_perrno** appelé avec un numéro d'erreur en paramètre → affiche le message correspondant.
- RPC s'appuyant à ce niveau sur UDP → risque de blocage de processus. Détection de blocage par répétition à intervalles de temps réguliers pendant un certain temps.
  ces répétitions peuvent avoir des effets de bord !!

Couche basse

- Fonctionnalités du même type que la couche intermédiaire
- Possibilités d'utiliser TCP au lieu de UDP, enregistrement complet d'un service en une seule fois (au lieu de procédure par procédure), ...
- Mise en œuvre plus lourde

Concepts avancés

- Démons RPC définis précédemment sont lancés une fois pour toutes (existent même lorsque le service n'est pas appelé).
- Possibilité de charger le processus **inetd** (jouant un rôle de super-serveur) pour recréer le démon associé à un service lorsque nécessaire
- Nécessite d'être **root**

1. **Fichier inetd.conf**

   - Contient la liste des services supervisés par **inetd**.

     ```
     # These are standard services.
     ....
     ftp stream tcp nowait root /usr/sbin/tcpd in.ftpd -l -a
     telnet stream  tcp  nowait  root    /usr/sbin/tcpd in.telnetd
     ....
     ```

   - Un service doit être aussi répertorié dans **rpc**
   - Pour traiter l'exemple on doit donc ajouter
     - A **/etc/inetd.conf**

       ```
       arithd sunrpc_udp wait root /usrf/etc/rpc.arithd arithd 858993459 1
       ```

     - A **/etc/rpc**

       ```
       arithd 858993459 arithd
       ```

2. Modifications sur le serveur RPC :

   - Serveur créé par **inetd** après que celui ci ait accepté une connexion sur le port du service (*select* puis *accept*).
   - **inetd** lance le processus de service (*fork* puis *exec*) et lui transmet la socket de service via le descripteur 0.
   - Enregistrement du service réalisé une fois pour toutes. A son lancement **inetd** informe **portmap** des services qu'il prend en compte → **portmap** créé avant **inetd**.
   - Le code des fonctions de service doit être modifié pour rendre le contrôle au processus **inetd** au lieu de main → appel à *exit* au lieu de *return*

3. Authentification des requêtes

   - Importante pour les serveurs dont le propriétaire est **root**
   - Authentification → doit permettre l'identification du client dans divers systèmes
   - Divers types d'authentification :
     - Style d'authenfication :
       Dans le domaine UNIX absence d'authentification (**AUTH_NULL**) ou authenfication UNIX (**AUTH_UNIX**) avec personne + groupe.
       Par défaut authentification nulle (constantes prédéfinies dans **/etc/auth.h**.
       Authentification = chaîne de caractères dont l'interprétation dépend du style correspondant.
     - Authentification = structure :

       ```
       struct authunix_parms {
         u_long     aup_time ;    /* date de creation de la structure */
         char       *aup_machine ;/* nom de la machine cliente */
       ```



       ```
         int        aup_uid ;     /* proprietaire effectif du client */
         int        aup_gid ;     /* groupe proprietaite effectif du client */
         u_int      aup_len ;     /* nombre d'elements du champ suivant */
         int        *aup_gids ;   /* tableau de groupes d'appartenance */
       ```

       Structure manipulée par le client et le serveur (mais opaque pour le client).

4. Traitement des erreurs

   - Les erreurs sont récupérables par le client avec **clnt_perrno** et **clnt_perror**
   - Que se passe-t-il lorsque le serveur détecte une anomalie dans le déroulement d'un appel à une fonction ?
   - Un certain nombre de fonctions permmetent au serveur de renvoyer aux client des erreurs spécifiques.
   - Exemples :
     - **sverr_noproc** : numéro de procédure incorrect
     - **svcerr_auth** : erreeur dans l'authentification
     - ...

5. Possibilité de personnaliser **svc_run** par exemple pour que le serveur réalise des lectures bloquantes sur d'autres descripteurs que la socket de service avec **svc_getreq** qui permet de traiter toutes les lectures spécifiées par un masque.

7 Network File System (NFS)

7.1 Introduction

- Permet de connecter ensemble de ressources : disques, fichiers, processeurs, ...

  machine virtuelle

- Gestion de cette machine virtuelle : système distribué assurant les fonctions de base d'un système d'exploitation de façon transparente
- Chaque machine a un système gérant ses ressources locales.
- Accès à un fichier distant → adresse réseau de la machine possédant le fichier + demande de transfert du fichier sur le système local
- NFS proposé par SUN : partage de fichiers en environnement hétérogène
- Objectif : maximum de transparence (on peut manipuler fichiers distants et locaux de la même façon).
- Présentation inspirée de [19]

7.2 Montage et démontage distants

- Modèle client / serveur
- Extension du montage permettant d'associer une référence de répertoire local à une référence distante (nom de machine + nom de répertoire sur cette machine)
- Exemple :

  ```
  wanda # mount julia:/export/libs /libs
  ```

  montage du répertoire /export/libs sur la machine julia en /libs sur la machine locale wanda (root).

  Il suffit ensuite de référencer /libs/file.

- **df, mount** sans arguments permettent de visualiser les montages distants.

  ```
  % df
  /proc              (/proc             ):       0 blocs      960 fichiers
  /                  (/dev/dsk/c0t3d0s0 ):  230246 blocs    79009 fichiers
  /usr               (/dev/dsk/c0t3d0s6 ):  443628 blocs   308109 fichiers
  /dev/fd            (fd                ):       0 blocs        0 fichiers
  /export/exterieurs (/dev/dsk/c0t2d0s4 ): 2571336 blocs   402517 fichiers
  /export/EuroPar    (/dev/dsk/c0t2d0s5 ):  732336 blocs   477982 fichiers
  /export/home       (/dev/dsk/c0t2d0s7 ): 4602588 blocs   489098 fichiers
  /tmp               (swap              ):  272928 blocs    10289 fichiers
  /var/mail          (mailhost:/var/mail): 2785118 blocs   484012 fichiers
  /home/dayde        (/export/home/dayde): 4602588 blocs   489098 fichiers
  ```

- Démontage de disque par **umount**.

  ```
  wanda # umount /libs
  ```

- Possibilité de monter un disque distant suppose :
  - Point de montage distant = racine d'un disque logique
  - Montage autorisé par la machine distante (/etc/export contient la lise des répertoires qui peuvent être montés et depuis quelle machine – absence de nom → toute machine)

Configuration initiale du système

- Au lancement du système /etc/fstab utilisé pour monter un certain nombre de disques :

  ```
  /dev/hda2           /           ext2    defaults        1 1
  /dev/hda1           /dos/c      vfat    defaults        0 0
  /dev/hda7           /usr        ext2    defaults        1 2
  /dev/hda5           swap        swap    defaults        0 0
  /dev/fd0            /mnt/floppy ext2    noauto          0 0
  /dev/cdrom          /mnt/cdrom  iso9660 noauto,ro       0 0
  julia:/export/home  /home       nfs     rsize=8192,wsize=8192,timeo=14,intr
  ```


- En de non-réponse les requête sont réitérée un certain nombre de fois (avec un intervalle de temps)
  - Montage *soft* → renvoie une erreur au programme appelant (en général nouvel essai jusqu'à une certaine durée ou un nombre maximal).
  - Montage *hard* → demande réitérée indéfiniment.
- Performances très dépendantes de la paramétrisation du système (en particulier détection de défaillance d'un serveur).

7.3 Implantation de NFS

1. Système de fichiers virtuels (VFS)
   - Notion de **vnode** : généralise celle de **inode**
   - Associe à un fichier du réseau un numéro non ambigü
   - A partir du vnode VFS permet de retrouver i-node correspondant
   - Du point de vue application interface NFS identique à l'interface usuelle : utlisation des primitives **open, read, write, close, . . .**
2. Programmes RPC : **mount, nfs**
   - Appel d'une application à une primitive système en vue d'une opération sur un fichier distant → appel au un programme RPC (programme **nfs** de numéro 100003).
   - XDR utilisé pour les échanges de données relatives au protocole (fichier = structure opaque XDR transmise au système distant) mais pas pour les données utilisateur.
   - Programme RPC **mount** (numéro 100005) restitue la structure (file handle) d'un fichier donné qui sera utilisée au cours des requêtes NFS.
   - **mount** et **nfs** mis en œuvre par les démons **nsfd** et **mountd**.
   - NFS au dessus de UDP pour les versions courantes.
   - Accés d'un fichier distant par application
     → appel au service **mount** pour obtenir structure opaque désignant le fichier
     → appel au service **nfs**
3. Caractéristiques principales du protocole :
   - Serveurs de fichiers **sans état** : un serveur n'a aucune mémoire des clientsutilisant ses fichiers → pas de trace des requêtes
   - Opération **open** sur un fichier distant → pas d'ouverture par le serveur de ce fichier sur le système auquel il appartient
   - Avantage : moins de problèmes suite aux défaillances d'un client ou d'un serveur

- Inconvénients : comportement différent entre accés à un fichier local et un fichier distant
  - droits d'accès à un fichier ne sont testées qu'à l'ouverture d'un fichier local par un processus (et plus ensuite). Toutes les opérations compatibles avec ces droits sont ensuite autorisées.
  - Avec un fichier distant, on peut avoir une écriture qui échoue alors que l'open a réussi (droits d'accès modifiés par le proriétaire du fichier qui n'est pas le propriétaire du processus accédant le fichier).
  - Même type de problème avec suppression de fichiers (suppression d'un fichier référencé par plusieurs processus).

4. Problème des droits d'accès
   - NFS utilise le protocole RPC et authentification du type UNIX (AUTH_UNIX).
   - A chaque appel → **nsfd** reçoit identité de l'utilisateur et du groupe propriétaires du processus client pour tester les droits.
   - Problème lorsque les systèmes ne sont pas administrés par les mêmes personnes et lorsque l'utilsateur n'a pas le même numéro d'identification sur les machines
   - Problème avec le **root** de numéro 0. Via NFS, il perd ses droits : identification transmise -2 ("nobody").
5. Les pages jaunes ("yellow pages") NIS (Network Information Service)
   - Base de données distribuées pour l'administration du réseau
   - Solution pour la mise à jour des divers exemplaires d'un même fichier sur les systèmes d'un réseau (exemples : /etc/passwd, /etc/hosts)
   - Numéros de compte utilisateurs (et mots de passe) identiques sur toutes les machines d'un réseau local administré via les yellow pages
   - Idée : définition d'un domaine (ensemble d'associations clé + valeur), correspondant au contenus des différents fichiers à partager
   - Un site est le **maître** à partir de qui on réalise la mise à jour des copies sur les machines esclaves. Mise à jour des fichiers faite par l'administrateur excepté le changement de mot de passe avec **yppasswd**.

8 Processus communicants par messages

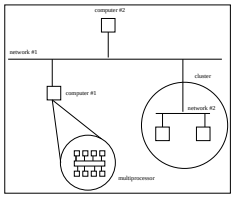
8.1 Contexte informatique, objectifs et besoins

Largement extrait de [1].

Contexte informatique

Multiprocesseur à mémoire distribuée ou réseau de stations de travail

Objectifs et besoins



Exemple de réseau de calculateurs.

- **But** : **répartir/gérer des calculs sur la machine cible**
- **Outils nécessaires** : (*minimum*)
  - *Sécurité et droits d'accés (machines et données)*
  - *Création de processus distants*
  - *Communication entre processus*
  - *Synchronisation entre processus*
- Gestion de la cohérence des donnés et des traitements
- Séquenceur des tâches réparties
- Gestion dynamiques des processeurs et des processus (gestion des pannes, gestion de points de reprises)

8.2 Le modèle de programmation par transfert de messages

- Permet d'exprimer la *communication et la synchronisation*
- **C'est le modéle le plus répandu en calcul réparti** mais ce n'est pas le seul (voir par ex. LINDA)
- Il n'apporte pas de solution à tous les problèmes posés.
- **Caractéristiques :**
  - expression du parallélisme à la charge du programmeur
  - distribution des données à la charge du programmeur
  - échange de données explicite
  - prise en compte possible d'un réseau hétérogènes de calculateurs avec gestion des pannes.

Modèle pelure d'oignon pour l'échange de message

Chaque niveau peut-être construit au dessus du précédent

- **Niveau le plus bas : adressage au niveau canal**
  - procédures pour transférer des paquets sur des liens

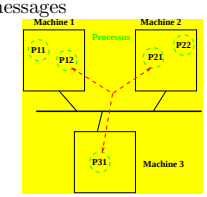
- **Adressage au niveau processus**
  - éventuellement plus d'un processus par processeur
  - échange de message en donnant l'adresse d'un processus
  - Exemples : Nx sur iPSC, Vertex sur nCUBE, Express, PARMACS, PVM, MPI, ...

- **Niveau plus élevé d'abstraction** : mémoire partagée virtuelle, LINDA, espace de tuples partagé ou réparti)

Hypothèse d'exécution

- Machine complètement connectée

- Routeur automatique de messages



les deux hypothèses ci-dessus ne sont pas toujours vraies (Transputers)

Librairies portables pour la programmation d'applications parallèles distribuées

- P4 de l'Argonne National Laboratory
  - offre à la fois les modèles mémoire partagée et transfert de message
  - communications entre processus
  - disponible et optimisé sur une large gamme de calculateurs (et réseaux de calculateurs)
- PICL de l'Oak Ridge National Laboratory portable sur une large gamme de multiprocesseurs à mémoire distribuée

- **PVM : Univ. Tennessee, Univ. Emory, Oak Ridge Nat. Lab., ...**
  - pour réseaux hétérogènes de calculateurs
  - aussi disponible sur multiprocesseurs
- **MPI : le standard pour le transfert de message**

8.3 Envoi et réception de messages

Environnement d'exécution des communications

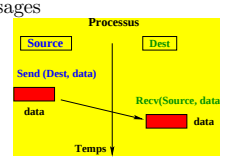
- Chaque processus est identifié par un numéro d'instance (**rang dans un groupe ou communicateur**)
- L'**enveloppe d'un message** doit permettre la caractérisation et le traitement du message. Elle contient:
  - le numéro de l'émetteur
  - le numéro du récepteur
  - le label du message
  - la taille du message
  - ...

Types de communication classiques

- **communications point à point (one-to-one)** : échange d'information entre 2 processus
- **communications collectives** (dans groupe / communicateur) :
  - **one-to-many** (broadcast, fan-out) : d'un processus vers un ensemble de processus
  - *many-to-one* (collect, fan-in) : un processus collecte des informations issues d'un ensemble de processus
  - **many-to-many** : échange global d'informations entre plusieurs processus

Communications point à point (quelques questions)

- Envoi et réception de messages



- Questions:
  - **Synchronisation entre envoi et réception ?**
  - **Quand peut-on réutiliser la donnée envoyée ?**
  - **Bufferisation des communications ?**

Mode de communication : **Synchrones/Asynchrones**

- **Envoi/réception synchrones**: Le premier arrivé attend l'autre (notion de rendez-vous).

- **Envoi/Réception asynchrones**: L'émetteur et le récepteur ne s'attendent pas.
- Un envoi asynchrone peut cependant être bloqué par la non consommation du message par le récepteur (sera détaillé par la suite)

Emetteur et récepteur n'ont pas à être tous les deux synchrones / asynchrones !!

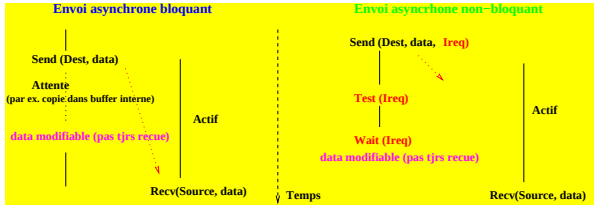
Envoi/Réception bloquants ou non bloquants

- **Envoi/Réception bloquants**:
  La ressource est disponible en retour de la procédure.
- **Réception non-bloquante simple**
  un paramètre de retour indique si l'information est disponible.

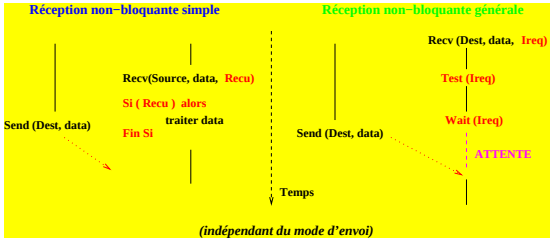
- **Envoi/Réception non-bloquants généraux** :
  -Retour de la procédure sans garantir que la donnée ait été envoyée/reçue.
  -L'utilisateur ne peut pas réutiliser l'espace mémoire associé (au risque de changer ce qui sera envoyé).
  -Il faut donc pouvoir **tester/attendre** la libération (si envoi) ou la réception effective de la donnée. **Send/Recv (Dest/Source, data, Ireq)** renvoie aussi **un numéro de requête**

  Test (Ireq) et **Wait (Ireq)**

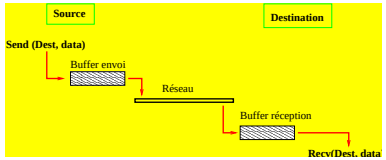
Envois asynchrone bloquant et non-bloquant



Réceptions bloquantes asynchrones



Où va l'information envoyée ?

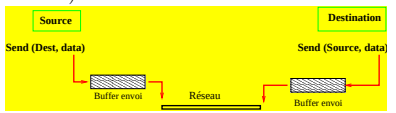


- Le(s) buffer(s) sont soit internes à la couche système soit gérés par l'utilisateur.

Proprietés de la communication bufférisée

- Buffer(s) implique(ent) **copies multiples**
  (coût mémoire et temps)
- Même dans un mode bloquant **l'envoyeur peut être libre immédiatement**
- Si Taille(buffer d'envoi) $\geq$ Taille (message) alors
  (envoi asynchrone bloquant $\equiv$ envoi non-bloquant)

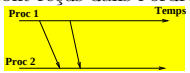
- Attention à gérer la saturation des buffers
  (**deadlock possible !!!**)



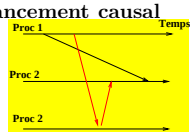
si l'envoi est asynchrone (bloquant) l'envoi peut être bloqué jusqu'au lancement de la réception correspondante.

Propriétés des communications

- **Diffusion des messages ordonnancée FIFO** :
  les messages issus de Proc1 sont reçus dans l'ordre par Proc2.



- Par contre **PAS d'ordonnancement causal**



Exemples (PVM et MPI) de communications point à point

- **Envoi/réception standard**

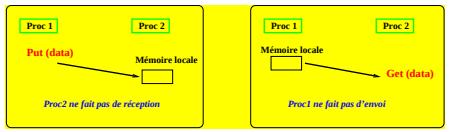
– *pvm_send/pvm_recv*: asynchrone bloquant
(*pvm_Nrecv* : réception non-bloquante simple)

– **mpi_send/mpi_recv**: bloquant (synchronisme dépend de l'implémentation)
(**mpi_Isend/mpi_Irecv** : communication non-bloquante générale)

- **Envoi synchrone**:
  - Bloquant : **mpi_Ssend**
  - Non-bloquant : **mpi_ISsend**
- **Envoi bufferisé**:
bloquant : **mpi_Bsend** et non-bloquant : **mpi_IBsend**

Communications non symétriques

- PUT(data) : écriture directe dans la mémoire d'un autre processus
- GET(data) : lecture dans la mémoire d'un autre processeur



Attention aux problèmes de cohérence de données !!

Examples d'opérations collectives

- Communications au sein d'un groupe de processus ou d'un communicateur
- *Les appels collectifs sont* **bloquants** *mais ne constituent pas un point de synchronisation fiable* (comparable à une barrière).
- Diffusion dans un groupe:
Broadcast (data, label, Groupe)

- Somme des données distribuées sur un groupe de procs:
Reduce (SUM, ValeurSum, ValeurLoc, Groupe, Dest)
ValeurSum n'est disponible que sur le processus **Dest**.
- Maximum de valeurs distibuées disponible sur chaque processus **AllReduce(MAX, ValeurMax, ValeurLoc, Groupe)**
ValeurMax est disponible sur tous les processus du **Groupe**.

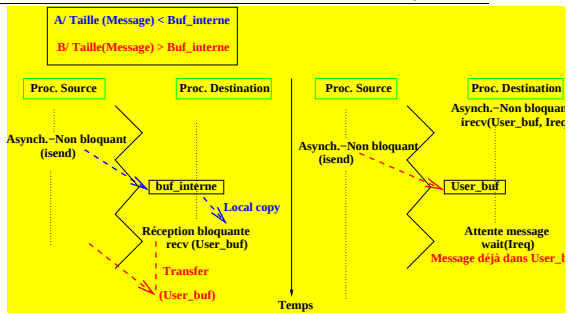
Commentaires sur les protocoles de communications

- un protocole différent est souvent utilisé pour les messages courts et les messages longs (ex. Cray T3E, SGI origin)

- **Protocole pour messages courts**
1/ écriture dans le buffer interne des données envoyées.
2/ le processus envoyeur continue son travail (si envoi asynchrone)

- **Protocole pour messages longs**
1/ envoi d'une **requète d'envoi** au destinataire
2/ **attente d'un message prêt à recevoir**
3/ envoi effectif des données rangées dans l'espace utilisateur du récepteur

Influence de la taille des buffers sur les Communications asynchrones



Remarques sur la taille des buffers systèmes

Changer la taille des buffers peut donc conduire à

- **Une modification de la performance**
(temps attente, nombre de copies internes ...)
- **Des résultats faux**
(causalité mal gérée)
- **De nouvelles situations d'interblocage**
(codes basés sur des communications standards (mpi_send et mpi_recv))

tous les cas d'erreur correspondent à des programmes/algo. erronés

Qu'est-ce qui influence les performances ?

- **La distribution des données**
- **L'équilibrage du travail sur les processus**
- *Recouvrement des communications par les calculs*

Optimisation des communications

- Choix du mode de communication (point à point ? symétrique ? synchrone ? bufferisé ? bloquant . . .)

- Optimisation algorithmiques: pipelining - blocage - envoi au plus tôt - prefetch ...
- Exploiter les protocoles d'implantation des communications (et taille des buffers internes)
- Exploiter l'architecture du réseau (topologie, connexions bidirectionnelles, fonctionnement de plusieurs canaux simultanés)

9 Librairies de transfert de messages

9.1 PVM

Distributed and heterogeneous computing using PVM

- Short overview of the PVM computing environment.
- Simple example.
- XPVM tracing tool used for illustration
- Complete description of PVM : [5, 11, 6].
- Many of the examples we use are coming from these references.

9.1.1 Overview of the PVM computing environment

PVM (Parallel Virtual Machine) :

- Public domain software available on *netlib*
- Developed by the Oak Ridge National Laboratory, the University of Tennessee, the University of Carnegie Mellon, the Pittsburgh Supercomputing Center and the Emory University of Atlanta.
- Allows to use a network of heterogeneous UNIX computers (either serial or parallel) as a unique computing resource referred to as a *virtual machine*.
- A variety of networks (Ethernet, FDDI, ...) may interconnect the nodes of the virtual machine
- Daemon on each node of the virtual machine coordinates work distributed on the virtual machine.
- Host file : contains the list of computers and allows to automatically activate the UNIX daemons and build the parallel virtual machine at start-up.
- Application viewed as a set of parallel processes being executed on the processors of the virtual machine
- Communication and synchronization using message passing

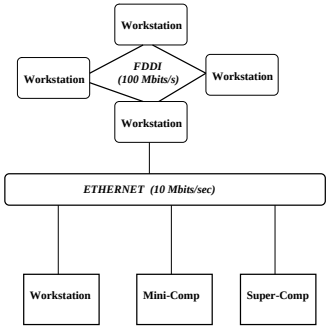


Figure 15: Example of virtual machine

- Processes can be organized into groups (a process can belong to several groups and groups can change at any time during computation).

From the user point of view, the PVM package is composed of two parts :

- **daemon process** *pvmd3*: resides on each computer of the parallel virtual machine. Can be started interactively or automatically.
- When user wants to run an application in the PVM environment:
  - *pvmd3* starts a daemon on each node of a virtual machine described in *host file*.
  - The application can then be started from any node.
  - *pvm* starts the PVM console used to interactively control and modify the virtual machine both in terms of host nodes and processes. *pvm* may be started and stopped multiple times on any of the hosts.
- **set of library procedures**: communication and synchronization procedures used from C or FORTRAN.
  - Several facilities for handling 'processes': to create and terminate processes,
  - to communicate between processes,
  - to synchronize processes,
  - to modify the parallel virtual machine,
  - and to manipulate process groups.

9.1.2 The PVM3 user library

We only describe the main procedures of the FORTRAN PVM user library.

Complete version of PVM3 user library → "PVM 3 user's guide and reference manual" [11] available on netlib.

Advanced features overviewed in [6].

Notations:

tid	integer	: identifier of the PVM process
ntask	integer	: number of processes
tids()	integer array	: array of PVM process identifiers
bufid	integer	: identifier of a buffer
msgtag	integer	: message label
encoding	integer	: message coding
bytes	integer	: length of a message in bytes
info	integer	: erro message
task	character	: name of an executable file
group	character	: group identifier
size	integer	: size of the group
xp	'what'	: data array
stride	integer	: stride between two elements
nitem	integer	: number of elements

Control and Activation of processes

Remark:
To use the predefined options and the error message coding, the file fpvm3.h must be included in the FORTRAN code (include '/usr/local/pvm3/include/fpvm3.h').

- **Procedure for enrolling a process into PVM**
  call **pvmfmytid**(tid)

  At its first call, the *pvmfmytid()* procedure creates a PVM process. *pvmfmytid()* returns the process identifier *tid* and may be called several times. If the host node does not belong to the parallel virtual machine then an error message is returned.

- **Leave PVM**
  call **pvmfexit**(info)

  pvmfexit indicates to the local daemon (*pvmd3*) that the process leaves the PVM environment. The process is not killed but it cannot anymore communicate (via PVM communication procedures) with the other PVM processes.

- **Kill another PVM process**: call **pvmfkill**(tid, info)

  pvmfkill kills the PVM process identified by *tid*.

- **Starting other processes on the virtual machine**
  call **pvmfspawn**(task, flag, where, ntask, tids, numt)

  Starts *ntask* copies of executable file *task*. *flag* allows to control the type of computer on which will be activated the processes.

  Predefined values of *flag* :

PvmDefault	PVM chooses the computers
PvmArch	*where* defines a target architecture.
PvmHost	*where* specifies a target computer.
PvmDebug	processes are activated in debugging mode.

  numt → # processes actually activated. Task identifiers → first *numt* positions of *tids(ntask)*. Error codes (neg. values) → last *ntask-numt* positions of *tids()*.

- **Getting the tid of the parent**: call **pvmfparent**(tid)

  On exit *tid* → *tid* of parent process , otherwise *tid* set to negative value *PvmNoParent*.

Interprocess communication

Communication between PVM processes based on message-passing. PVM provides asynchronous send, blocking receive, and nonblocking receive facilities. Sending a message is done in three steps (see Figure):

1. initialization of a send buffer and choice of an encoding format to send data; (*pvmfinitsend*)
2. packing of data to be sent into the send buffer (*pvmfpack*);
3. actual send/broadcast of the message stored in the send buffer to destination(s) process(es) (*pvmfsend, pvmfcast*).

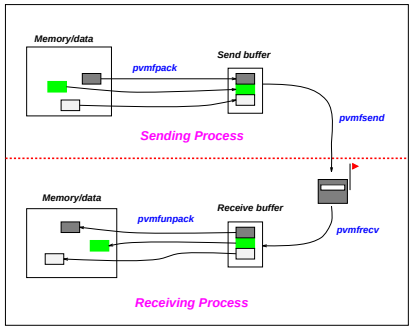


Figure 16: Illustration of send/receive main steps

- Main advantage of this strategy : user can compose his message out of various pieces of data and therefore decrease the number of messages effectively sent.
- With the broadcast option, only one send buffer has to be filled.
- If one large array of only a given data type needs to be sent (*pvmfpsend*) has been designed to pack and send data in one call to improve performance.

Reception is symmetric to the three step sending procedure After reception of message into the active buffer, data are unpacked into the destination arrays.

Various options to receive data are provided:

pvmfrecv	:	blocking receive
pvmftrecv	:	timeout receive
pvmfnrecv	:	nonblocking receive
pvmprecv	:	combines blocking receive and unpacking.

• **Management of buffers:**

Clear/initialize send buffer

call **pvmfinitsend**(encoding, bufid)

clears the send buffer and prepare it for packing a new message. Encoding scheme used during data packing defined by *encoding*.

Predefined values of *encoding* in FORTRAN:

PvmDefault	The XDR encoding used (heterogeneous network of computers).
PvmRaw	No encoding, native format of the host node.
PvmInPlace	Data are not copied into the buffer which only contains the size and pointers to the data.

Several buffers can be used simultaneously, but only one is active buffer for sending/receiving data. Procedures to create/release buffers (*pvmfmkbuf, pvmffreebuf*) to get/set the active send/receive buffer (*pvmfgetrbuf, pvmfgetsbuf, pvmfsetsbuf, pvmfsetrbuf*) are designed for this purpose.

•**Packing/unpacking data:**

call **pvmfpack**(what, xp, nitem, stride, info)

pvmfpack packs an array of data of a given type into the active send buffer. A message containing data of different types may be built using successive calls to *pvmfpack*. *nitem* elements chosen each *stride* elements of the linear array *xp* of type *what* are packed into the buffer.

Predefined values of *what* :

STRING, BYTE1, INTEGER2, INTEGER4

REAL4, REAL8, COMPLEX8, COMPLEX16

call **pvmfunpack**(what, xp, nitem, stride, info)

Similarly, *pvmfunpack* is used to unpack informations held into the active receive buffer. The unpacked data are then stored into the array *xp*.

• **Sending/Receiving messages:**

call **pvmfsend**(tid, msgtag, info)

pvmfsend sets the message label to *msgtag* then sends it to the pvm process of number *tid*.

call **pvmfmcast**(ntask, tids, msgtag, info)

pvmfmcast broadcast the message to *ntask* processes specified into the integer array *tids*.

call **pvmfnrecv**(tid, msgtag, bufid)

pvmfnrecv performs a non-blocking receive. If the message of label *msgtag* issued by process *tid* is not arrived then *bufid = 0*, otherwise the message is stored into a new buffer *bufid* automatically created. If *tid =-1* then the first message with label *msgtag* from any process will be received. If *msgtag =-1* the label is ignored.

call **pvmfrecv**(tid, msgtag, bufid)

pvmfrecv blocks the process until a message with label *msgtag* has arrived from *tid*. The other functionalities are similar to those of *pvmfnrecv*.

Check for arrived messages

call **pvmfprobe**(tid, msgtag, bufid)

If the message is not arrived then *bufid =0*, otherwise a buffer number is returned but the message is not received.

call **pvmfbufinfo**(bufid, bytes, msgtag, tid, info)

pvmfbufinfo returns the characteristics of the message stored in *bufid*: label *msgtag*, sending process *tid*, length in bytes *bytes*. *pvmfbufinfo* is particularly useful in conjunction with *pvmfprobe* or when the label –or the source– of the message to be received have not been specified.

Management of process group

The procedures for managing process groups form a layer on top of the PVM layer. They are provided into a separated library **libgpvm3.a**. A group server (**pvmgs**) is automatically activated at the first called to a procedure of the libgpvm3.a library.

Main characteristics of PVM groups:

- Any PVM process can join a group **pvmfjoingroup** and **pvmfleave**;
- A process can belong to several groups;
- A message can be broadcasted to a PVM group from any PVM process **pvmfbcast**
- Synchronization within a group can be performed using barriers **pvmfbarrier**.
- Useful procedures: **pvmfgettid, pvmfgetinst pvmfgsize, . . .**

PVM 3.3 has several collective communication routines such as **pvmfreduce()** that performs a global arithmetic operation (e.g. global maximum or global sum) across a group. Must be called by all processes in the group, final result is sent to a member specified as root. Gather/scatter routines are also available.

There is also a way of managing with system signals.

Manufacturer implementations of PVM

- Manufacturers (IBM, CRAY ...) often provide a tuned implementation of the PVM communication library on top of native communication calls, shared memory or virtual shared memory.

  → Portability and efficiency of parallel code

- Performance across networks of computers improved by using of Unix domain sockets between the tasks and the local daemon (improvement by a factor of 1.5 to 2).

- Using task-to-task direct communications (*PvmRouteDirect*) also increases communication performance.

9.1.3 Illustrative Example: a dot version of the matrix vector product

- Straightforward static parallelization of the matrix-vector product, $y = A \times x$ where A is a $(m \times n)$ matrix, x an n-vector, and y an m-vector.

- Master-slave paradigm: Each process is in charge of computing one block y. Additionally master process broadcasts the data to the slaves and collects the final results.

  The sequential FORTRAN code is:

```
do i = 1, m
   y(i) = 0.0D0
   do j = 1, n
     y(i) = y(i) + A(i,j) * x(j)
   enddo
enddo
```

In the parallel implementation,

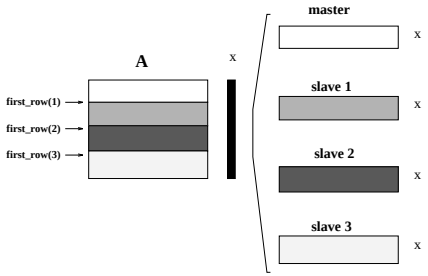


Figure 17: Static parallelization of the matrix vector product: A partitioned into block of rows distributed on the slave processes.

• Description of the master process (the master holds A and x)

enroll into PVM and create slaves
send data to the slaves
compute the first block of vector y
receive results from the slaves
leave PVM application

• Description of a slave process

enroll into PVM
wait for data from the master
compute my block of vector y
send back results to the master
leave PVM application

Computation of one block of y is performed using **_GEMV** from the Level 2 BLAS (see [10]) which performs:

$y = \alpha A x + \beta y$

Codes for master and slave processes

Master's code:

```
      PROGRAM dotmatvec
      integer slave_max, lda
      parameter (slave_max=32, lda=1000)
* PVM variables
      integer my_id, info, inst(slave_max), numt, bufid
     *        nb_of_slaves, no_slave, nb_of_processes, type,
     *        retcode, first_row(slave_max)
*
* Message types used:
* type = 0 to broadcast initial informations
*      = 1 to distribute data to the slaves
*      = 2 to receive results from the slaves

* Data declaration
      double precision a(lda,lda),x(lda),y(lda),one,zero
      integer          incx,n,m,i,j
      data             zero/0.0/, one/1.0/
      include '/usr/local/pvm3/include/fpvm3.h'

* Enroll this program into PVM
      call pvmfmytid(my_id)

* read input data (nb_of_slaves, m, n)
      read(*,*) nb_of_slaves, m, n

* initiate nb_of_slaves instances of slave program
      call pvmfspawn('slave',PVMDEFAULT,'*',nb_of_slaves,inst,numt)
      if (numt .ne. nb_of_slaves) stop
      nb_of_processes = nb_of_slaves +1
* Initialize data for computation
* and compute first_row(slave_no)
      do j=1,n
         do i = 1,m
           a(i,j) = DBLE(i+j)/DBLE(m+n) + one
         enddo
         x(j) = one + DBLE(j)/DBLE(n)
      enddo
```

```
      j = (m / nb_of_processes)
      do i=1,nb_of_slaves
        first_row(i) = i*j +1
      enddo
      first_row(nb_of_processes) = m+1
*     work balancing
      j = mod(m,nb_of_processes)
      do i=1, j-1
        first_row(nb_of_processes-i) =
     &       first_row(nb_of_processes-i) +j -i
      enddo

* broadcast the number of columns
* and x to each slave process
      type = 0
      call pvmfinitsend(PvmDefault, bufid)
      call pvmfpack(INTEGER4, n, 1, 1, info)
      call pvmfpack(REAL8,    x, n, 1, info)
      call pvmfmcast(nb_of_slaves, inst, type, info)
* send its sub-matrix data to each slave process
      type = 1
      do 60, no_slave = 1, nb_of_slaves
*       number of components computed by slave no_slave
        j = first_row(no_slave+1) - first_row(no_slave)
*       initialization of send buffer
        call pvmfinitsend(PvmDefault, bufid)
*       pack data into send buffer
        call pvmfpack(INTEGER4, j, 1, 1, info)
        call pvmfsend(inst(no_slave), type, info)
        do 70, i=1, n
           call pvmfinitsend( PVMDATADEFAULT, bufidS)
           call pvmfpack(REAL8,a(first_row(no_slave),i),j,1,info)
* send message stored in send buffer to slave inst(no_slave)
           call pvmfsend(inst(no_slave), type, info)
 70     continue
 60   continue

*  compute its part of the work
*     perform y <-- one*Ax + zero*y
*     where A is an matrix of order (first_row(1)-1) x n.
      incx = 1
      call dgemv('N',first_row(1)-1,n,one,a,lda,x,incx,zero,y,incx)

*  collect results of slave processes and quit PVM
      type = 2
      do 80, no_slave = 1, nb_of_slaves
*  j = number of components computed by the slave no_slave
        j = first_row(no_slave+1) - first_row(no_slave)
        call pvmfrecv(inst(no_slave), type, bufid)
        call pvmfunpack(REAL8,y(first_row(no_slave)),j,1,info)
 80   continue
      call pvmfexit(retcode)
      stop
```



```
      end
```

Slave's code:

```
      PROGRAM slave
*
      include '/usr/local/pvm3/include/fpvm3.h'
      integer from_tid, p_id, bufid, type, recvlen,
     *        my_id, info, lda
      parameter(lda=1000)
      double precision a(lda,lda),x(lda),y(lda),one,zero
      integer incx,n,m,i
      data    zero/0.0/, one/1.0/
* Enroll this program in PVM_3
      call pvmfmytid(my_id)
* Get the tid of the master's task id
      call pvmfparent(p_id)
* receive broadcasted data: number of columns and vector x
      type = 0
      call pvmfrecv(p_id, type, bufid)
      call pvmfunpack(INTEGER4, n, 1, 1, info)
      call pvmfunpack(REAL8   , x, n, 1, info)
* receive my block of rows
      type = 1
      call pvmfrecv(p_id, type, bufid)
      call pvmfunpack(INTEGER4, m, 1, 1, info)
      do 10, i=1,n
        call pvmfrecv(p_id, type, bufidR)
        call pvmfunpack(REAL8, a(1,i), m, 1, info)
 10   continue

*   perform matrix-vector on my block of rows
      incx = 1
      call dgemv('N',m,n,one,a,lda,x,incx,zero,y,incx)

*   send back results to master process
      type = 2
      call pvmfinitsend(PVMRAW, bufid)
      call pvmfpack(REAL8, y, m, 1, info)
      call pvmfsend(from_tid, type, info)

* leave PVM environment}
      call pvmfexit(info)
      stop
      end
```

Makefile for Compilation - Link

PvmArch and *PvmDir* correspond respectively to target computer and to location of the PVM library.

```
F77   = /usr/lang/f77
FOPTS = -O -u
# Specification of the target computer
```

```
PvmArch        =      SUN4
# Location of PVM libraries
PvmDir   =       /usr/local/pvm3/lib
# PVM libraries (C, FORTRAN, Group)
PVMLIB_C =       $(PvmDir)/$(PvmArch)/libpvm3.a
PVMLIB_F =       $(PvmDir)/$(PvmArch)/libfpvm3.a
PVMLIB_G =       $(PvmDir)/$(PvmArch)/libgpvm3.a
LIBS     =       $(PVMLIB_F) $(PVMLIB_C) $(PVMLIB_G)
# Location of the executable files
IDIR     = $(HOME)/pvm3/bin/$(PvmArch)
all : dotmatvec slave
dotmatvec : master.o $(BLAS) $(TIMING)
        $(F77) -o dotmatvec master.o $(LIBS) -lblas
        mv dotmatvec $(IDIR)
slave : slave.o $(BLAS)
        $(F77) -o slave slave.o $(LIBS) -lblas
        mv slave $(IDIR)
.f.o :
        $(F77) $(FOPTS) -c $*.f
clean :
        /bin/rm *.o
```

Configuration of the virtual machine

- Configuration file describe the list of computers used.
- Used to start the *pvmd3* daemon on each computer listed.
- Parallel Virtual Machine can be controled using the *pvm* console.
- Other solution use directly the *pvm* console to build the virtual machine.
- Both solutions illustrated in the following example : a parallel virtual machine of 4 RISC workstations (HP, IBM and two SUN) is build. The HP workstation, *pie*, is our host computer.

Example

```
pie> cat hostfile
# comments
pie
pinson
goeland
aigle
pie> pvmd3 hostfile &
pie> pvm
> conf
4 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED
                     pie    40000     HPPA    1000
                  pinson    80000     SUN4    1000
                 goeland    c0000     RS6K    1000
                 aigle     100000     SUN4    1000
```



```
pie> pvm
pvm> conf
1 host, 1 data format
                    HOST     DTID     ARCH   SPEED
                     pie    40000     HPPA    1000
pvm> add pinson goeland aigle
3 successful
                    HOST     DTID
                  pinson    80000
                 goeland    c0000
                 aigle     100000
```

9.1.4 Performance analysis and graphical interface

- Analysis of the efficiency of the parallel execution of a program complex problem by itself.
- Time measures and speed-up estimations often not sufficient to understand the behaviour of a parallel application.
- Automatic tracing of parallel execution → indispensable tool both to visualize the parallel behaviour and to debug.
- PVM allows to control which events are generated and where messages will be sent → quite complex to use.
- XPVM: tracing tool exploiting automatically features of PVM.
- PVM's tracing facilities generate extra traffic in the network → will perturb program execution.
- We show traces obtained during parallel execution of matrix-vector product. Target vitual machine = heterogeneous set of 4 RISC workstations. Master process located on computer node *rosanna*.

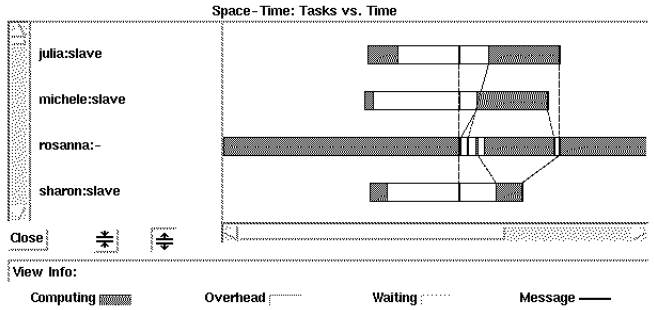


Figure 18: XPVM trace of the static parallelisation of the matrix-vector product

No exploitation of potential of fastest computer (*sharon*) idle most of the time.

9.2 MPI : standard pour le transfert de message

1. Effort de définition d'un standard de transfert de message pour les utilisateurs de développeurs
2. Objectifs :
   - portabilité, simplicité
   - utilisation plus large du calcul distribué
   - implantation par les constructeurs
   - Figé en 1994
3. Pour multiprocesseurs, clusters et réseaux de calculateurs

Accessible sur les sites **netlib** :

`http://www.enseeiht.fr`

ou sur le Web, voir par exemple :

`http://www.mcs.anl.gov/mpi/index.html`

Versions publiques disponibles: CHIMP (EPCC), DISI (Univ. Genova), LAM (Univ. Notre Dame), MPICH (Argonne Nat. Lab.), ...

Présentation largement inspirée de [12].

Caractéristiques de MPI

- Définition d'un processus MPI : groupe et numéro dans le groupe
- Message : contexte et un numéro de message relatif au contexte
- Contexte : entier utilisé pour définir des flots de messages indépendants
- exemple appel par une application parallèle d'une librairie effectuant des échanges de messages
- Buffers avec définition de structures, hétérogénéité des messages
- Echange de messages : bloquants, non-bloquants, synchrones, bufferisés
- Utilisable pour le calcul hétérogène
- Communication collectives et définition de sous-groupes
- Modèle de programmation SPMD, 125 fonctions dont 6 de base : MPI_Init, MPI_Finalize, MPI_Comm_size, MPI_Comm_rank, MPI_Send, MPI_Recv

Environnement

- enregistrement : call MPI_Init(info)
- terminaison : call MPI_Finalize
- contexte par défaut : MPI_COMM_WORLD : tâches numérotées 0, ..., N-1
- création de nouveaux contextes : définir des nouveaux groupes de processus et un nouveau contexte

Example : Hello world

```
#include "mpi.h"
#include <stdio.h>

int main( argc, argv )
int argc;
char **argv;
{
int rank, size
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
printf( "Hello world ! I'm %d of %d\n", rank, size );
MPI_Finalize();
return 0:
}
```

Exécution sous MPICH avec : **mpirun -np 2 hello**
printf exécuté sur les 2 processeurs requis.

Envoi / réception message : opérations de base

- envoi de nb données de type datatype rangées à @

  call MPI_Send (@, nb, datatype, dest, mess_id, context, info)

  et la réception

  call MPI_Recv (@, nb, datatype, source, mess_id, context, status, info)
- Broadcast : toutes les tâches émettent la requète, données envoyées par root

  call MPI_Bcast (@, nb, datatype, root, mess_id, context, info)
- Opération collective :

  call MPI_Reduce (@, results, nb, datatype, operation, root, context, info)
- datatype peut être prédéfini (MPI_real) ou défini par l'utilisateur
- context : défini le groupe de tâches et le contexte

Types de communication

- asynchrone, bloquants : MPI_Send et MPI_Recv
- non bloquants : MPI_Isend, MPI_Irecv, MPI_Wait
- bloquants (rendez-vous) : MPI_Ssend, MPI_Srecv
- bufferisés : MPI_Bsend (l'utilisateur spécifie le buffer)
- Communication globales
  - barrières
  - broadcasts
  - scatter / gather, all_to_all
  - réduction : max global, somme globale, ...

Topologies

On peut définir une topologie pour un ensemble de processus → permet d'identifier ses voisins

Topologies disponibles :

- Grille (Cartesian mesh) : MPI_Cart_create, MPI_Cart_coords (coordonnées d'un processus dans la grille
- Autres topologies disponibles (tores, ...)

Types de données

- Elémentaires : existant en C ou Fortran (MPI_INT, ...)
- Vecteurs : données séparées par un stride constant
- Accés indirect par un tableau d'indices (gather/scatter)
- Structures définies par l'utilisateur : en spécifiant le nombre d'éléments, la distance entre ces éléments et leur type

  MPI_Type_structure(nb, array_of_len, array_of_displs, array_of_types, &newtype)

Objectifs de MPI-2

- gestion dynamique de processus
- extensions temps réel
- client/serveur
- put/get
- C++ et Fortran 90

9.3 PVM versus MPI

- PVM très répandu mais MPI résultat d'un effort de standardisation
- Constructeurs supportent à la fois PVM et MPI mais PVM est plus ancien
- Pas de gestion de tâche en MPI (création, destruction, allocation, ...) mais SPMD largement utilisé
- Pour HPF : MPI est intéressant par sa richesse
- MPI très riche (126 fonctions) : 24 façons d'envoyer un message, quelques fonctions suffisent la plupart du temps
- MPI plus adapté aux MPP mais PVM plus adapté aux réseaux hétérogènes mais différences de performance disparaissent
- Conclusion
  - MPI : richesse des communications point-à-point + communications globales non-bloquantes + topologie virtuelle de processeurs
  - Absence dans MPI de : gestion de processus, d'accés à des mémoires distantes, de gestion de threads
  - PVM : tolérance aux fautes, gestion des tâches et modèle MPMD
  - **MPI-2** devrait mettre tout le monde d'accord.

10 Concepts avancés

10.1 Introduction

- Répartition :
  - Peut être construite par ajout d'outils et de services permettant l'interopérabilité entre machines et systèmes hétérogène
  - Utilisation de systèmes d'exploitation conçus pour la répartition comme **Mach** et **Chorus** → environnement homogène (même système d'exploitation)
- Convergences :
  - Threads pour la gestion du parallélisme
  - Intégration des services dans des architectures cohérentes
  - Approche objet pour le gestion des ressources
- Internet computing : servlets, applets, ...
- Objets répartis

10.2 Systèmes d'exploitation répartis ([14])

On se limite à évoquer **Mach** et **Chorus** .

- Organisés autour d'un *micro-noyau* fournissant des services de base :
  - Ordonnancement des tâches (threads ou activité)
  - Gestion de la mémoire virtuelle
  - Communication entre activités quelle que soit leur localisation (IPC)
- Autour du noyau on trouve des *serveurs* fournissant un ensemble de services modulaires. Ces serveurs permettent de construire un système d'exploitation complet.
- IPC réalisent la communication entre noyau et serveurs.

Quelques définitions :

- *Thread* : entité minimale active
- *Task* ou *acteur* : unité de gestion de ressources (mémoire, ports de communication, ...) s'exécutant dans un espace mémoire protégé. Une tâche contient un ou plusieurs threads. Processus UNIX = tâche. Tous les threads appartenant à une même tâche partagent ses ressources.
- *Port* : point d'accès à un canal de communication (généralisation de la notion de port TCP/IP).
- *Objet* : toutes les ressources du système sont vues comme des objets y compris la mémoire. Les objets communiquent en envoyant des messages via des ports. C'est un serveur de messages qui prend en charge les communications qui peuvent être locales ou distantes.
- NB : il existe même un serveur de mémoire réseau (le "mappeur" de **Chorus**) qui permet de partager la mémoire à travers le réseau.

Mach

- Développé à Carnegie-Mellon
- **Mach** = Multiple Asynchronous Communication Host
- Peut être utilisé sur un monoprocesseur, un multiprocesseur ou un réseau de machines
- Compatibilité avec environnement UNIX
- Bibliothèque pour la manipulation des threads (POSIX)
- On peut aussi utiliser les primitives classiques UNIX pour la commmunication

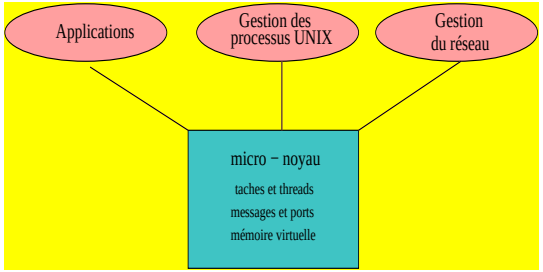


Figure 19: **Mach**.

- MIG (**Mach** Interface Generator) permet d'engendrer les applications utilisant les RPC propres à **Mach**.

Chorus

- Système d'exploitation réparti développé et vendu par **Chorus**
- Structuré autour d'un micro-noyau orienté temps réel ou systèmes répartis.
- Introduit la notion d'acteur superviseur qui s'exécute dans un espace d'adressage propre avec des instructions privilégiées.
- **Chorus** fournit un mécanisme plus simple que **Mach** pour désigner les ports et les objets

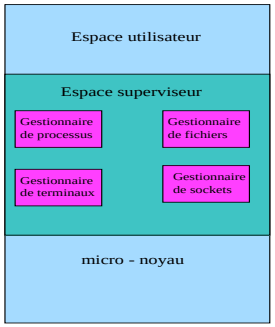


Figure 20: **Chorus** .

Aspects système du Web : mécanismes de cache

- Intérêts d'un cache

- Introduit un niveau intermédiaire, d'accès rapide (car local)
- Réduit temps moyen d'accès en conservant informations les plus utilisées
- Réduit le trafic entre les niveaux de stockage de l'information

• Web bien adapté

- Informations changent peu souvent dans la plupart des cas
- On peut regrouper les demandes :
  * Cache individuel sur disque
  * Cache local pour un département
  * Cache régional

• Problèmes

- Choix des informations à conserver
- Politique de mise à jour du cache en particulier quand il est plein
- Rafraîchissemnt des informations
- Coopération entre caches

Gestion des caches

• **Politique de remplacement**

- FIFO : dans l'ordre des arrivées
- RANDOM : choisir un document au hasard
- SIZE : éliminer le document le plus gros, gestion à court terme
- LRU (Least Recently Used) : hypothèse de localité, fréquemment utilisé

• **Cohérence** : comment garantir que les documents sont à jour ?

- Invalidation : le serveur prévient le cache quand l'original est modifié : idéal mais coût gestion par le serveur qui doit garder trace des copies
- TTL (Time To Live) : durée de vie limitée (élimination ou rappel à la date d'expiration)
- Durée de vie proportionnelle à l'age du document

• **Coopération entre caches**

- Hiérarchie : tout cache à un parent auqeul il transmet la requête s'il ne peut la résoudre, et ainsi de suite, si pas de parent contacter le serveur puis réponse au fils éventuel
- Entre égaux : un cache transmet la requête aux caches frères et au serveur : il prend la première réponse
- Mode de coopération pas fixé et peut dépendre de la nature des requêtes

10.3 Objets répartis ([13], [4])

• Motivations

- Fournir un mode d'organisation des applications réparties privilégiant le partage d'informations réparties sur plusieurs sites entre des utilisateurs eux-mêmes répartis
- Améliorer structuration et réutilisation des programmes
  → facilité de compréhension et de modification, constructions génériques

• Principes : définition d'un ensemble d'objets répartis, utilisables via leurs méthodes avec les conditions d'accès suivantes

- Transparence de la localisation : objet désigné par un nom logique indépendant de sa localistion physique (localisation peut changer sans que nom change)
- Transparence d'accès : accès à un objet distant identique à accès à un objet local

• Utilisations possibles : applications mettant en œuvre des données réparties que l'on veut rendre globalement accessibles

- Edition coopérative
- Ingénierie coopérative
- Documentation
- ...

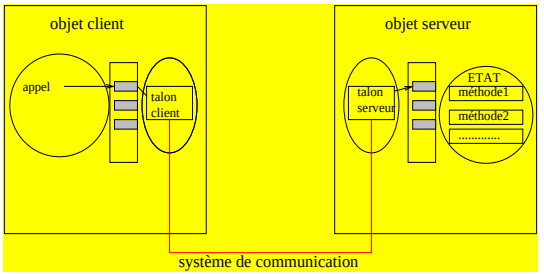


Figure 21: Exemple de client/serveur à objet.

• Invocation d'un objet :

- Référencce d'objet (pointeur universel)
- Identification d'une méthode
- Paramètres d'appel et de retour : passage par valeurs ou par référence

- Objets d'un langage :
  - Représentation propre au langage : instance d'une classe
  - Exemple : Java RMI
- Objets système :
  - Représentation arbitraire définie par l'environnement d'exécution
  - Exemple : CORBA

Notions de base

- Encapsulation :
  - Un objet "encapsule" un *état* (ensemble de données), accessibles uniquement au moyen d'un ensemble de fonctions (*méthodes*) qui constituent l'interface de l'objet
  - L'interface définit tout ce qui est nécessaire à l'utilisation de l'objet : on peut remplacer une réalisation par une autre en respectant l'interface

10.4 Applications mobiles ([2])

- Utilisation d'ordinateurs portables avec connexion sans fil de plus en plus fréquente
- Environnement très différent de celui des stations de travail
- Contraintes de taille et de poids → ressources disponibles limitées
- Source d'énergie limitée (batterie) et niveau de consommation très variable en fonction des périphériques
- Réseaux sans fil = faible bande passante et périodes de déconnexion (interférences, zones d'ombre' ...)
- Utilisation de protocoles réseaux pas toujours suffisantes pour masquer les problèmes liés aux communications
- Quelques solution spécifiques pour adapter le fonctionnement des applications aux environnements mobiles :
  - Algorithmes de préchargement de données dans le acahe d'un mobile en prévision de sa déconnexion
  - Oracle Lite version allégée du système de gestion de bases de données Oracle proposant de nouvelles de cohérence tenant compte des clients mobiles
  - ...

- Cadre de conception et de réalisation spécifique pour les applications mobiles permettant de rendre le cœur d'une application indépendant des aspects mobile qui peuvent être isolés.
- Caractéristiques de mobiles :
  - Station de travail portable : proche des postes fixes mais encombrantes (2Kg) et autonomie limitée (< 4h)
  - PC de poche : petit écran souvent monochrome, processeur peu performant, capacité mémoire faible (8 - 16 Mo de mémoire flash remplaçant RAM et disque), peu encombrants (< 500 g) et grande autonomie (3 - 55h).
  - Peuvent être relié à un réseau filaire ous ans fil (GSM ou réseaux à petite échelle de type Wavelan).
  - GSM ≈ 9600 bps et fréquentes déconnexions.
  - Wavelan ≈ 2 Mbps mais très variable, plus fiables.
- Quelques propriétés utiles voire indispensables :
  - Offrir une connexion durable même en présence de déconnexion transitoires.
  - Assurer qu'un message envoyé est reçu en respectant l'ordre chronologique des messages.
  - Plate-forme permettant l'exécution de calculs provenant d'un système distant.

10.5 Codes mobiles ([20])

- Programmes pouvant se déplacer d'un site à l'autre
- Exemple : applet Java.
- Motivations : rapprocher le traitement des données → réduction du volume de données échangées sur le réseau et moindre charge des serveurs
- Caractéristiques :
  - Code interprétable
  - Sécurité
  - Schémas d'exécution à base de code mobile

Modèles d'exécution pour la mobilité

- Code à la demande :
  - Mobilité faible (code exécutable sans contexte)
  - Exemple : Applet Java

- Agents mobiles
  - Mobilité faible : code exécutable plus données modifiées
  - Exemple : “aglets”.
  - Mobilité forte : code exécutable + données + contexte d'exécution
  - Exemple Agent Tcl.

11 Problème de la répartition ([17])

11.1 Introduction

- Problématique née avec l'idée de faire communiquer des machines via un réseau, par exemple avec des échanges de message
- Développement et programmation d'applications réparties → langages, systèmes d'exploitation, environnements
- Difficultés à développer une application répartie :
  - Pas d'état global (état d'une autre machine ?)
  - Pas d'horloge globale (horloge propre à chaque machine)
  - Fiabibilité toute relative (certaine tolérance aux défaillances)
  - Sécurité relative (plus diffcile à protéger qu'une architecture centralisée)
  - Non-déterminisme dans l'exécution des appplications

Avantages de la répartition

- Partage de ressources et de services

  Exemple : gestion de fichiers répartis service de base des systèmes d'exploitation répartis
- Répartition géographique : répartition essentielle pour accéder aux moyens locaux nécessaires tout en gardant accès aux ressources et services distants
- Puissance de calcul cumulée, disponibilité, flexibilité
- **Définition ([17]) :** La répartition est la mise à disposition d'un ensemble de ressources et de services connectés via un réseau pour tous les usagers possédant un droit d'accès en un point quelconque

11.2 Solutions au problème de la répartition

Outils théoriques

- Modélisation de façon abstraite et formelle des proriétés d'un traitement réparti

- Algèbres de processus communicants avec communication sous forme de rendez-vous point-à-point
- Concept d'acteur : protocole de communication entre acteurs asynchrone, chaque acteur a une boîte à lettres pour la réception des messages
- ...

Algorithmique répartie

- Problèmes bien spécifiques posés par les architectures réparties
- Définition de protocoles de communication point-à-point et de diffusion → formalismes de description (automates communicants, réseaux de Pétri, ...) et outils d'aide à la validation. Quelques standards existent (appels de procédure à distante) mais protocoles souvent adaptés à une classe d'applications.
- Problèmes généraux (exclusion mutuelle, interblocage, atomicité, réplication, ...) ou issus de la répartition des traitements et des données (terminaison d'une application, réalisation d'un consensus, ...)

Langages de programmation

- A priori une interface de programmation (**API**) permettant échange de message devrait suffire (e.g. sockets)
- Introduction de structures de contrôle pour faciliter la programmation :
  - non-déterminisme en réception avec possibilités d'associer à chaque type de message attendu une action spécifique
  - Appel de procédure à distance (modèle client-serveur) : problème il faut introduire un langage de définition de l'interface (**IDL**) pour appeler les procédures distantes → génération automatique du traitement des appels côté client et serveur.

Systèmes d'exploitation

- Ils assurent entre autre l'interface avec le réseau de communication
- Deux approches possibles
  - Conception de nouveaux noyaux d'exécution répartie en utilisant des micros-noyaux (gestion mémoire + périphériques, parallélisem et communication). Les autres services (gestion de fichiers par exemple) → services hors du noyau (e.g. Chorus et Mach)
  - Extension des systèmes d'exploitation centralisés en ajoutant au moins une interface de communication et quelques services répartis (gestion de fichiers), par exemple UNIX avec introduction sockets, RPC, puis NFS. Avantages : continuité et réutilisation mais moins modulaire.

Environnements d'exécution répartie

- Problème de base des systèmes répartis : prise en compte de l'hétérogénéité matérielle et logicielle
- Objectif : faire communiquer et coopérer des composants hétérogènes
- Modèle adopté : schéma de communication client / serveur et notion de *bus logiciel*. *Bus logiciel* permet d'accéder à des services spécifiés par leur interface (enregistrées dans des annuaires permettant de trouver le ou les nœuds serveurs.
- Norme de fait : **CORBA** (Common Object Request Broker) défini par l'OMG (Object Management Group Architecture), c'est un bus logiciel à objet qui se place entre le système d'exploitation et les applications.

11.3 Conception d'un système réparti

- Concepteurs cherchent souvent à concevoir un système réparti qui a l'air centralisé
- On cherche à masquer certaines des difficultés due à la répartition (mais impossible à masquer totalement)
- Propriétés de transparence permettent de masquer tout ou partie de la répartition des données et des traitements

1. **Transparence d'accès** : accès identique que la ressource soit locale ou à distance
2. **Transparence de localisation** : désignation de la ressource indépendante de sa localisation (les usagers peuvent ignorer sa localisation réelle). Transparence localistion + accès on peut utiliser la ressource en ignorant si elle est locale ou à distance
3. **Transparence du partage** : accès concurrents à une ressource contrôlés afin que son intégrité soit garantie (pour un fichier assurer les règles de synchronisation du lecteurs/rédacteurs, pour une impimante ne pas mélanger les impressions).

   Les systèmes assurent le minimum vital.

   Systèmes dédiés à un contexte particulier offrent parfois cette transparence (base de données réparties par exemple avec atomicité des transactions)
4. **Transparence de la réplication** : assurer que l'accès à une ressource soit identique quelle que soit la forme d'implantation de cette ressource en particulier répliquée

   Dédié à des systèmes très spécifiques : tolérants aux fautes par exemple.
5. **Transparence aux fautes** : assurer une bonne tolérance aux défaillances des services sur un système réparti.

6. **Transparence de la migration** : assurer qu'une ressource pourra migrer d'un nœud à l'autre sans que les usagers s'en aperçoivent (en particulier migration de processus → régulation de charge)
7. **Transparence de charge** : régulation de la charge des nœuds → exploitation plus efficace. Problème : connaissance de l'état global du système (difficle à obtenir).
8. **Transparence d'échelle** : architecture répartie plus modulaire et adpatable qu'une architecture centralisée (ajout de nœud sans arrêt du système. Mais passage de 10 à 100 sites pas toujours transparents pour les utilisateurs.

11.4 Représentation d'un calcul réparti

- Application répartie structurée en un ensemble fixe de processus
- Processus communicant : unité de répartition
  - Encapsule un ensemble de variables locales dont les valeurs définissent l'état courant du processus
  - Comportement : exécute séquentiellement une suite d'instructions (atomiques). Exécution du processus = suite d'évènements dont des envois et réception de messages.
  - Identification : identification par processus (nom symbolique – e.g. URL – ou numéro IP).
  - Connaissane locale : un processus n'a qu'une coannaissance très partielle du calcul global, on admet qu'il connaît son identification, ses voisins via les canaux de communication et son état interne
- Communication par messages :
  - Echanges de messages via canaux logiques point à point (asynchrones, uni/bidirectionels, FIFO i.e. respectant la chronologie d'envoi en réception, ...)
  - Peut être représenté sous forme de graphe (sommets = processus, arêtes = canaux)

- (e1,r1) message point à point
- diffusion avec émission e2
- perte de message e3
- ...
- Calcul réparti représenté sous forme d'un ensemble d'évènements produits par chaque processus (évènements internes ou envois ou réceptions de messages).

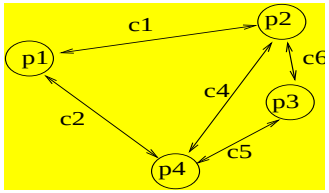


Figure 22: Exemple de représentation graphique d'un calcul réparti.

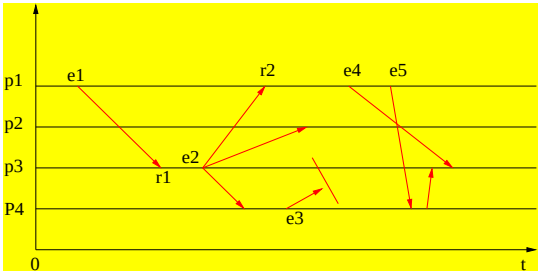


Figure 23: Exemple de chronogramme.

- Les évènements issus de processus différents ne sont pas forcément ordonnés mais ordre partiels induit par les messages échangés.
- Ordre partiel fondé sur une relation de *causalité*.
- Relation notée $\prec$ satisfaisant :
  1. Pour tout couple d'évènements (*e*,*e'*) issu d'un même processus telque *e* précède *e'* dans la suite associée au processus $e \prec e'$.
  2. Pour un échange de message entre 2 processus (envoi = *e* et réception *r*), on a $e \prec r$.

Protocoles ordonnés

Message *m1* reçu près *m2* par le processur *p3*, alors que un lien causal existe en émission : *e1* ($\prec$ *r* $\prec$) *e2* .

Eventuellement, le message peut être incompréhensible.

Protocoles ordonnés évitant ce type de problème

- Protocole ordonné d'ordre causal assure la propriété suivante pour toute destination S :

$$\forall m, m' \quad \text{vers} \quad S \quad : \quad e_m \prec e_{m'} \Rightarrow r_m \prec d_{m'}$$

- Diverses implantations de ces protocoles sont possibles.

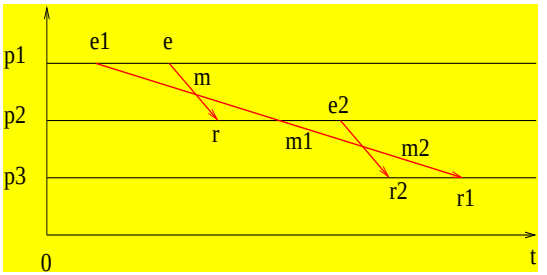


Figure 24: Exemple de liaison causale en émission.

- Contrôle pour la délivrance d'un message toujours local au site récepteur.

Problèmes posés par les protocoles de diffusion

- Intérêt de la répartition : duplication des traitements et des données → services plus fiables et plus disponibles
- Diffusion → complexité croissante des échanges de messages
- Problèmes majeurs :
  1. Séquentialité : les messages successifs émis par un nœud seront-ils reçus dans le même ordre par les nœuds visés par la diffusion ?
  2. Atomicité : tous les sites recevront-ils chaque message diffusé ?

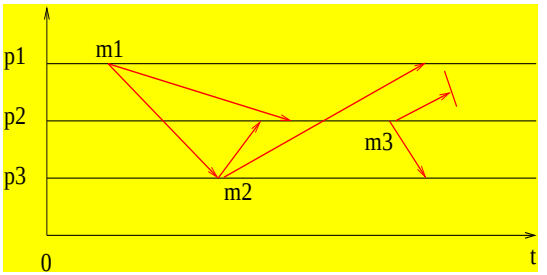


Figure 25: Exemple de problèmes posés par l'utilisation d'un protocole de diffusion à faible coût.

- Diffusion vers groupes de processus utile pour réaliser des applications robustes et à haute disponibilité (redondance des ressources)
- Par exemple dans un client-serveur classique : 2 serveurs jumeaux.

- Pb : diffusion des requêtes vers les deux serveurs en garantissant qu'elles vont arriver dans le même ordre et qu'elles seront prises en compte (ou ignorées par les deux serveurs).
- **Synchronisme virtuel** : modèle d'exécution fournit des protocoles vérifiant ces proprriétés (systèmes ISIS ou HORUS).
  - Calcul réparti virtuellement synchrone garanti que les processus d'un même groupe peuvent être cibles de diffusion totalement ordonnées atomiques.
  - Réalisation d'un noyau d'exécution virtullement synchrone implique deux types de primitives :
    * Primitives de gestion de groupes : entrée / sortie et connexion en tant que client à un groupe
    * Primitives de diffusion

11.5 Abstractions de niveau plus élevé

- Communication en mode message = niveau d'abstraction peu élevé
- D'où la proposition de mécanismes de communication plus élaborés :
  - Appel de procédure à distance pour les traitements
  - Pour les donnés communication par mémoire partagée ou par fichiers partagés.
  - **Notion de mémoire partagée répartie** ou **mémoire partagée virtuelle** ou **mémoire partagée distribuée** :
    * Objectif fournir un espace d'adressage global (modèle de programmation centralisé) : LINDA, BBN, KSR, ...
    * Difficulté de réalisation sur une architecture distribuée : éviter une trop forte synchronisation des accès à cette mémoire partagée répartie
    * Utilisation de la réplication → augmentation du parallélisme d'accès à la mémoire
      cohérence des copies !!!
    * Souvent implantation au dessus de transferts de messages (hard/soft)

Mémoire partagée virtuelle
Approches

- Modèles à cohérence faible : BBN
- Modèles basés sur un mécanisme de cache ou de pagination mémoire avec répertoires distribués : KSR, Convex SPP, ...
- Modèles à espace de tuples :
  - Base de données (tuples) partagée

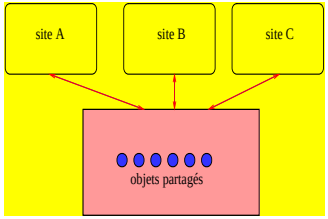


Figure 26: Mémoire partagée virtuelle.

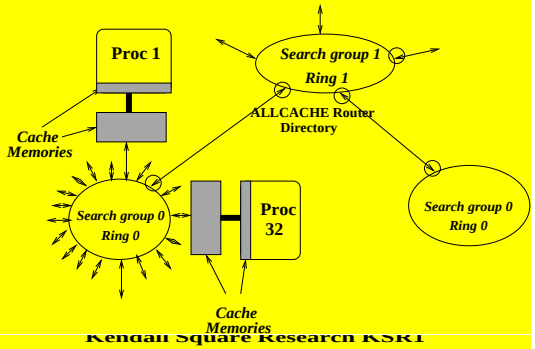
  - Modèle de programamtion à la Linda (dépôt, retrait et consultation d'objets)
  - Exemple : JavaSpaces
- Modèles à objets répartis partagés
  - Espace d'objets répartis partagés
  - Langage à objets extension d'un lange existant : expression de la distribution, parallélisme, synchronisation, ...
  - Désignation universelle d'objets
  - Gestion du partage des objets : synchronisation, cohérence (image unique d'un objet ↔ copies cohérentes)
  - Divers modes de réalisation
    * Objets répliqués (Javanaise)
    * Objets à image unique (Guide)

Exemple d'architecture à mémoire partagée virtuelle : Kendall Square Research KSR1

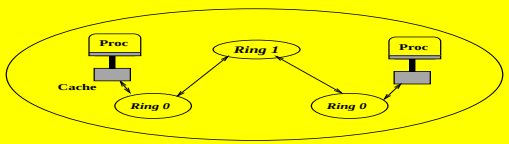
- Architecture
  - organisée en anneaux de 32 processeurs
  - mémoires locales des processeurs gérées comme des caches

Virtual Shared Memory Computers

Kendall Square Research KSR1



Kendall Square Research KSR1



ALLCACHE architecture
Two levels of rings
32 Processor/Ring

Peak Perf:
1.28Gflops/Ring
32 rings: 40.96Gflops

Processor
RISC (cycle time 50ns
20MIPS
40Mflops
.5 MB Sub−cache
32 MB Cache memory

Memory	Data location	Latency (cycles)
.5MB	Sub−cache	2
32MB	cache	18
992MB	#Proc, same ring 0	175
32GB	Proc on #ring 0	600
	Page fault disc	400K

Hiérarchie mémoire

- Registres : 64 registres 64-bits dans l'unité flottante
- Sous-cache :
  - Cache données et cache instructions de 256 Koctets
  - Latence : 2 cycles (0.1 microsec.)
  - Taille de la ligne de cache : 64 octets
  - 2-way set associative, random (et pas LRU), write-back
- Cache local :
  - 32 Moctets
  - 16-way set associative
  - Latence : 20 cycles (1 microsec.)
  - ligne de cache de 128 octets chargée de l'anneau
  - Plusieurs stratégies lorsque le cache est plein

Programmation de la KSR1

- Deux niveaux de parallélisme :
  - Parallélisme sur les processus :
    - outils UNIX classiques
    - Communication inter-processus avec les mécanismes usuels (pipes, sockets, mémoire partagée, streams)
  - Parallélisme sur des threads (processus légers)
    - Thread : unité d'exécution
    - Thread : flot séquentiel d'instructions dans un processus
    - Processus : nombre arbitraire de threads partageant un même espace d'adressage
  - Parallélisme fonctionnel (MIMD) entre processus
  - Parallélisme multi-thread interne à un processus
  - Mémoire partagée virtuelle
    - tous les threads d'un processus partagent le même espace mémoire virtuel

Parallélisation

- Parallélisation automatique avec le préprocesseur KAP
- Utilisation de directives
- Appels à la librairie sur gérant les threads
  - Primitive similaire à un fork

- Locks

Tiling loops

- Utilisé pour paralléliser les boucles imbriquées

```
c*ksr* tile ( index list, options, ... )
       do
          do
             ...
          enddo
       enddo
c*ksr* end tile
```

- Options :
  - order = order_list
  - private = variable_list
  - lastvalue = variable_list
  - reduction = variable_list
  - tilesize = tilesize_list
  - strategy = slice, mod, grab, wawe
  - numthreads = numthreads or teamid = team_id
  - affmenber = 0 or 1
- Directives peuvent être insérées automatiquement par KAP ou par le programmeur
- Parallel region

```
c*ksr* parallel region (  numthreads = ..,
                          private = variable_list )
       ...
c*ksr* end parallel region
```

- Exemple : Code séquentiel

```
      do j = 1, n
         do i = 1, m
            a(i,j) = b(i,j)
         enddo
      enddo
```

Parallélisation avec 4 threads (copies des boucles do) sur des tranches de la boucle en j

Code parallèle

```
c*ksr* parallel region ( numthreads = 4,
c*ksr*&                  private = (myid, j1, j2, i, j) )
       my_id = ipr_mid()
       j1 = 1 + ( my_id * (n/4) )
       j2 = ( my_id + 1 ) * ( n/4 )
       do j = j1, j2
          do i = 1, m
             a(i,j) = b(i,j)
          enddo
       enddo
c*ksr* end parallel region
```

- Parallel section

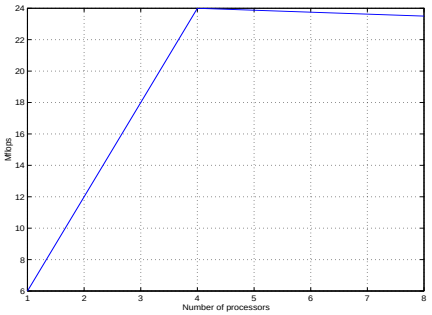
```
c*ksr* parallel sections ( teamid = ...,
                           private = variable_list )
c*ksr* section
       code segment 1
c*ksr* section
       code segment 2
       ...
c*ksr* end parallel sections
```

- Données partagées et privées
  - Par défaut toutes les données et les commons sont partagés sauf spécification dans une liste 'private'
  - Les variables index de Tile sont tjs privées
  - procédure appelée à l'intérieur d'un constructeur parallèle : les variables indéfinies en entrée sont privées à chaque appel
  - idem pour les commons (peuvent être déclarés private dans un constructeur parallèle)
- Locks disponibles (pthread_mutex_lock and pthread_mutex_unlock), ainsi que barriers
- Optimisations :
  - pcsp : anticipe le chargement d'une sous-page dans le cache local
  - pstsp : diffuse une copie read-only d'une sous-page à toutes le processeurs qui en possèdent une copie)

Parallélisation automatique réalisée par KAP sur SGEMM

```
c*ksr* tile (j, private=(temp,i,l) )
       do j = 1, n
          do i = 1, m
             C(i,j) = beta*C(i,j)
          enddo
          do l = 1, k
             temp = alpha*B(l,j)
             do i = 1, m
                C(i,j) = C(i,j) + temp*A(i,l)
             enddo
          enddo
       enddo
c*ksr* end tile
```

Performance sur des matrices 512-par-512



Multiplication matricielle optimisée

- Parallélisation sur des sous-matrices
- Code séquentiel optimisé → proche de 30 MFlops
- Pas de tableaux de travail privés

Version parallèle

```
      nblig = int(m/nb)
      nbcol = int(n/nb)
c*ksr*  tile (i1,j1,tilesize=(i1:1,j1:1), strategy=mod,
c*ksr*& private(i,j,l,ib, kb, jb))
      do 60 j1 = 1, nbcol
         do 70 i1 = 1, nblig
            i = (i1-1)*nb + 1
            j = (j1-1)*nb + 1
            ib = min( m-i+1, nb)
```



```
            jb = min( n-i+1, nb )
            do 50 l = 1, k, nb
               kb = min( k-l+1, nb )
               call sgemm_tuned_serial(
     $              'n', 'n', ib, jb, kb, alpha, a(i,l), lda,
     $               b(l,j), ldb, beta, c(i,j), ldc )
50          continue
70       continue
60    continue
```

Performance de la version parallèle

Computer	Precision	Uniproc.	Number of processors					
			1	2	4	8	16	24
KSR1	64 bits	27.5	25.4	42.9	81.9	165.4	305.4	418.3

Table 3: Performance in Mflops of GEMM with matrices of order 512 on a KSR1.

References

[1] P. Amestoy, M. Daydé (2001). Calcul Réparti, Cours 3ème Année Informatique, INPT-ENSEEIHT.

[2] F. André, M.-T. Segarra (2000). MolèNE : un système générique pour la construction d'applications mobiles. Calculateurs Parallèles, Réseaux et Systèmes Répartis, **12**, 1/2000, 9–29.

[3] Ph. D'Anfray (1996). Une Présentationde MPI. PARANOTES, Avril 96.

[4] R. Balter. Modes de structuration d'applications réparties. Université J. Fourier, Grenoble, http://sirac.imag.fr.

[5] Beguelin, A., Dongarra, J., Geist, A., Manchek, R. a,d Sunderam, V. (1991). A User's Guide to PVM Parallel Virtual Machine, Tech. Rep. ORNL/TM-11826, Oak Ridge National Laboratory, Oak Ridge, Tennessee.

[6] Beguelin, A., Dongarra, J., Geist, A., Manchek, R. a,d Sunderam, V. (1995). Recent Enhancements to PVM, Int. Journal of Supercomputer Applications, **9**, 108–127.

[7] Max Buvry, Support de cours Base de Données, 1ère année Télécommunications et Réseaux, 2001.

[8] V. Charvillat et Romulus Grigoras, Un peu plus loin avec les technologies multimédia, Polycope ENSEEIHT, 2001.

[9] J.J. Dongarra (1992). An overview of High-Performance Computers and Performances Issues, Lecture Notes, CERFACS Training Cycle.

[10] J. J. Dongarra and Du Croz, J., and S. Hammarling and R. J. Hanson (1988). "An extented set of Fortran Basic Linear Algebra Subprograms, ACM Trans. Math. Softw., **14**, pp. 1-17 and 18-32.

[11] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V. (1993). PVM 3 User's Guide and Reference Manual, Tech. Rep. ORNL/TM-12187, Oak Ridge National Laboratory, Tennessee.

[12] W. Gropp (1999). Tutorial on MPI : The Message-Passing Interface, MCS, Argonne National Laboratory, IL, USA.

[13] Sacha Krakowiak, *Introduction aux Systèmes et Réseaux Informatiques*, Université J. Fourier, Grenoble, http://sirac.imag.fr.

[14] Michel Gabassy et Bertrand Dupouy, *L'Informatique Répartie sour UNIX*, Collection de la Direction des Etudes et Recherches d'Electricité de France, Eyrolles, 1992.

[15] Chuck Musciano et Bill Kennedy, *HTML et XHTML, La référence*, O'Reilly, Paris, 2001.

[16] Gérard Padiou, *Systèmes Opératoires*, Cours et notes de cours, 2ème année Informatique et Mathématiques Appliquées, ENSEEIHT, Toulouse.

[17] Gérard Padiou, *Précis de répartition : définition et problématique*, Cours et notes de cours, 3ème année Informatique et Mathématiques Appliquées, ENSEEIHT, Toulouse.

[18] Equipe Systèmes Opératoires, *Sysèmes Opératoires – Système UNIX*, Travaux Dirigés et Travaux Pratiques, ENSEEIHT, Toulouse.

[19] Jean-Marie Rifflet, *La communication sous UNIX*, Collection Informatique, Ediscience International, 1992.

[20] Michel Riveill, *Construction d'applications réparties - Introduction*, Notes de Cours INPG / ENSIMAG, 1999.