INTRODUCTION A LA

PROGRAMMATION DISTRIBUEE

Des mécanismes IPC… à la programmation de services distants…

Jean-Charles Fabre

LAAS-CNRS

OBJECTIFS ET PLAN

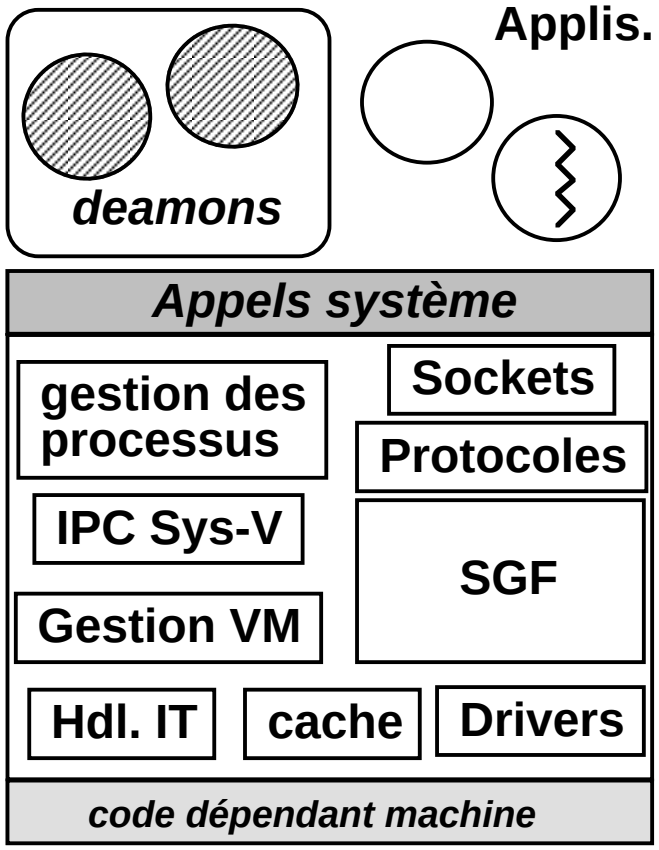
1) Communication et synchronisation inter-processus
   - communication et synchronisation
   - passage de message
2) Communications à distance : l'exemple des sockets
3) Réalisation de serveurs
   - simple
   - par polling
   - à la demande
   - multithreadés
4) Introduction à la communication par appel de procédure distantes

COMMUNICATION ET SYNCHRONISATION

INTER-PROCESSUS

Notions de base

Architecture classique de système opératoire



- **applications** sous forme de processus

  - séquentiel
  - multi-activités (notion de *thread*)

- **services**

  ➠ *deamons* (processus de niveau applicatif remplissant un service générique)

  ➠ noyau (ensemble de composants logiciels fournissant les fonctionnalités de base)

- **processus et threads**

  - éléments de structuration du contrôle et de l'activité des applications

Notions de base

Architecture de système opératoire nouvelle génération

- sous-systèmes

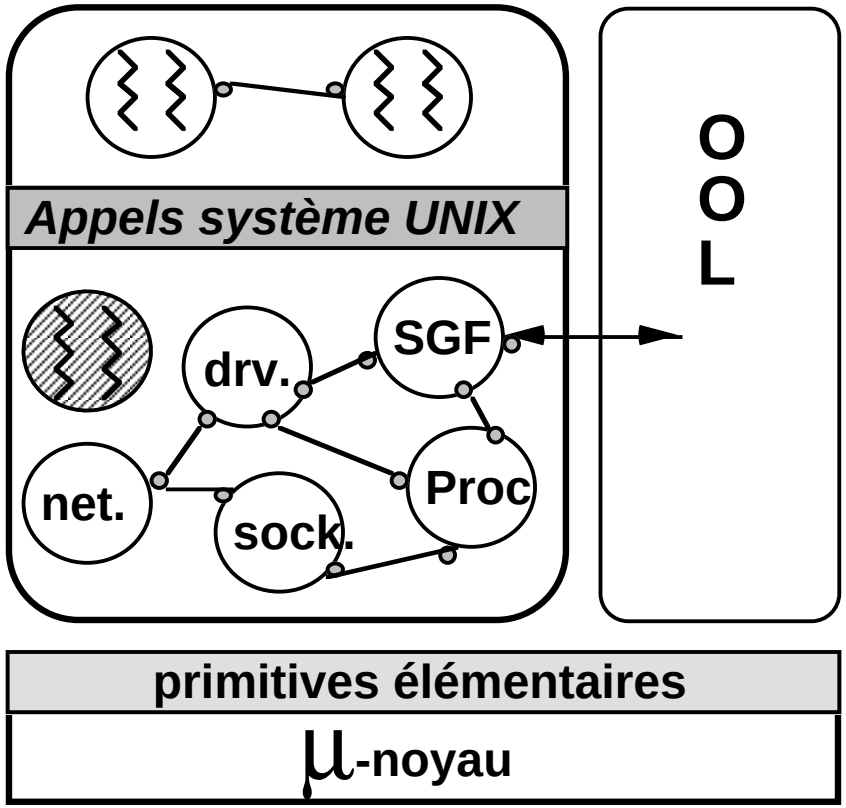
- unix
- Object Oriented Layer (CORBA)

- micro-noyau

- processus (multi-threads)
- ports / IPC
- scheduling, synchro, mémoire

- avantages:

- répartition
- modularité
- adaptativité

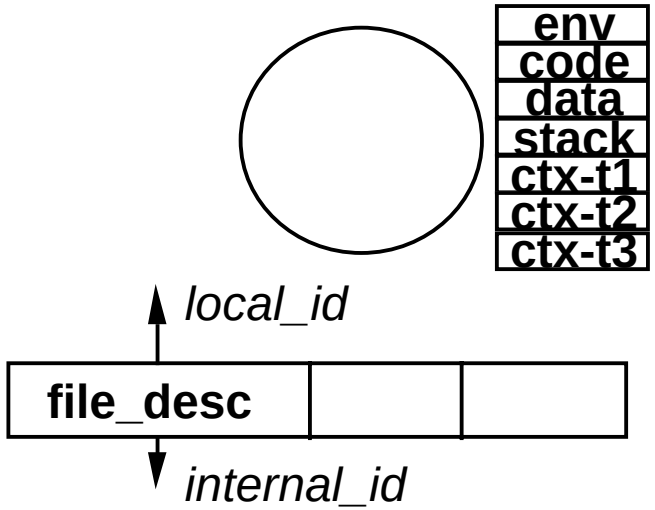


Notions de base

Notion de processus

Un processus *(aspect dynamique)* correspond à l'exécution d'un programme binaire ou shell script *(aspect statique)*. Le noyau gère les processus dans des tables où il conserve les informations relatives à :

- identifications (PID, propriétaire...)
- environnement (variables)
- espace mémoire (données, code, pile)
- contexte machine (registres, pile)



Un processus réagit à :
➪ séquence d'évènements (interruptions, exceptions)

Un processus utilise :
➪ fichiers ouverts (input, output, error)

L'ordonnancement et les classes de processus :

- temps partagé (classe TS) : quantum de temps attribué à chacun
- priorité fixées (classe RT) : ordonnancé selon leur priorité ou leur échéance

Les types de processus

- système ➠ services généraux
  - init/getty ⇨ connections
  - lpd ⇨ impressions
  - cron ⇨ actions périodiques
  - inetd ⇨ services réseau
- utilisateur ➠ applications

Autres notions apparentées : lightweight, tasks, threads

Exemple:

boot ➠ init ➠ getty ➠ login_shell (local)
➠ rlogind ➠ login_shell (distant)

Une application UNIX ⇨ un ou plusieurs processus coopérants

Notion de thread :

- Processus léger qui s'exécute dans un processus
- Plusieurs processus légers peuvent s'exécuter de façon concurrente
- Les blocs d'instructions au sein d'un thread peuvent accéder de façon simultannée à des ressources partagées ➪ notion de section critique

Généralités sur les mécanismes de communication

Les mécanismes de communication inter-processus

Des moyens de communications purement locaux, a priori :

- fichiers : communication séquentielle!!!
- tubes : communication asynchrone (producteur-consommateur)
- mémoire partagée (tampons)
- files de messages

Des moyens de synchronisation locaux:

- sémaphores (notion de sections critiques)

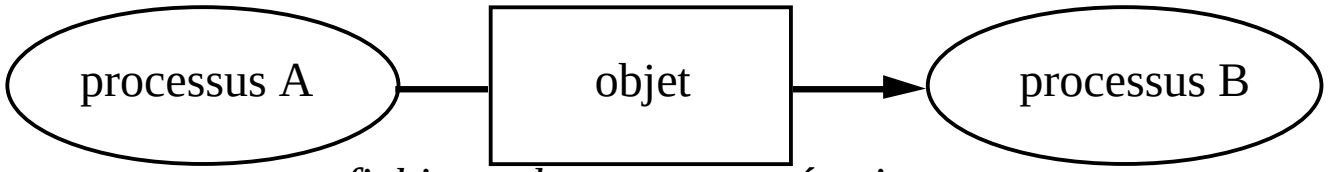
Des mécanismes de communication inter-processus distants

- protocoles UDP et TCP/IP
- moyens de communications par message : sockets, TLI

Remarque : dépendance / fonctionnement du système (e.g. UNIX, filiation et héritage)

Communication locales

Un seul Ecrivain ➡ un seul Lecteur



fichier, tube, tampon mémoire

Exemples:

- Ecrivain > F ; Lecteur < F (séquentiel par fichier)
- Ecrivain | Lecteur (notion de tube)

Règle de synchronization : *principe du producteur - consommateur*

- le processus écrivain est suspendu quand le tube est plein ;
- le lecteur est suspendu quand le tube est vide.

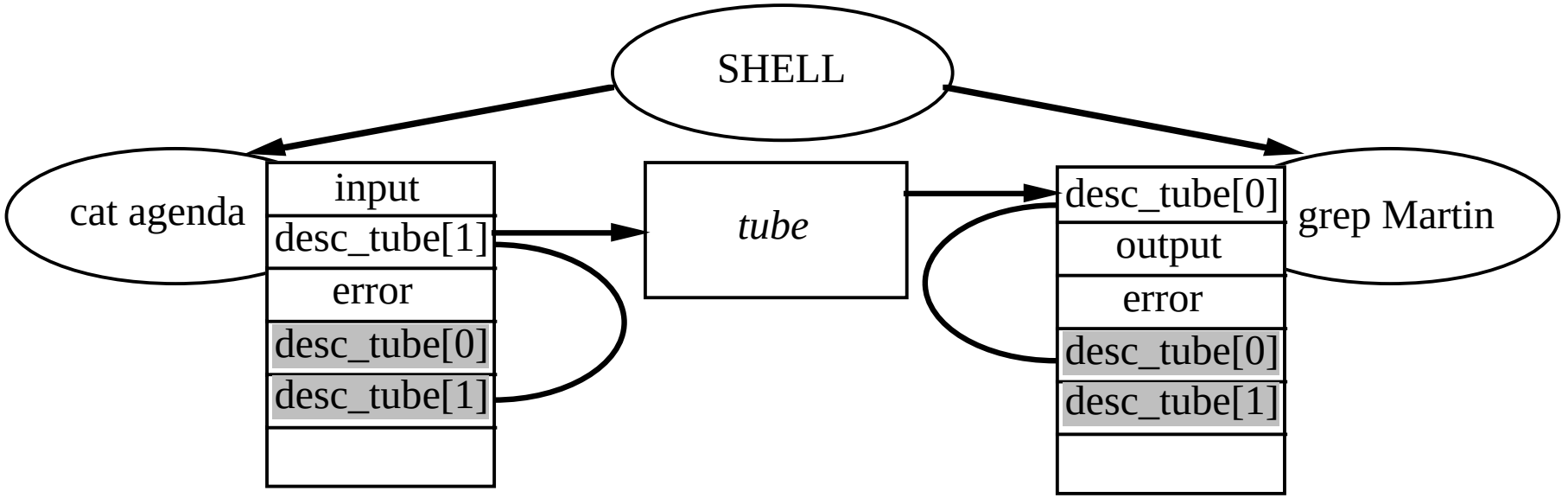
La communication est non structurée par flot de caractères et la synchronisation est *gratuite* puisque effectuée par le noyau.

Exemple de communication/synchronisation par pipeline

SHELL : cat agenda | grep Martin

Tube Unix basé sur la notion d'héritage d'environnement (table des descripteurs)

- Création du pipe line ➠ descripteur[0] (lecture), descripteur[1] (écriture)
- Création des deux processus « cat » et « grep » par un appel système fork



Communications locales multiples

N Ecrivains ➪ M Lecteurs

Synchronisation des accès multiples :

- Les processus sont concurrents pour accéder à l'objet (temps partagé)
- Synchronisation = exclusion mutuelle d'accès ➪ cohérence et intégrité des données
- Différents mécanismes (masquage IT, sémaphores...)

Problème de base sur un exemple générique :

table et pointeur (entrée_libre) partagés

PRINCIPE ➪allouer(entrée) ; entrée:=entrée+1 ; écrire(valeur, entrée)

P1 : allouer (entrée) = 36 processus suspendu ➪ **P2** entrée_libre = entrée_libre + 1 table(entrée) = "mickey"	P2 : allouer (entrée) = 36 entrée_libre = entrée_libre + 1 processus suspendu ➪ **P1** table(entrée) = "donald"

Résultat: entrée libre = 38 ; valeur(36)=Donald ; valeur(37)=?????

Résultat correct : entrée libre =38 ; valeur (36) = mickey ; valeur (37) = donald

Synchronisation des accès concurrents

"Race conditions" (course) ➪ "Critical sections" (sections critiques)

Principes de l'exclusion mutuelle :

- Pas deux processus simultanément dans une section critique
- Pas d'hypothèse sur la vitesse relative des processus
- Aucun processus suspendu hors section critique ne peut en bloquer un autre

Les programmes P1 et P2 sont des exemples de sections critiques : atomicité de quatre actions. Dans cet exemple les deux processus P1 et P2 sont simultanément en section critique

Des mécanismes :

- (Busy Waiting) "P1 est occupé en SC, donc P2 attends!"
- Masquage des IT : utilisé dans le noyau, mais pas une solution générale pour les processus utilisateur
- Variables verrouillées : Test-and-Set = opération H/W de lecture/mise-à-jour indivisible

Sémaphores

Opérations indivisibles permettant d'autoriser l'accès à une donnée partagée à un ou plusieurs processus (notion de nombre **de tickets d'autorisation**) ;

- un processus qui n'a pas obtenu de ticket est suspendu
- un processus libère un ticket lorsqu'il sort de la section critique.

Autres méthodes : compteurs d'événements, moniteurs (évitement des deadlocks)

Exemple de synchronisation par Sémaphores

Production d'items dans une table à N entrées et consommation des items

Producteur:	Consommateur:
int element production (element) DOWN (vide) DOWN (element) ecrire (item) UP (mutex) UP (plein)	int element DOWN (plein)) DOWN (mutex) lecture (element) UP (mutex) UP (vide) consommer (item)

Définitions :

- mutex, vide et plein sont des sémaphores

Initialisations :

- mutex=1 ; vide=N ; plein=0

Discussion :

- DOWN(idem P) : décrémentation d'un sémaphore (valeur négative implique blocage du processus)
- UP(idem V) : incrémentation d'un sémaphore (valeur positive implique réveil d'un processus)
- PROBLEME : inversion d'accès aux sémaphores dans un processus ➪ DEADLOCK

Remarque : Les opérations sur les sémaphores telles que DOWN et UP sont des appels systèmes

Exemple de synchronisation par Moniteurs

Moniteur Producteur_Consommateur

conditions plein, vide; integer compte; procedure METTRE; si compte=N alors wait(plein) ajouter (element); compte=compte + 1; si compte=1 alors signal(vide) fin_proc	procedure ENLEVER; si compte=0 alors wait(vide) retirer (element); compte=compte - 1; si compte=N - 1 alors signal(plein) fin_proc compte=0 Fin_Moniteur

Commentaires :

- les procédures METTRE et ENLEVER sont utilisées pour accéder à la table partagée
- METTRE et ENLEVER sont exclusives
- un processus suspendu libère l'accès au moniteur
- exclusion d'accès réalisée par un sémaphore (compilation/génération)
- WAIT: attente sur une condition ; SIGNAL: réveil de processus en attente sur condition
- Exclusion mutuelle automatique d'accès au opérations ➪ opérations atomiques.

Les Moniteurs sont une construction "langage" ; très peu de langages offrent des moniteurs, mais Java offre une construction équivalente : **synchronize**

Synchronisation par messages

La synchronisation par message est aussi valable entre processus sur une même machine

Gestion des objets

- Les données sont propres aux processus et accessibles au travers de fonctions (objet)
- encapsulation ➪ liberté de représentation interne ➪ souplesse / adaptativité

Initialisation de la communication

- Etablissement de communications unidirectionnelles et bidirectionnelles
- Indépendance vis-à-vis de la filiation : pas de lien entre processus communicants

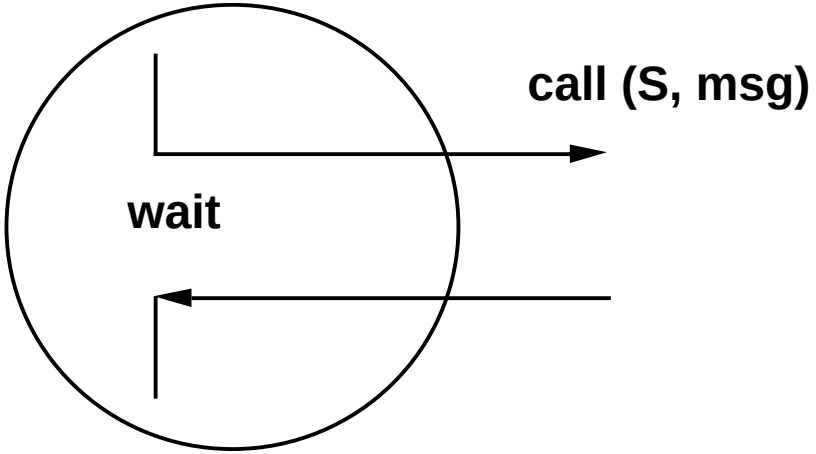
Quelques définitions...

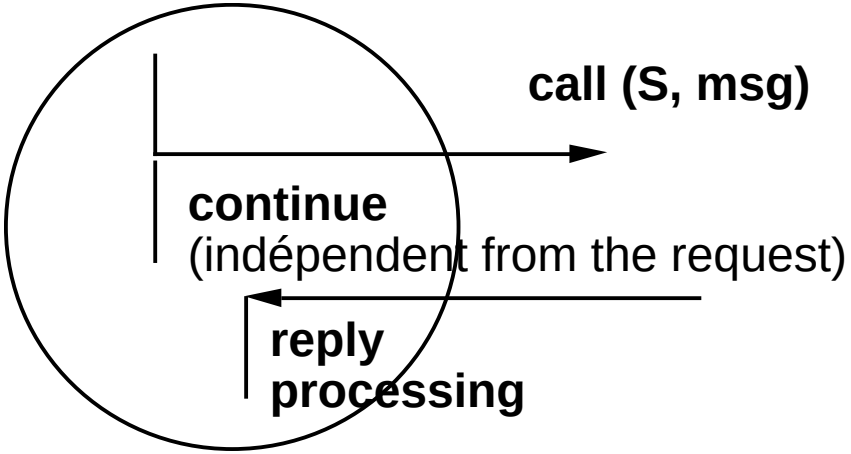
- Notion de Client : processus qui appelle par RPC des fonctions réparties
- Notion de Serveur : processus qui offre des fonctions réparties appelables par RPC.

Mode de fonctionnement d'un client

Deux types possibles d'appel de service :

- Le mode synchrone : appel + blocage + réception réponse
- Le mode asynchrone :
  - ➢ appel + poursuite de l'exécution
  - ➢ événement signalant réception de la réponse ➪ suite du traitement



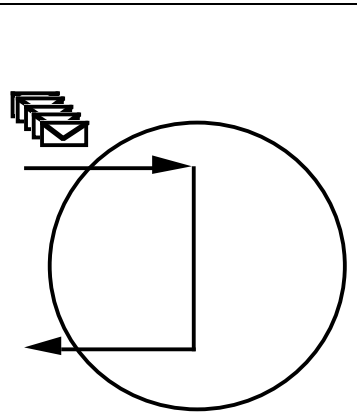
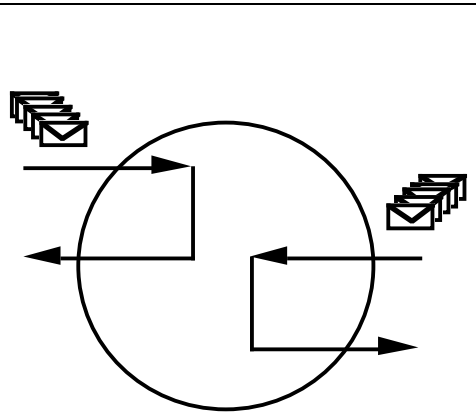
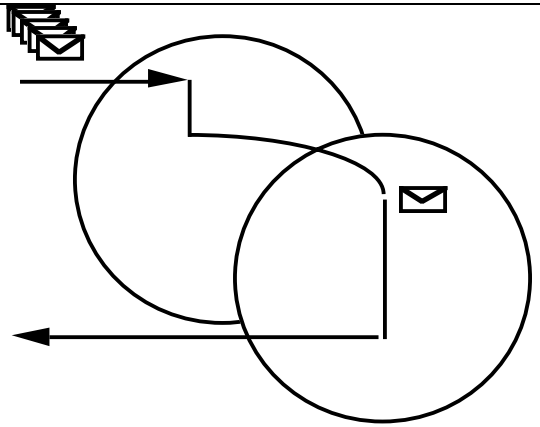
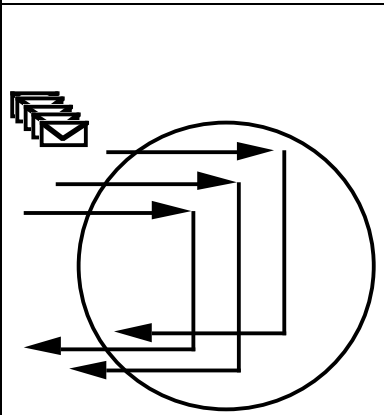


Remarque :
La construction asynchrone est plus complexe, mais plus performante

Mode de fonctionnement d'un serveur

Mode séquentiel ➪ exclusion des fonctions dans le contexte du serveur

Mode multi-tâche ➪ sections critiques : données locales partagées!!!

Simple	*Polling*	*A la demande*	*Multithread*
			

Les requêtes client peuvent être :

- Simple : mode datagramme
- Fiables: mode connecté

Tous les modes peuvent être implémentés selon ces deux protocoles

Exemples de mécanismes IPC de base dans UNIX

Les tubes

- Création : pipe (file_desc) + Re-directions Unix (pipes Shell) / fork+exec
- Communication par flot de caractères : read/write (file_desc, buffer, compte)

La mémoire partagée

- Allocation d'une entrée dans la table des descripteurs de mémoire partagée ➪ id
- Attachement d'une entrée à un processus ➪ @ segment mémoire (virtuelle)
- Accès direct au segment : pointeur sur des variables structurées ➪ mapping

Les sémaphores

- Allocation d'une entrée dans la table des descripteurs de sémaphores ➪ id
- Initialisation ➪ compte
- Fonctions de contrôle (protection...), d'incrémentation et de décrémentation

Les messages

- Création d'une file de message : identifieur d'un descripteur de file de message
- Fonctions de contrôle, d'émission et de réception de messages typés

Mécanismes de communication/synchronisation locaux

Principe :

- Chaque IPC est associé à une table du noyau.
- Chaque entrée contient une clé (key) associée : identificateur utilisateur.
- Chaque IPC est créé par une opération "get" qui retrouve ou alloue une entrée en fonction de la clé et de flags, et possède une opération de contrôle "ctl" permettant des manipulations d'état.

Le noyau renvoie un index (modulo le nombre d'entrées).

Chaque IPC est associé à un ensemble de permissions similaires à celles pour les fichiers.

Chaque entrée contient des informations d'état : nom des processus, dates...

Une commande de contrôle permet d'accéder à l'état d'un IPC, de le modifier ou de le détruire.

Des opérations spécifiques sont associées à chaque type d'IPC.

Exemple : les sémaphores sous UNIX

Généralisation des sémaphores de Dijkstra: N sémaphores avec incrémentation/décrémentation > 1

Un sémaphore correspond aux éléments suivants :

VALEUR	cette valeur est a priori un entier positif manipulée par P et V
PID_LAST	numéro du dernier processus ayant manipulé le sémaphore
NB_WAIT	nombre de processus en attente d'une incrémentation du sémaphore

Les opérations sur les sémaphores sont les suivantes :

Semget : création d'une liste de sémaphores ➪ semid

Semctl : manipulation des paramètres associés (propriétaire, droits)

Semop : opérations sur un (ou plusieurs) sémaphores :
Incrémentation, Décrémentation

Remarques :

- La création permet de créer un ensemble de sémaphores
- Les opérations s'effectuent sur plusieurs sémaphores de façon atomique
- Un sémaphore est un objet associé à un propriétaire et à des droits type Unix

Création d'un sémaphore

semid = **semget**(key, count, flag)

key = clé permettant de rechercher une entrée existante

count = nombre de sémaphores associés à cette entrée

flag = code de création (IPC_CREAT) et protection (octal)

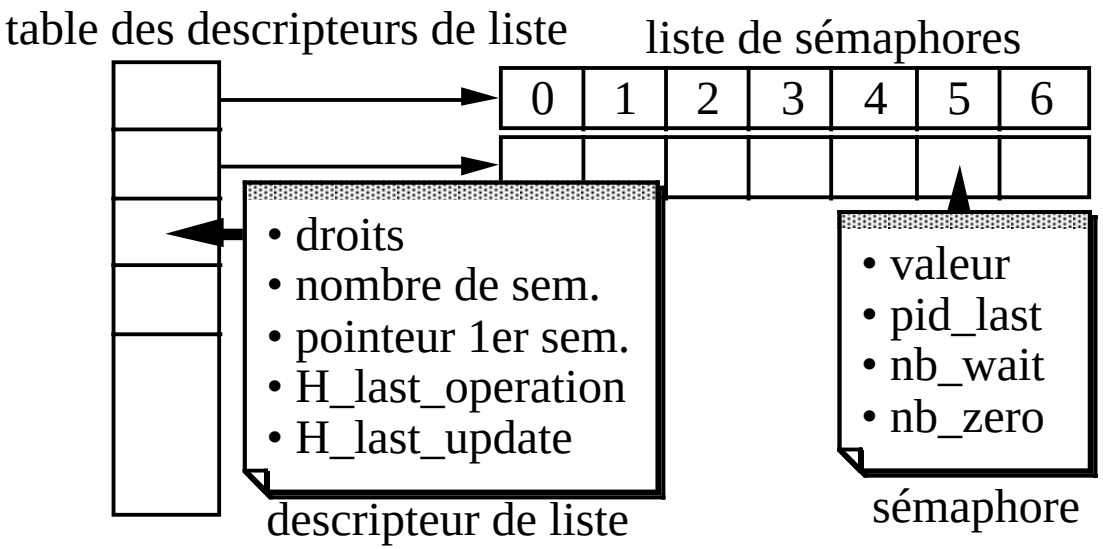
Exemple: semid = semget(SEMKEY, 2, 0777 | IPC_CREAT)

avec SEMKEY= valeur entière fixée ou attribuée par le noyau (IPC_PRIVATE)

et | le OU INCLUSIF bit-à-bit avec un code IPC_CREAT (création de nouveau sémaphore) ou IPC_EXCL (pas de semid retourné si le sémaphore SEMKEY existe)

Retour ➪ semid est un identifieur de descripteur de liste de sémaphores

Structures de données noyau



- les opérations peuvent être effectuées sur un ou sur plusieurs sémaphores
- toutes les opérations sont effectuées ou aucune (atomicité)
- il existe donc une notion d'annulation de modification
- opération non bloquante (retour appelant ➪ errno=-1)

Opérations P et V

Les processus manipulent les sémaphores au travers de la primitive **semop**.

oldval = semop(semid, oplist, count)

semid = identificateur de liste de sémaphore
oplist = pointeur sur un tableau d'opérations sur sémaphore
count = taille du tableau d'opérations

La commande **semop** permet donc d'appliquer de manière atomique un ensemble d'opérations sur une liste de sémaphores.
La valeur de retour oldval correspond au numéro du dernier sémaphore sur lequel a été effectué une opération.

Structure d'une entrée du tableau pointé par oplist :
1) numéro d'un sémaphore dans la liste identifiée par semid

2) opérations P (nombre négatif),V (nombre positif) et comparaison à zéro

3) flags
- IPC_NOWAIT : retour code d'erreur au lieu du blocage en attente
- SEM_UNDO : inversion des opérations effectuées par un processus

Logique du fonctionnement des opérations sur sémaphore :

- vérification par le noyau des droits et de la légalité des numéros de sémaphores dans oplist.
- Puisqu'une opération peut s'effectuer sur plusieurs sémaphores (évitement de *deadlock*), il faut qu'il existe une structure de données (noyau) permettant de sauvegarder les valeurs précédentes. Notion de restauration : le noyau restaure les valeurs précédentes si échec
- semop avec IPC_NOWAIT ➪ pas d'attente si échec de l'opération : code d'erreur en retour
- **Principe P :** si nb_tokens >= 1 alors { décrémenter ; continue P } sinon { décrémenter puis mettre P en attente }
- **Principe V :** si nb_tokens = 0 alors { réveiller P premier en attente} sinon { incrémenter }

Opérations UNDO

Le problème :

Supposons qu'un processus ait bloqué une ressource partagée en ayant effectué une opération P sur un sémaphore et que ce processus effectue un exit (erreur, signal non masquable).

➪ Tous les autres processus trouveront le sémaphore bloqué !!!

La solution : utiliser le flag SEM_UNDO lors de l'appel à semop.

Lors de l'exit de P, le noyau effectue dans l'ordre inverse les opérations effectuées par P

Mise en œuvre de la solution

- le noyau maintient une table avec une entrée pour chaque processus dans le système.
- chaque entrée dans la table pointe sur des structures UNDO (une par sémaphore)
  - ID du sémaphore (liste de sémaphores)
  - numéro du sémaphore dans la liste
  - valeur d'ajustement
- allocation dynamique des entrées (premier semop avec le flag SEM_UNDO)
- pour chaque opération semop avec SEM_UNDO positionné:
  - recherche du sémaphore
  - valeur d'ajustement = valeur d'ajustement - valeur de l'opération sur le sémaphore

(valeur d'ajustement = somme négative des opérations effectuées avec SEM_UNDO)

Exemple (en pseudo code)

Déclarations	
#include <sys/ipc.h>, <sys/sem.h>, …	*Inclusion de fichiers systèmes*
#define SEMKEY 75	*Déclaration de nom **global** de sémaphore*
#define SHMKEY 75 (mémoire partagée)	*Déclaration de nom **global** de zone de mémoire partagée*
int shmid, semid;	*Déclaration de variables ➪ noms **locaux***
{ **Code du programme**	
semid=**semget**(SEMKEY,1,0777|IPC_CREAT);	*Creation sémaphore*
semctl(semid,0,SETVAL,1)	*Initialisation à 1*
semop(semid,&psembuf,1);	*Appel de P (structure psembuf à +1)*
Section critique : access a SHMKEY	
semop(semid,&psembuf,1);	*Appel de V (structure vsembuf à -1)*
}	

Les files de messages

Transmission locale de messages inter-processus

- Le noyau gère des files de messages identifiées par une clé
- Des permissions sont associées à une file de message pour déposer, retirer un message

Les appels systèmes

• Création d'une file ou accès à une file existante :

msgget (key, flag) ➪ msqid

• Gestion des informations d'état et destruction :

msgctl (msqid, cmd, mstatbuf).

• Emission de messages :

msgsnd (msqid, msg, count, flag).

• Réception de messages :

msgrcv (msqid, msg, maxcount, type, flag).

Création de files de messages

Déclarations	
#include <sys/ipc.h>, <sys/msg.h>, …	*Inclusion de fichiers systèmes*
#define MSGKEY	*Déclaration de nom **global** de file*
struct message	*Déclaration de variable message*
{ long mtype;	
char mtext[20]; } bufsnd	
int msqid ;	*Déclaration de variables ➪ noms **locaux***
{ **Code du programme Emetteur**	{ **Code du programme Récepteur**
semid=**msgget**(MSGKEY, IPC_CREAT);	semid=**msgget**(MSGKEY,IPC_CREAT);
R=**msgsnd**(msqid,bufsndp,20, IPC_NOWAIT);	R=**msgrcv**(msqid,bufrcvp,20, bufrcv.mtype,IPC_NOWAIT);
Emission de message (20 c)	*Réception de message (20c, type)*
}	}

Les problèmes des IPC System V

Identification globale des IPC (System V) pour tous les processus d'un SITE

Gestion et connaissance globale des numéros associés

Edition de lien statique vis-à-vis des objets IPC

Edition de lien dynamique ➪ serveur de numéro ➪ désignation symbolique...

- Communication IPC (System V) locales à un seul site
- Destruction d'un processus par SIGKILL ➪ structures IPC orphelines!

MAIS...

- La programmation système nécessite des mécanismes de synchronisation et de communication efficaces et performants
- Mise-en-œuvre modulaire et multi-processus asynchrones ➪ parallélisme (pseudo), programmation dynamique, performances

COMMUNICATIONS A DISTANCE

Communications distantes

Les processus A et B sont installés sur des machines distinctes inter-connectées par un réseau

Il peut donc y avoir des messages perdus:

- mode **datagramme** : pas de contrôle de réception
- mode **connecté** : accusé de réception (renvoi d'un accusé + time-out et retransmission)
- perte d'un accusé de réception : numéros de séquence associé aux messages

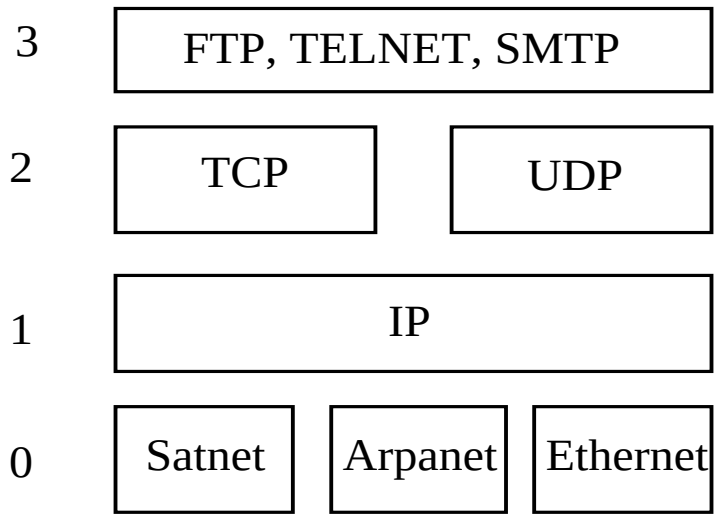
Les processus A et B doivent être désignés de manière unique:

- A@site1, B@site2 ou A@site.domaine1, A@site.domaine2

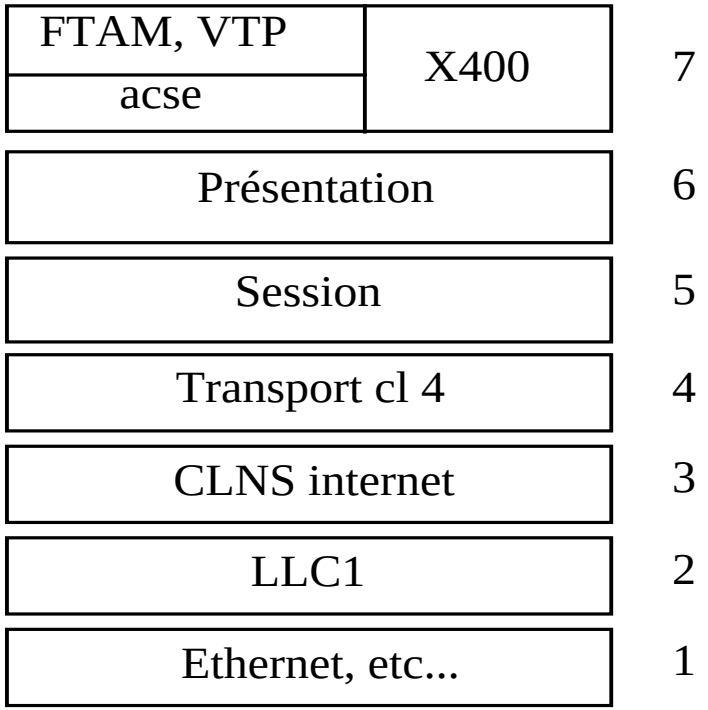
Les appels de procédures à distance (RPC) basées sur l'échange de messages : transparence

Introduction aux réseaux ouverts: ISO, TCP/IP

Les protocoles TCP/IP



Les couches ISO



- *Defense Advanced Reaserch Project Agency* : ARPANET ➪ TCP/IP
- *Berkeley* : TCP/IP sous UNIX

☞ IP: routage, adressage, frag./réass.

☞ UDP: transport (n° port, @IP)

☞ TCP: protocole orienté connexion (fiable)

- TCP/IP très répandu et utilisé
- Applications sur TCP/IP ≈ applications ISO
- mécanisme des sockets sur TCP/IP
  ....mais aussi potentiellement sur X25, ISO

Les sockets BSD / UDP, TCP/IP

Mécanisme de boîtes à messages de base

1) Etablissement d'un *endpoint* protocole ➪ socket_id (identificateur de boîte à message);

2) Attachement d'un descripteur ➪ @ (association d'une adresse visible par tous)

3) Initialisation ➪ nombre de messages en attente de demande de connexion;

4) Acceptation de connexions : TCP entraîne la création d'un canal privé client-serveur
   - TCP : de la part d'un client distant ➪ **accept**

5) Connexion à un socket distant correspondant à une demande de connexion client
   - TCP : **connect** ➪ à destination d'un serveur distant

6) Fonctions d'émission, de réception, de sélection : celles-ci dépendent du mode
   - UDP (**sendto**, **recvfrom**)
   - TCP (**send, recv, read, write**)

Principes des "sockets" BSD

Les Sockets permettent des communications IPC intra et inter-sites d'un réseau

Plusieurs Domaines de Communication:

- Domaine Unix : adressage par nom de fichier ou message
- Domaine InterNet :adressage par "points finaux de communication" (endpoints)

Plusieurs protocoles

- Datagramme :pas de garantie de séquencement, réception, duplication (UDP)
- Streams : transmission de messages séquencée et fiable (TCP)

Les fonctionnalités pour réaliser des services selon le principe du Client - Serveur

Création d'une structure socket	Attente d'une nombre max de requête
Attachement à une adresse	Emission / Réception / Acceptation de rqt
Etablissement de connexion sur socket	Accès logique aux adresses de services

L'architecture de communication

Les sockets mode stream

- Flot de caractères bi-directionnel séquencés, fiables non dupliqués
- Pas de frontière d'enregistrement
- Notion de paire de socket mode stream ➪ analogue au tube

Les sockets mode datagram

- Flot de caractères bi-directionnel
- Notion d'enregistrement de données

Fonctions Analogues au réseaux de communication à commutation de paquets

Les sockets mode raw

- analogue au datagramme, mais interface de bas niveau dépendant du protocole

  (ésotérique!)

Création d'un socket

s = **socket** (domaine, type, protocole)

domaine = domaine Unix ou Internet

type = datagram, stream ou raw

protocole = protocole associé au mode

Cet appel crée une structure socket (c-a-d une boîte à message) et renvoi un numéro de descripteur : **s** est un identificateur local (comme un fileid)

Le choix du domaine est implicitement associé à un protocole particulier :

- protocole = 0 ➪ (datagram : UDP) (stream : TCP/IP)
- autre protocole peut être spécifié; protocoles disponibles dans <netinet/in.h>!

Exemples : #include<sys/socket.h>

- mode datagram : s = socket (AF_UNIX, SOCK_DGRAM, 0)
- mode stream : s = socket (AF_INET, SOCK_STREAM, 0)

Attachement - déclaration de nom

Pour être connue de ses interlocuteurs, une boîte aux lettres doit porter un nom.

- **bind**(s, nom, longueur)
  - s = numéro de descripteur de socket
  - nom = pointeur sur un nom dépendant du domaine de création
  - longueur = nombre de caractères du nom (de la structure correspondant à l'adresse

Un socket est créé sans nom ; **bind** associe un nom connu des processus (clients)

Un serveur doit systématiquement associer des noms à ses boîtes à messages

- Domaine UNIX:
  - Un nom de socket est un chemin d'accès (au sens SGF)
- Domaine INTERNET:
  - Un nom de socket correspond à une adresse internet et à un numéro de port

Exemple de noms dans différents domaines de communication

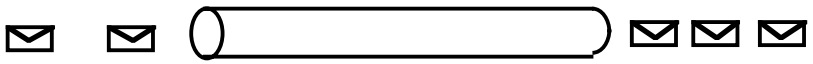
Domaine local	**Domaine distant**

Déclaration d'adresse
#include <sys/un.h>
struct sockaddr_un sun;
sun.sun_family = AF_UNIX;
strcpy(sun.sun_path,"/dev/toto");
bind(s, &sun, strlen("/dev/toto")+2);
 	
Etablissement d'une connexion
bind(s, &sin, sizeof(sin));
	Déclaration d'adresse
	#include <netinet/in.h>
	struct sockaddr_in sin;
	....sin.sin_family = ... host->h_addrtype
	➪ **cf. gethostbyname**
	....sin.sin_port = numéro_de_port
	
	Etablissement d'une connexion
	bind(s, &sin, sizeof(sin));

L'établissement de connexion permet de définir un canal de communication asymétrique entre un processus CLIENT (socket non attaché) à un processus SERVEUR (nom de socket attaché i.e. déclaré)

Logique du fonctionnement Client-Serveur (TCP)

CLIENT	**SERVEUR**
Création d'un socket pour le client CS = socket (...) /********** Le socket CS du client sert à la connexion et à l'émission. Une structure serveur décrit le socket serveur (son adresse). **********/ *Connexion Client ➪ Serveur* connect (CS,@_serveur...)	*Création d'un socket serveur* S = **socket** (...) *Déclaration du nom* **bind** (S, @_serveur...) *Attente de nbmax connexions* **listen** (S, nbmax) *Acceptation de connexion* **accept** (S, from...)

Send (message)		**Recv** (message)

Commentaires sur l'établissement d'une connexion

Primitive connect :

- Domaine UNIX: struct sockaddr_un server;
  - connect (CS, &server, strlen(server.sun_path)+2)
- Domaine INTERNET: struct sockaddr_in server;
  - connect (CS, &server, sizeof(server));

Primitive listen :

- spécifie le nombre maximum de demande de connexion (taille file d'attente)
- bufferisation de requêtes <asynchrones> + évitement de file infinie

Primitive accept :

- effectue l'acquisition des demandes de connexion avec identification du demandeur
- réservation d'espace pour le socket client et création d'un socket (≈ tube)
- si le nom du client n'offre pas d'intérêt le second champ peut être 0
- un processus ne peut pas accepter des connexion uniquement d'un client désigné
- acceptation puis fermeture
- fermeture d'un socket : close (s).

Transfert de données

Les liaisons connectées (stream):

primitives read et write	**primitives send et recv**
write(s, buffer, sizeof(buffer)) **read** (s, buffer, sizeof(buffer))	**send**(s, buffer, sizeof(buffer), flags) **recv**(s, buffer, sizeof(buffer), flags)

Les flags sont utiles dans certains cas très particuliers: consultation, sans lecture par exemple.

Les liaisons non connectées (datagram):

primitive sendto	primitive recvfrom
sendto(s, buf, buflen, flags, &to, tolen)	**recvfrom**(s, buf, buflen, flags, &from, fromlen)

s = numéro de descripteur de socket
buf, buflen = buffer et longueur correspondante
flags = mode de fonctionnement spécial
to, tolen = @ destinataire et longueur from, fromlen = @ émetteur et longueur

REALISATION DE SERVEURS

Librairie réseau

Description d'un socket INTERNET

- Recherche de l'hôte sur lequel tourne le serveur : **gethostname**(*nom, longueur).
- Autres : **sethostname**(...) par root au boot, **gethostid**()

```
struct  hostent {
  char *h_name;         /* nom du site */
  char **h_aliases;     /* liste d'alias */
  int h_addrtype;       /* type adresse */
  int h_length;         /* long. adresse */
  char **h_addr         /* adresse*/
}
```

```
struct sockaddr_in {
short   sin_family;           /*AF_INET*/
u_short sin_port;             /*n° de port*/
struct  in_addr sin_addr;     /* @ */
char    sin_zero[8];
}
```

- Recherche de l'adresse internet d'un hôte : **gethostbyname**.

  nom d'hôte ➪ gethostbyname recherche /etc/hosts ➪ structure hostent

- Définition d'un port serveur

  #define PORT_SERVEUR 8000 ➪ Attention aux numéros réservés par le système!

Librairie réseau (suite)

Accès aux adresses INTERNET :

- recherche de l'hôte local sur lequel tourne le processus : **gethostent**.

  Cette fonction est similaire à **gethostbyname** qui est utilisé lorsque un client recherche l'adresse (hostent) du site distant sur lequel tourne un serveur.

- recherche de l'adresse internet d'un hôte : **gethostbyaddr**.

  adresse socket INET ➪ gethostbyaddr ➪ structure hostent

Exemple :

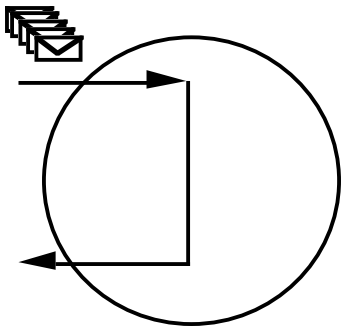
recvfrom(srv_desc,texte, sizeof(texte),0,&C_adr,&lc_a)➪ @ socket client

H_client = gethostbyaddr(&C_adr.sin_addr, sizeof(C_adr.sin_addr), sin_addr.sin_family)

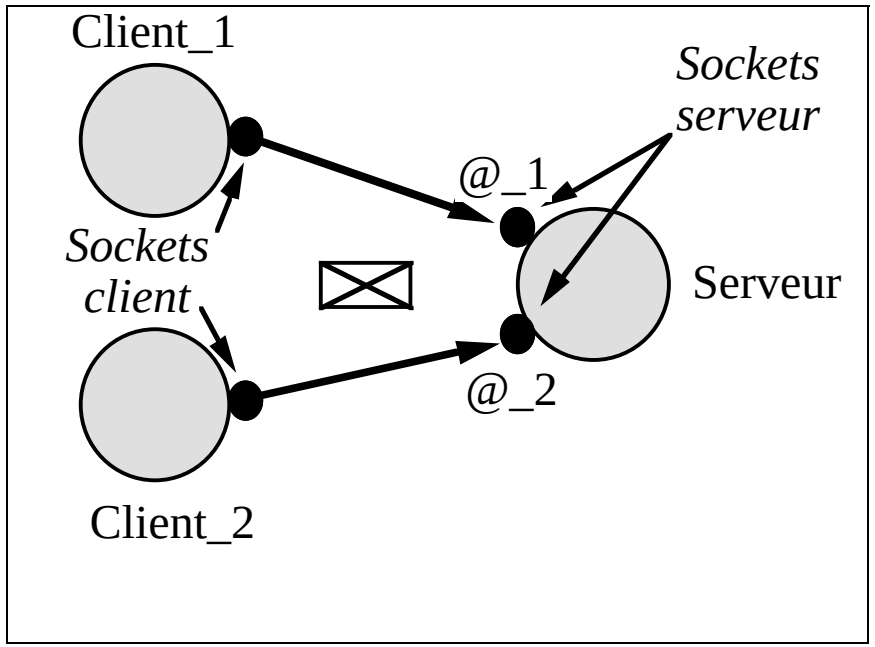
Accès aux noms de réseaux :

getnetbyname(name),
getnetbyaddr(net, type),
getnetent
	struct netent {
	char *n_name; /* nom du réseau */
	char **n_aliases; /* liste d'alias */
	int n_addrtype; /* type adresse */
	int n_net; /* long. adresse */
	};

Client – Serveur (simple) : datagram + domaine INET

Client.Call (serveur, host)	*modèle*	*Serveur*
struct sockaddr_in C_adr, S_adr; char buffer[LNGMAX]; struct hostent *S_host; char host[LNGHOST]; /* recherche @IP hôte serveur S_host= gethostbyname(host); /* initialisation @ serveur S_adr.sin_addr:=S_host->h_addr S_adr.sin_port := PORT_SERVER /* création socket Client S_desc=socket (AF_INET,SOCK_DGRAM, 0); /* emission buffer nbeff=sendto (S_desc, buffer, LG,0,&S_adr,sizeof(S_adr));		/* recherche hôte serveur gethostname(host,LNGHOST) /* recherche @IP hôte serveur S_host= gethostbyname(host); /* initialisation @ serveur S_adr.sin_addr :=S_host->h_addr S_adr.sin_port := PORT_SERVER /* création socket serveur S_desc = socket (AF_INET,SOCK_DGRAM, 0); /* attachement de nom bind(S_desc,&S_adr, lg_name) /* réception buffer+ @client LG = recvfrom (S_desc,LG, buffer, flags, &C_adr, fromlen)

Serveur par polling : Multiplexage d'entrées



• **SELECT ⇨ test de descripteur socket**

```
struct timeval timeout; int nfds,nds;
    fd_set readfds, writefds, exceptfds;
    for (;;) {
    ...FD_SET(e_d, &readfds);
    ...FD_ZERO(&writefds);
    ...FD_ZERO(&exceptfds);
    ...nds = getdtablesize();
    ...nfds = select(nds,&readfds, &writefds,
                          &exceptfds ,&timeout);
    if (nfds>0) { /* lecture - traitement */ } }
```

- Les macros FD_SET,etc. , permettent de définir des **masques** à partir des descripteurs
- La fonction *getdtablesize()* renvoie le nb maximum de descripteurs du processus

SELECT teste s'il existe une entrée (sortie) sur le descripteur correspondant

➥ la lecture sur le socket et effectuée (*recvfrom, read, recv*) si la valeur de retour >

```
struct timeval {
   long    tv_sec;   /* seconds */
   long    tv_usec; /* and microseconds */ };
```

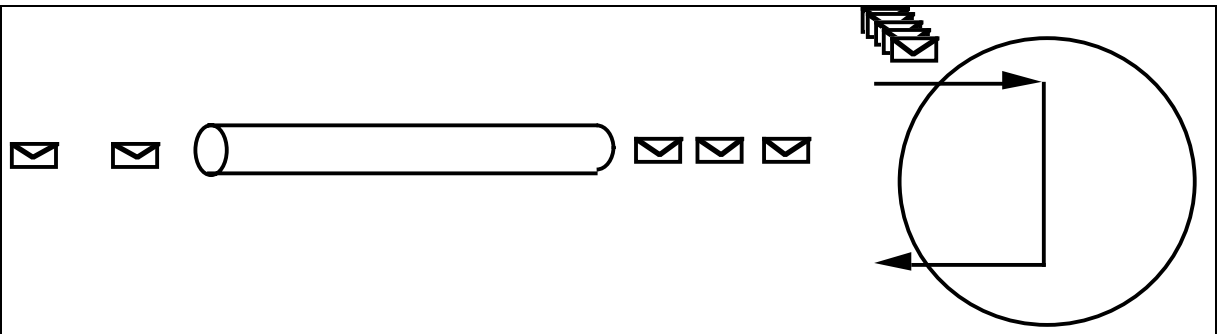
Notion de timeout :

- **valeur 0 ⇨ attente bloquante**
- **valeur ≠ 0 ⇨ test de descripteur**

Modèle Client/Serveur (Généralités)

Modèle le plus fréquemment utilisé dans la construction d'applications réparties

- fonctionnement généralement asymétrique (exemples : login, ftp):
- fonctionnement symétrique (exemple : telnet)
  - ➢ Ce modèle s'appuie sur des sockets INET en mode STREAM (connecté)
  - ➢ Déclaration d'une adresse sockaddr_in associée au serveur, et connue des clients!

1) Client : - établissement connexion 2) Serveur : - attente de connexions - acceptation de connexions	

Pb: connaissance globale des serveurs? hôte de résidence? numéro de port?

La notion de service

- Rendre globale la connaissance des serveurs ➪ Service = Abstraction des serveurs

- Exemple dans UNIX : /etc/services

- Fonctions:

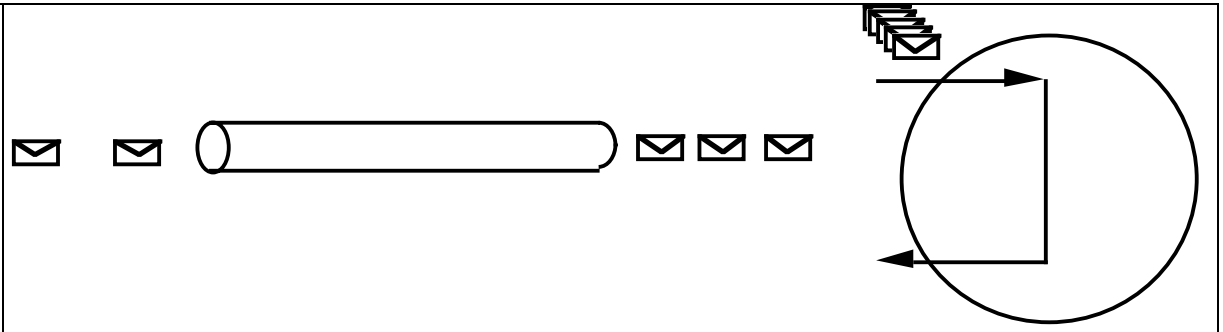
getservent, getservbyport(name, proto), getservbyname(port, proto	struct servent { char *s_name; /* nom de service */ char **s_aliases; /* liste d'alias */ int s_port; /* numéro de port */ char *s_proto; /* protocole utilisé */ };

Exemple Client/Serveur : Client dans les deux modes

Modèle datagram :	**CLIENT**	**Modèle stream :**
S_d=socket(AF_INET, SOCK_DGRAM, 0)		S_d=socket(AF_INET, SOCK_STREAM, 0)
	Connexion au serveur	connect(S_d, &srv_adr, sizeof(srv_adr))
----------		----------
nbeff=sendto(sock_desc,texte, LG,0,&srv_adr, sizeof(srv_adr))	**Emission**	>nbeff=send(S_d,texte, LG,0)

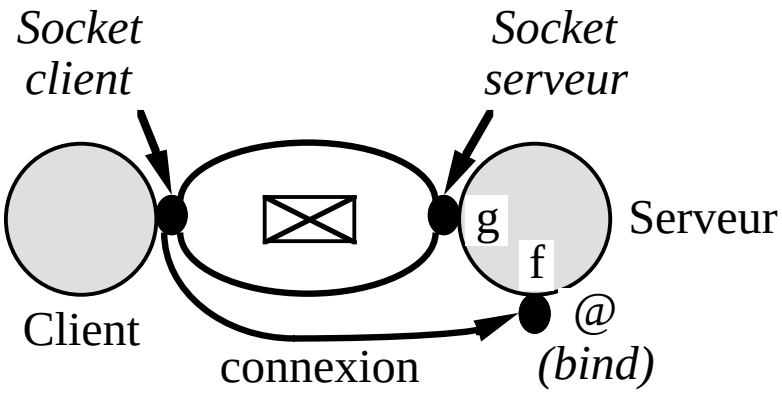
L'action de demande de connection correspond à l'émission d'un premier message qui permet au serveur d'identifier le client par :

- Son adresse IP
- Son Port



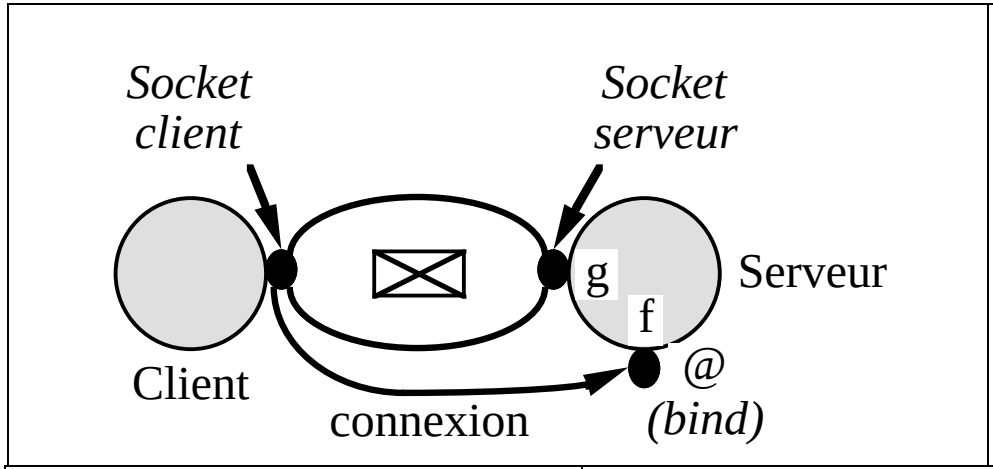
Exemple Client/Serveur : Serveur dans les deux modes

Modèle datagram:	**SERVEUR**	**Modèle stream :**
S_d=socket(AF_INET, SOCK_DGRAM, 0)	**Attente & Acceptation connexion sur nouveau socket**	S_d =socket(AF_INET, SOCK_STREAM, 0) listen(S_d,5); **new_sock**=accept(S_d,&cli_adr, &lc_a)
---------- <l_t=recvfrom(sock_desc,texte, sizeof(texte),0,&cli_adr,&lc_a)	réception	---------- l_t=recv(**new_sock**,texte,sizeof(texte),0)



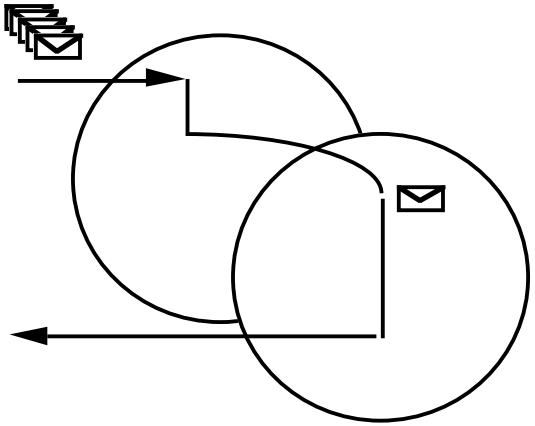
Service distant - Client/Serveur mode connecté

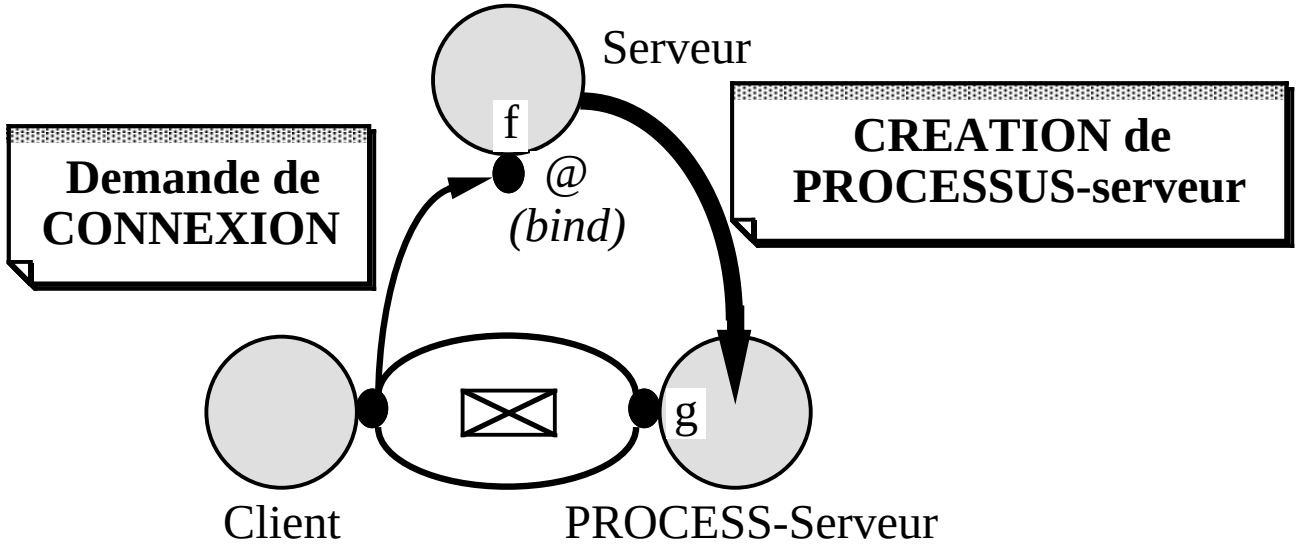
Schéma client-serveur mode connecté




```
for (;;) {
     int g, len=sizeof(from)
     g=accept(f, &from, &len);
     if (g<0) perror("accept");
     if (fork()==0) {
                    close(f);
                    PROCESS(g,&from);}
close(g) }
```

Serveur à la demande





Services

- Les services sont référencés dans ***/etc/services*** par :
  ➥ leur **nom**, leur **numéro de port**, le **protocole** associé

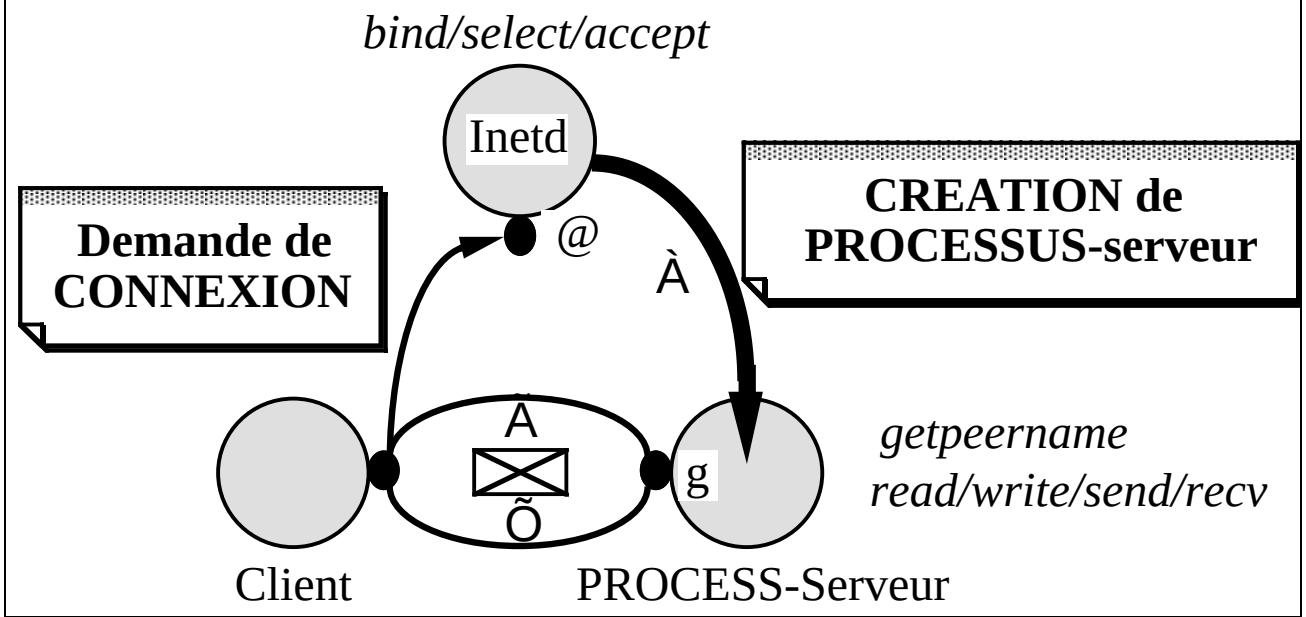
- Un serveur *deamon* particulier permet de lancer un serveur : **inetd**.
  - ➢ fichier de configuration : ***/etc/inetd.conf***
    ➥ le **nom_service**, le **mode**, le **protocole** associé, le flag **wait/nowait**, l'**utilisateur**, le **pathname** du binaire
  - ➢ fonctionnement :
    1) inetd se met en écoute (***select***) sur le numéro de port de tous les services mentionnés dans inetd.conf au boot (inetd ré-activé par kill -1)
    2) inetd accepte (***accept***)les connections sur le port du service correspondant
    3) inetd génère (***fork***, ***dup***, ***exec***) le processus serveur correspondant .

Remarques :

- ➢ 1) wait/nowait : nowait + STREAM ⇨ remise en attente de connexion wait + DGRAM ⇨ attente traitement du message (single thread)
- ➢ 2) stdin et sdtout sont fermés par inetd lors de la création du serveur.

Mise en œuvre de services

Rôle de inetd




```
desc=0; lc_a=sizeof(client);
if (getpeername(desc,
                (struct sockaddr *)&client,&lc_a )<0){
               perror("S-> getpeername");
               bye(); }
le_t = recv(desc,e_t,sizeof(e_t),0)
```

int **getpeername**(s, name, namelen)

```
int s;
struct sockaddr *name;
int *namelen;
```

Renvoie le nom d'un client connecté sur un socket s (n° descripteur)

Un serveur utilise s=0 pour read et s=1 pour write

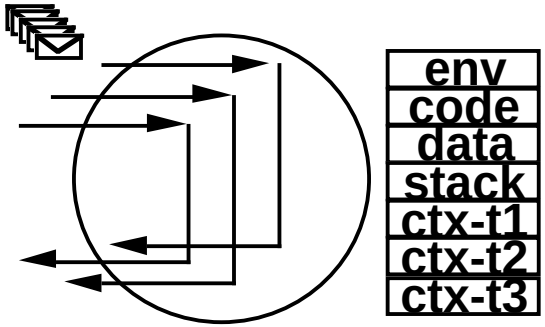
- ***client* est l'adresse internet du client**
- **le descripteur *desc* vaut la valeur 0**
- **la réception de message se fait sur 0**
- **l'émission de message se fait sur 1**

Serveur multithreaded

Principe : plusieurs flots d'exécution (threads) sont actifs au sein d'un même processus.

Remarque :

- les données globales sont potentiellement partagées (partageables)
- les threads sont donc concurrents (attention aux sections critiques)



- **pthread_create :** création de thread
- **pthread_yield :** suspension de thread
- **pthread_join :** attente du résultat final du thread
- **pthread_exit :** terminaison et fin d'activité

Autres remarques :

- Attention, l'ordonnancement des threads (e.g. dans POSIX) est indépendant de la volonté des programmeurs.
- L'ordonnancement peut se faire sur quantum de temps, par priorité, par échéance temporelle…

"Advanced topics" (un aperçu!)

Données "hors-bande" (Out-of-Band data)

- Transmission de données entre sockets STREAM (connectés) indépendante de l'échange normal de données ➪ transmission de message urgent! (+ signal SIGURG)

Sockets non bloquant

- Un socket peut être marqué comme non-bloquant "fcntl (s, F_SETFL, FNDELAY)"

Sockets pilotés par interruption

- Ce type d'utilisation permet de recevoir la notification de réception sur interruption. Un handler d'interruption est associé à SIGIO (signal) pour un processus ou pour un groupe.

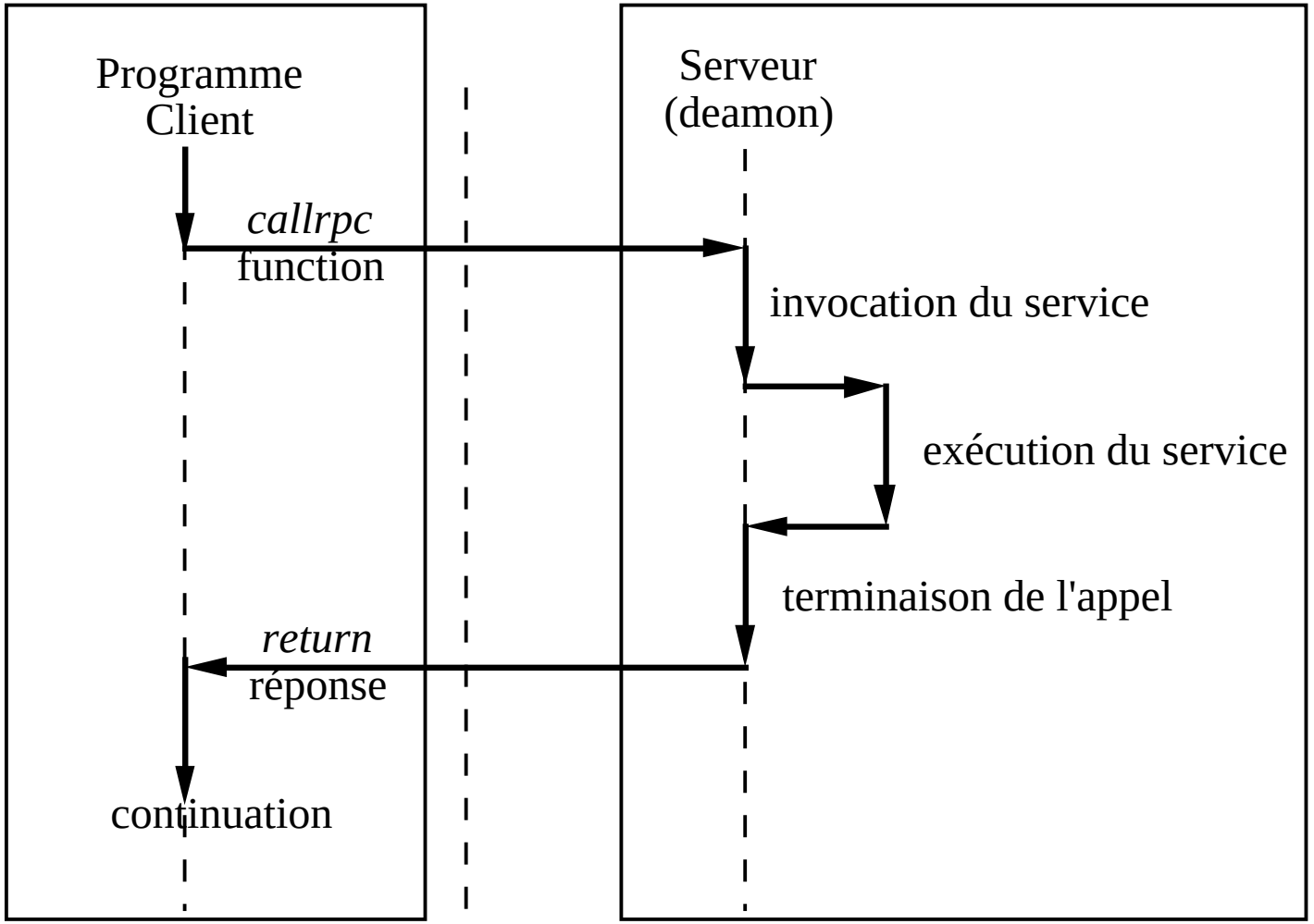
Diffusion

- Les sockets en mode DGRAM permettent d'effectuer de la diffusion sur des sockets préalablement marqués (setsockopt) comme acceptant des messages en diffusion (exemple d'usage: recherche de correspondant inconnu a priori). Une adresse en diffusion (INADDR_ANY) doit être déclarée (bind).

INTRODUCTION A LA COMMUNICATION

PAR APPEL DE PROCEDURE A DISTANCE

Notion de RPC



Le compilateur **rpcgen** permet de masquer l'usage des appels systèmes tels que callrpc pour la mise en œuvre de "remote" procédures

Généralités

- Le *Remote Procedure Call* est basé sur TCP/IP et utilise un langage particulier proche du langage C ainsi que :

  - XDR : *External Data Representation language*

  - mécanisme de communication basé sur les sockets

  - un mécanisme d'authentification client-serveur

  - un serveur de gestion des ports associés aux services (*portmapper*)

- Beaucoup de services distribués ont été bâtis en utilisant les RPC; un exemple NFS.
- Une procédure "remote" est associée à un "numéro_de_programme";
- En ce qui concerne les programmes utilisateur la plage de numéros réservée à cet usage est : 0x20000000-0x3fffffff.
- Les "numéros_de_programme" 0x0-0x1fffffff sont réservés aux applications système.

L'interface RPC

Elle peut être perçue selon 3 niveaux :

① - niveau RPC où l'on utilise seulement des primitive cataloguées, *rnusers()* par ex.

② - niveau RPC "propre"qui permet à un utilisateur de réaliser des procédures "remotes"

③ - niveau "bas" qui permet à un utilisateur d'accéder au fonctionnement de base.

Exemple de programme qui écrit un message passé en argument dans un fichier du répertoire courant (fichier message)

Tequila% machine sur laquelle est lancée un serveur de réception de message

Sunrise% machine sur laquelle s'exécute de client.

- il invoque le serveur par une commande
- il passe à cette commande le « message » en paramètre

Réalisation de l'exemple :

printmsg.c :

cc printmsg.c -o printmsg

printmsg "Salut petit" ---> cat message

RPC-Client-Serveur:

① **msg.x :** *Description de la procédure "remote" en langage RPC*

Les procédures "remotes" sont définies dans des programmes; dans notre cas le programme ne contient qu'une seule procédure.

② **msg_proc.c :** *Description de la procédure remote avec les différences suivantes*:

a) pointeur vers tout argument

b) retour d'un pointeur sur tout résultat.

c) "_1" a été ajoute' au nom;

le nom mentionné *PRINTMESSAGE_1* dans msg.x ci-dessus est converti en minuscules par *rpcgen* et suivi de "_NuméroDeVersion".

③ **rprintmsg.c :** *Description du Client*

a) clnt_create : création d'un "handle" utilise' par les stubs.

b) même appel à printmessage_1 + handle en premier paramètre.

④ **Etapes de génération:**

a) rpcgen msg.x :

- création du header file "msg.h" (#defines)
- création de stubs (msg_clnt.c et msg_svc.c)

b) cc rprintmsg.c msg_clnt.c -o rprintmsg

c) cc msg_proc msg_svc.c -o msg_server

⑤ **Exécution:**

tequila% msg_server &

sunrise% rprintmsg tequila "sunrise"

```
/* msg.x :
protocole d'impression a distance
 */
program MESSAGEPROG {
version MESSAGEVERS {
int PRINTMESSAGE(string) = 1;
} = 1;
} = 99;

======================
/* Please do not edit this file.
 * It was generated using
 * rpcgen.
*/

#include <rpc/types.h>

#define MESSAGEPROG ((u_long)99)
#define MESSAGEVERS ((u_long)1)
#define PRINTMESSAGE ((u_long)1)

extern int *printmessage_1();
```

```
/* Please do not edit this file.
 * It was generated using rpcgen.
 */
#include <rpc/rpc.h>
#include "msg.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

int *
printmessage_1(argp, clnt)
        char **argp;
        CLIENT *clnt;
{
        static int res;

        bzero((char *)&res, sizeof(res));
        if (clnt_call(clnt, PRINTMESSAGE,
                        xdr_wrapstring, argp, xdr_int, &res, TIMEOUT) !=
                        RPC_SUCCESS) {
                return (NULL);
        }
        return (&res);
}
```

```
/* * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include <stdio.h> <rpc/rpc.h> "msg.h"

static void messageprog_1();

main()
{
        register SVCXPRT *transp;

        if (!svc_register(transp, MESSAGEPROG, MESSAGEVERS,
            messageprog_1, IPPROTO_TCP)) {
                fprintf(stderr, "unable to register (MESSAGEPROG
                                , MESSAGEVERS, tcp).");
                exit(1);
        }

        svc_run();
        fprintf(stderr, "svc_run returned");
        exit(1);
        /* NOTREACHED */
}

messageprog_1(rqstp, transp)
        struct svc_req *rqstp;
        register SVCXPRT *transp;
{
        union {
                char *printmessage_1_arg;
        } argument;
        char *result;
```

```
static void bool_t (*xdr_argument)(), (*xdr_result)();
        char *(*local)();

switch (rqstp->rq_proc) {
        case NULLPROC:
        (void) svc_sendreply(transp, xdr_void, (char *)NULL); return;

        case PRINTMESSAGE:
                xdr_argument = xdr_wrapstring;
                xdr_result = xdr_int;
                local = (char *(*)()) printmessage_1;
                break;

        default:
                svcerr_noproc(transp);
                return;
        }
        bzero((char *)&argument, sizeof(argument));
        if (!svc_getargs(transp, xdr_argument, &argument)) {
                svcerr_decode(transp);
                return;
        }
        result = (*local)(&argument, rqstp);
        if (result != NULL && !svc_sendreply(transp, xdr_result, result))
{
                svcerr_systemerr(transp);
        }
        if (!svc_freeargs(transp, xdr_argument, &argument)) {
                fprintf(stderr, "unable to free arguments");
                exit(1);
        }
        return;
```

Conclusion

La programmation distribuée peut s'effectuer à deux niveaux :

- Utilisation des mécanismes de base des exécutifs (comme POSIX), c'est-à-dire le passage direct de message, la gestion de service, et la gestion des problèmes de synchronisation des accès concurrents
- En utilisant les appels de procédure à distance ou tout ce qui précède est masqué.

L'objectif d'un intergiciel (CORBA et autres) est de fournir des constructions « toutes faites » pour réaliser des applications réparties :

- Client + **Stub client**
- Serveur + **Skeleton serveur**

Des services de désignation/localisation sont aussi fournis. Attention à la synchronisation, cependant !