Communications dans les clusters

Olivier GLÜCK
Université LYON 1/UFR d'Informatique
ENS LIP projet INRIA RESO
Olivier.Gluck@ens-lyon.fr
http://www710.univ-lyon1.fr/~ogluck

Copyright

Plan

Bibliographie

- « La communication sous Unix », 2ième édition, Jean-Marie Rifflet, Ediscience international, ISBN 2-84074-106-7
- « Bulding Clustered Linux Systems », R. W. Lucke, Prentice Hall PTR, ISBN 0-13-144853-6
- « High Performance Cluster Computing », Vol. 1&2, R. Buyya, Prentice Hall PTR, ISBN 0-13-013784-7
- « Beowulf Cluster Computing with Linux », 2nd edition, Gropp, M.I.T. Press, ISBN 0-262-69292-9
- Internet…
  - http://www.cs.mu.oz.au/678/
  - http://www.cs.wmich.edu/~gupta/teaching/cs526/sp03/tutorials/mpich.html
  - http://www.buyya.com/
  - http://www.cs.wmich.edu/gupta/teaching/cs626/w98/mpich_doc.html

Introduction

Modèle client/serveur
Communications inter-processus
Rappel : API socket et RPC

Les applications réseau (1)

- Applications = la raison d'être des réseaux infos
- Profusion d'applications depuis 30 ans grâce à l'expansion d'Internet
  - années 1980/1990 : les applications "textuelles"
    - messagerie électronique, accès à des terminaux distants, transfert de fichiers, groupe de discussion (forum, newsgroup), dialogue interactif en ligne (chat), la navigation Web
  - plus récemment :
    - les applications multimédias : vidéo à la demande (streaming), visioconférences, radio et téléphonie sur Internet
    - la messagerie instantanée (ICQ, MSN Messenger)
    - les applications Peer-to-Peer (MP3, …)

Les applications réseau (2)

- L'application est généralement répartie (ou distribuée) sur plusieurs systèmes
- Exemples :
  - L'application Web est constituée de deux logiciels communiquants : le navigateur client qui effectue une requête pour disposer d'un document présent sur le serveur Web
  - L'application telnet : un terminal virtuel sur le client, un serveur telnet distant qui exécute les commandes
  - La visioconférence : autant de clients que de participants
- --> Nécessité de disposer d'un protocole de communication applicatif !

Terminologie des applications réseau

- Processus :
  - une entité communicante
  - un programme qui s'exécute sur un hôte d'extrémité
- Communications inter-processus locales :
  - communications entre des processus qui s'exécutent sur un même hôte
  - communications régies par le système d'exploitation (tubes UNIX, mémoire partagée, ...)
- Communications inter-processus distantes :
  - les processus s'échangent des **messages** à travers le réseau selon un **protocole** de la couche applications
  - nécessite une infrastructure de transport sous-jacente

Protocoles de la couche Applications

- Le protocole applicatif définit :
  - le format des messages échangés entre les processus émetteur et récepteur
  - les types de messages : requête, réponse, …
  - l'ordre d'envoi des messages
- Exemples de protocoles applicatifs :
  - HTTP pour le Web, POP/IMAP/SMTP pour le courrier électronique, SNMP pour l'administration de réseau, …
- Ne pas confondre le protocole et l'application !
  - Application Web : un format de documents (HTML), un navigateur Web, un serveur Web à qui on demande un document, un protocole (HTTP)

Le modèle Client / Serveur

- Idée : l'application est répartie sur différents sites pour optimiser le traitement, le stockage...
- Le client
  - effectue une demande de service auprès du serveur (**requête**)
  - initie le contact (parle en premier), ouvre la session
- Le serveur
  - est la partie de l'application qui offre un service
  - est à l'écoute des requêtes clientes
  - répond au service demandé par le client (**réponse**)

Le modèle Client / Serveur

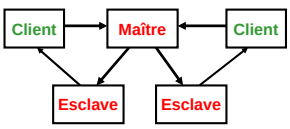
- Le client et le serveur ne sont pas identiques, ils forment un système coopératif
  - les parties client et serveur de l'application peuvent s'exécuter sur des systèmes différents
  - une même machine peut implanter les côtés client ET serveur de l'application
  - un serveur peut répondre à plusieurs clients simultanément

Des clients et des serveurs...

Un client, un serveur :

Requête/Réponse

Plusieurs clients, un serveur :

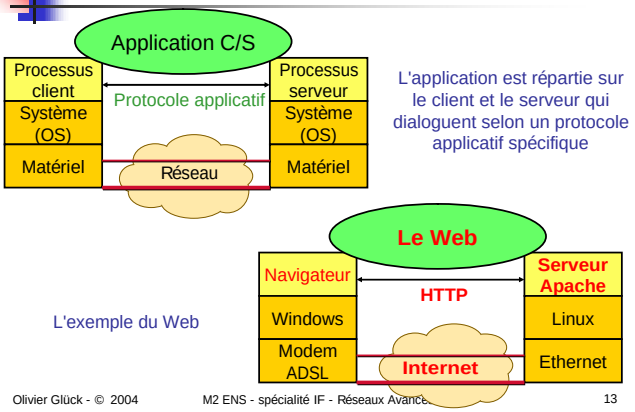


Le serveur traite plusieurs requêtes simultanées

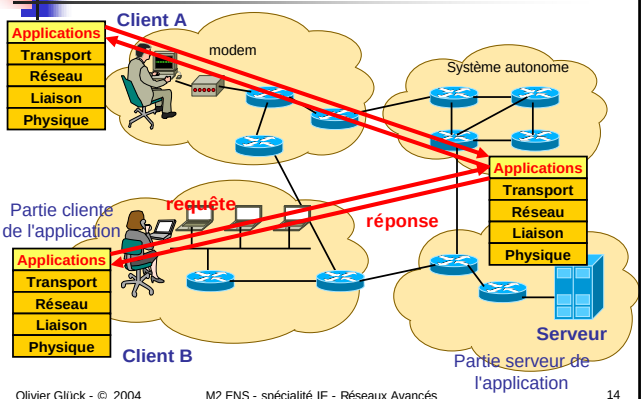
Un client, plusieurs serveurs :

Le serveur contacté peut faire appel à un service sur un autre serveur (ex. SGBD)

Le modèle Client / Serveur



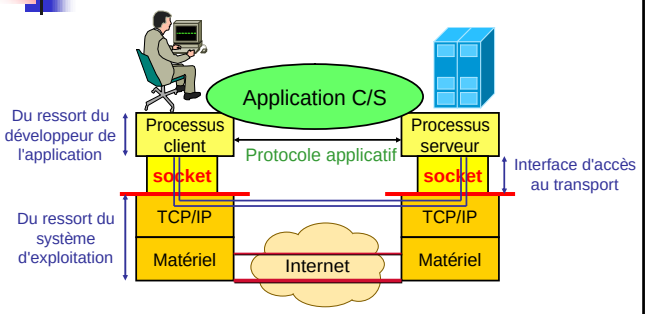
Le modèle Client / Serveur



Interface de programmation réseau

- Il faut une interface entre l'application réseau et la couche transport
  - le transport n'est qu'un tuyau (TCP ou UDP dans Internet)
  - l'API (Application Programming Interface) n'est que le moyen d'y accéder (interface de programmation)
- Les principales APIs de l'Internet
  - les sockets
    - apparus dans UNIX BSD 4.2
    - devenus le standard de fait
  - les RPC : Remote Procedure Call - appel de procédures distantes

Interface de programmation réseau



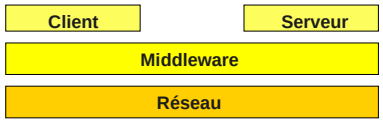
Une socket : interface locale à l'hôte, créée par l'application, contrôlée par l'OS
Porte de communication entre le processus client et le processus serveur

Application C/S - récapitulatif

- Une application Client/Serveur, c'est
  - **une partie cliente** qui exécute des requêtes vers un serveur
  - **une partie serveur** qui traite les requêtes clientes et y répond
  - **un protocole applicatif** qui définit les échanges entre un client et un serveur
  - **un accès via une API** (interface de programmation) à la couche de transport des messages
- Bien souvent les parties cliente et serveur ne sont pas écrites par les mêmes programmeurs (Navigateur Netscape/Serveur apache) --> rôle important des RFCs qui spécifient le protocole !

Le Middleware

- Grossièrement : la gestion du protocole applicatif+l'API d'accès à la couche transport+des services complémentaires
- C'est un ensemble de services logiciels construits au dessus d'un protocole de transport afin de permettre l'échange de requête/réponse entre le client et le serveur de manière transparente



Le Middleware

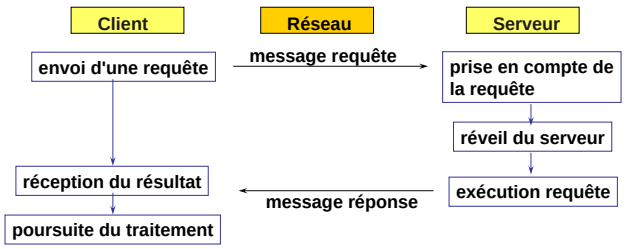
- Complément de services du réseau permettant la réalisation du dialogue client/serveur :
  - prend en compte les requêtes de l'application cliente
  - les transmet de manière transparente à travers le réseau jusqu'au serveur
  - prend en compte les données résultat du serveur et les transmet vers l'application cliente
- L'objectif essentiel du middleware est d'offrir aux applications une interface unifiée permettant l'accès à l'ensemble des services disponibles sur le réseau : l'API

Fonctions d'un Middleware

- Procédures d'établissement/fermeture de connexion
- Exécution des requêtes, récupération des résultats
- Initiation des processus sur différents sites
- Services de répertoire
- Accès aux données à distance
- Gestion d'accès concurrents
- Sécurité et intégrité (authentification, cryptage, …)
- Monitoring (compteurs, …)
- Terminaison de processus
- Mise en cache des résultats, des requêtes

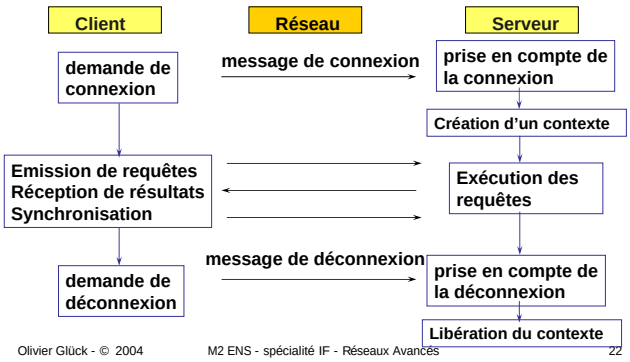
Les modes de communication

- Communication en mode non connecté



Les modes de communication

- Communication en mode connecté

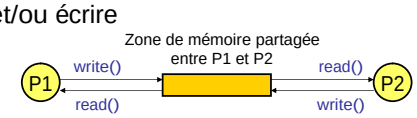


Les schémas de communication

- Dès lors qu'une application est répartie, elle se décompose en plusieurs processus qui doivent communiquer (échanges de données)
- Deux grands types de schéma de communication
  - communication par mémoire partagée (ou fichier)
  - communication par passage de messages
- On retrouve ces deux schémas de communication
  - dans des **communications locales** : entre processus s'exécutant sur le même hôte
  - dans des **communications distantes** : entre processus s'exécutant sur des hôtes distants

Communication par mémoire partagée

- Les processus se partagent une zone de mémoire commune dans laquelle ils peuvent lire et/ou écrire



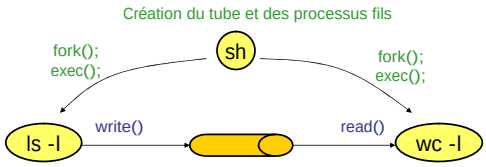
- Intérêt : communications transparentes, limitation des copies mémoire
- Problème : gestion de l'accès à une ressource partagée
  - problème si deux écritures simultanées (ordre d'ordonnancement, atomicité des opérations)
  - les processus P1 et P2 doivent se synchroniser pour accéder au tampon partagé (verrou, sémaphore, …)

Communication par mémoire partagée

- Communications locales
  - les deux processus s'exécutent sur la même machine donc peuvent se partager une partie de leur espace d'adressage
  - exemple : les threads s'exécutent dans le contexte d'un même processus
- Communications distantes
  - la mémoire partagée est physiquement répartie
  - le gestionnaire de mémoire virtuelle permet de regrouper les différents morceaux selon un seul espace d'adressage
  - problème de cohérence mémoire...

Les tubes de communication (pipes)

- Communications locales type mémoire partagée
  - le canal de communication est unidirectionnel (pas de problème de synchronisation)
  - communications entre 2 processus uniquement : l'un écrit dans le tube, l'autre lit
- Exemple : `sh$ ls -l | wc -l`

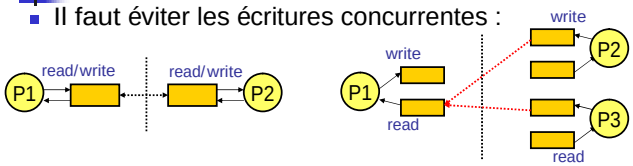


Communication par passage de msg

- Les processus n'ont pas accès à des "variables" communes
- Ils communiquent en s'échangeant des messages
  - au moins deux primitives : send() et recv()
  - des zones de mémoire locales à chaque processus permettent l'envoi et la réception des messages
  - l'émetteur/récepteur doit pouvoir désigner le récepteur/émetteur distant
- Problèmes
  - zones d'émission et réception distinctes ?
  - nombre d'émetteurs/récepteurs dans une zone ?
  - opérations bloquantes/non bloquantes ?

Communication par passage de msg

- Il faut éviter les écritures concurrentes :



- Pour se ramener à des communications point-à-point
  - --> dissocier le tampon d'émission et de réception
  - --> avoir autant de tampons de réception que d'émetteurs potentiels
  - --> il ne reste plus alors au protocole qu'à s'assurer que deux émissions successives (d'un même émetteur) n'écrasent pas des données non encore lues (contrôle de flux)

Opérations bloquantes/non bloquantes

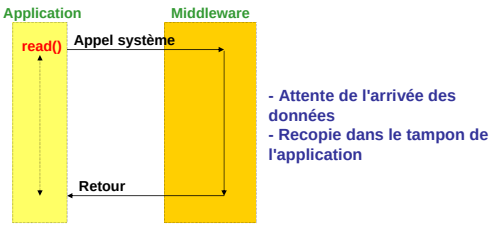
- Quand un appel à une primitive send() ou recv() doit-il se terminer ?
- Plusieurs sémantiques en réception :
  - recv() peut rendre la main
    - aussitôt (recv() non bloquant)
    - quand les données ont été reçues et recopiées depuis le tampon de réception local (le tampon de réception est de nouveau libre)

Opérations bloquantes/non bloquantes

- Plusieurs sémantiques en émission :
  - send() peut rendre la main
    - aussitôt (send() non bloquant)
    - quand les données ont été recopiées dans le tampon d'émission local (les données peuvent être modifiées au niveau de l'application)
    - quand les données ont été recopiées dans le tampon de réception distant (le tampon d'émission local est de nouveau libre)
    - quand le destinataire a consommé les données (le tampon de réception sur le destinataire est de nouveau libre)

Opérations bloquantes

- Le processus se bloque jusqu'à ce que l'opération se termine :



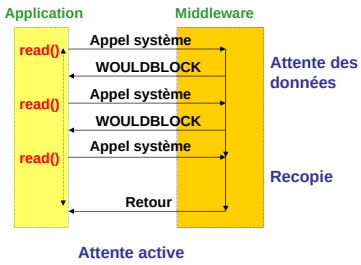
Opérations non bloquantes

- Intérêt :
  - le processus peut faire autre chose en attendant que les données soient émises ou reçues
- Le processus a tout de même besoin d'être informé de la complétion de l'opération (lecture ou écriture)
- Deux possibilités :
  - attente active : appels réguliers à la primitive jusqu'à complétion
  - attente passive : le système informe le processus par un moyen quelconque de la complétion de l'opération (signaux par exemple)

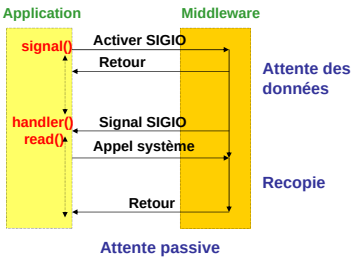
Communication par signaux

- Mécanisme de communications **locales** inter-processus (ou depuis le noyau vers un processus) permettant de notifier un événement
- Principe : interruption logicielle quand l'événement se produit
- Le processus
  - indique les signaux qu'il souhaite capter (provoquant son interruption)
  - met en place un handler (fonction particulière) qui sera exécuté quand l'événement se produira
- Exemple : arrivée de données urgentes sur une socket

Opérations non bloquantes



Opérations non bloquantes



Désignation du destinataire/émetteur

- Pour faire du passage de messages, il est nécessaire de désigner l'autre extrémité de la communication
- Désignation explicite
  - du ou des processus destinataire(s)/émetteurs
- Désignation implicite
  - recevoir un message de n'importe qui
  - émettre un message à n'importe qui (diffusion)
  - une phase d'établissement de connexion désigne les deux entités communicantes

Les sockets - adressage

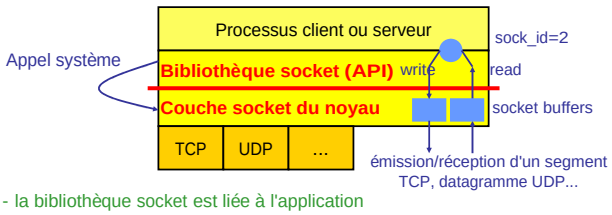
- Deux processus communiquent en émettant et recevant des données via les sockets
- Les sockets sont des portes d'entrées/sorties vers le réseau (la couche transport)
- Une socket est identifiée par une adresse de transport qui permet d'identifier les processus de l'application concernée
- Une adresse de transport = un numéro de port (identifie l'application) + une adresse IP (identifie le serveur ou l'hôte dans le réseau)

Les sockets - adressage

- Le serveur doit utiliser un numéro de port fixe vers lequel les requêtes clientes sont dirigées
- Les ports inférieurs à 1024 sont réservés :
  - "well-known ports"
  - ils permettent d'identifier les serveurs d'applications connues
  - ils sont attribués par l'IANA
- Les clients n'ont pas besoin d'utiliser des well-known ports
  - ils utilisent un port quelconque entre 1024 et 65535 à condition que le triplet <transport/@IP/port> soit unique
  - ils communiquent leur numéro de port au serveur lors de la requête (à l'établissement de la connexion TCP ou dans les datagrammes UDP)

Les sockets en pratique

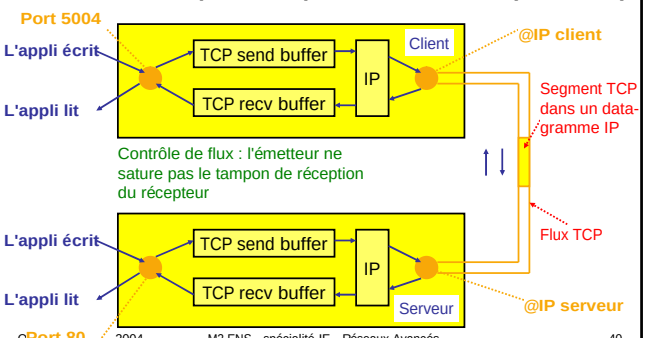
Un descripteur de socket (sock_id) n'est qu'un point d'entrée vers le noyau



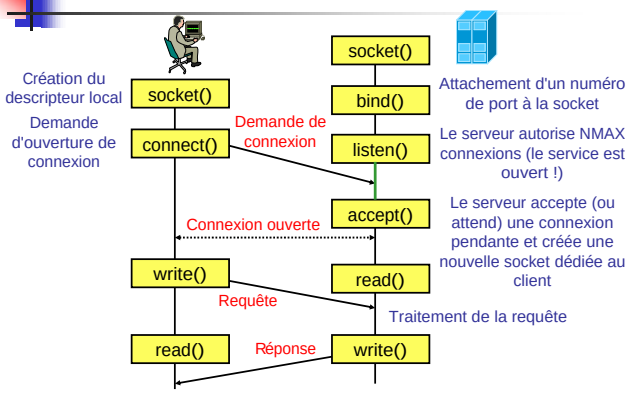
- la bibliothèque socket est liée à l'application
- la couche socket du noyau réalise l'adaptation au protocole de transport utilisé

Rappel - une connexion TCP

- Une connexion = **(@IP_src,port_src,@IP_dest,port_dest)**



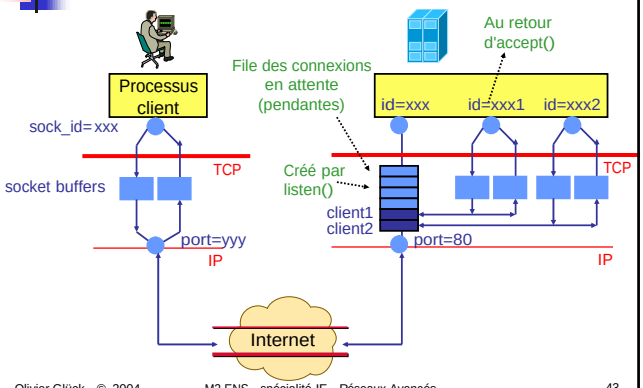
En mode connecté...



En mode connecté...

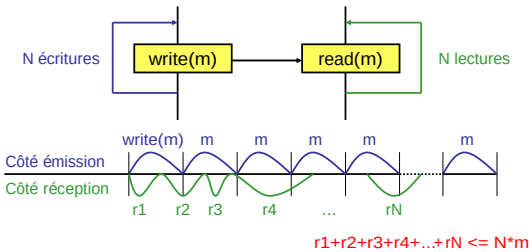
	Paramètres en entrée	Paramètres en sortie
socket()	type, domaine, protocole	sock_id
bind()	sock_id, port	
listen()	sock_id, NMAX	
connect()	sock_id, @sock_dest	
accept()	sock_id	@sock_src, client_sock_id
read()	client_sock_id, @recv_buf, lg	read_lg
write()	client_sock_id, @send_buf, lg	write_lg

En mode connecté...



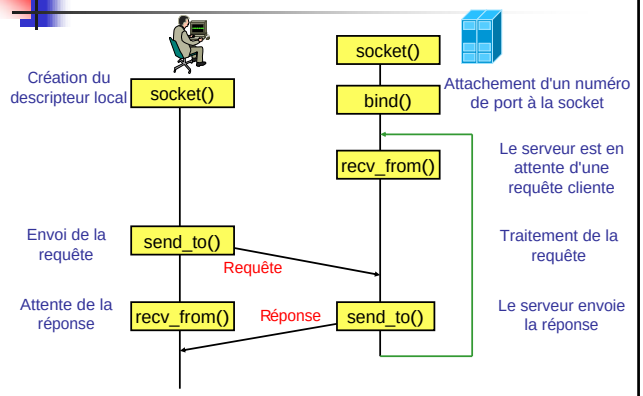
En mode connecté...

- Attention : les émissions/réceptions ne sont pas synchrones
  - read(m) : lecture **d'au plus** m caractères
  - write(m) : écriture de m caractères



r1+r2+r3+r4+...+rN <= N*m

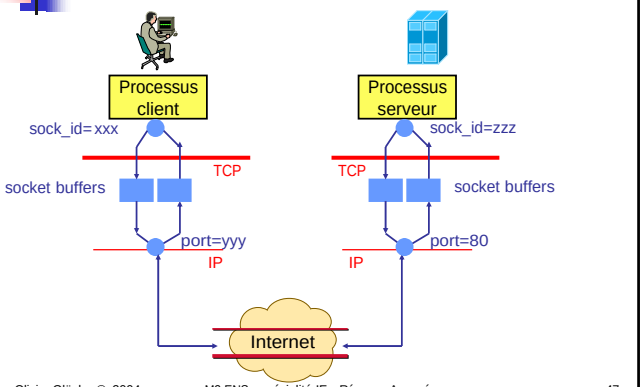
En mode non connecté...



En mode non connecté...

	Paramètres en entrée	Paramètres en sortie
socket()	type, domaine, protocole	sock_id
bind()	sock_id, port	
recv_from()	sock_id, @recv_buf, lg	read_lg, **@sock_src**
send_to()	sock_id, **@sock_dest**, @send_buf, lg	write_lg
Rappel en mode connecté :		
read()	client_sock_id, @recv_buf, lg	read_lg
write()	client_sock_id, @send_buf, lg	write_lg

En mode non connecté...



Opérations bloquantes/non bloquantes

- Par défaut, les primitives `connect()`, `accept()`, `send_to()`, `recv_from()`, `read()`, `write()` sont bloquantes
  - `recv()` sur un tampon vide attendra l'arrivée des données pour rendre la main
  - `send()` sur un tampon plein attendra que les données quitte le tampon pour rendre la main
  - `accept()` ne rend la main qu'une fois une connexion établie (bloque si pas de connexions pendantes)
  - `connect()` ne rend la main qu'une fois la connexion cliente établie (sauf si pas entre `listen()` et `accept()`)

Opérations bloquantes/non bloquantes

- Il est possible de paramètrer la socket lors de sa création pour rendre les opérations non bloquantes
- Comportement d'une émission non bloquante
  - tout ce qui peut être écrit dans le tampon l'est, les caractères restants sont abandonnés (la primitive retourne le nombre de caractères écrits)
  - si aucun caractère ne peut être écrit (tampon plein), retourne -1 avec errno=EWOULDBLOCK (l'application doit réessayer plus tard)
- Comportement d'une lecture non bloquante
  - s'il n'y a rien à lire dans la socket, retourne -1 ... (l'application doit réessayer plus tard)

La scrutation de plusieurs sockets

- La primitive `select()` rend la main quand une de ces conditions se réalise :
  - l'un des événements attendus sur un descripteur de l'un des ensembles se réalise : les descripteurs sur lesquels l'opération est possible sont dans un paramètre de sortie
  - le temps d'attente maximum s'est écoulé
  - le processus a capté un signal (provoque la sortie de `select()`)

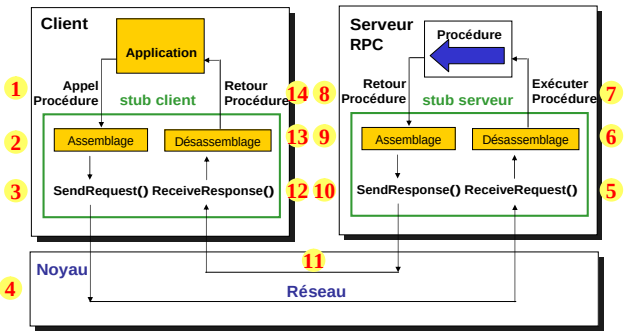
Deux approches de conception

- Un concepteur d'application distribuée peut procéder selon deux approches :
  - conception orientée communication :
    - définition du protocole d'application (format et syntaxe des messages) inter-opérant entre le client et le serveur
    - conception des composants serveur et client, en spécifiant comment ils réagissent aux messages entrants et génèrent les messages sortants
  - conception orientée application :
    - construction d'une application conventionnelle, dans un environnement mono-machine
    - subdivision de l'application en plusieurs modules qui pourront s'exécuter sur différentes machines

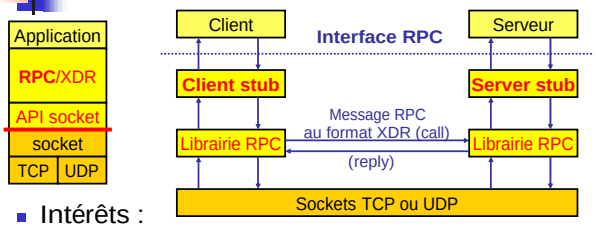
Principe général

- Souvent, quand un client envoie une requête (des paramètres), il est bloqué jusqu'à la réception d'une réponse
- Analogie avec un appel de fonction
  - la fonction ou procédure ne rend la main au programme appelant qu'une fois le traitement (calcul) terminé
- RPC - Remote Procedure Call
  - permettre à un processus de faire exécuter une fonction par un autre processus se trouvant sur une machine distante
  - se traduit par l'envoi d'un message contenant l'identification de la fonction et les paramètres
  - une fois le traitement terminé, un message retourne le résultat de la fonction à l'appelant

Le modèle RPC



L'interface RPC



- Intérêts :
  - l'application n'a pas à manipuler directement les sockets (le transport des données est transparent)
  - l'implémentation des RPC est indépendante de l'OS
- Inconvénient :
  - l'utilisation des RPC est moins performante que l'utilisation directe des sockets (couches supplémentaires)

Restrictions liées aux RPC

- Pas de passage de paramètres par adresse : impossible de passer des pointeurs (ou références)
  - en effet, les espaces d'adressage du client et du serveur sont différents donc aucun sens de passer une adresse
- La procédure distante n'a pas accès aux variables globales du client, aux périphériques d'E/S (affichage d'un message d'erreur !)
- Un appel de procédure obéit à fonctionnement synchrone : une instruction suivant un appel de procédure ne peut pas s'exécuter tant que la procédure appelée n'est pas terminée