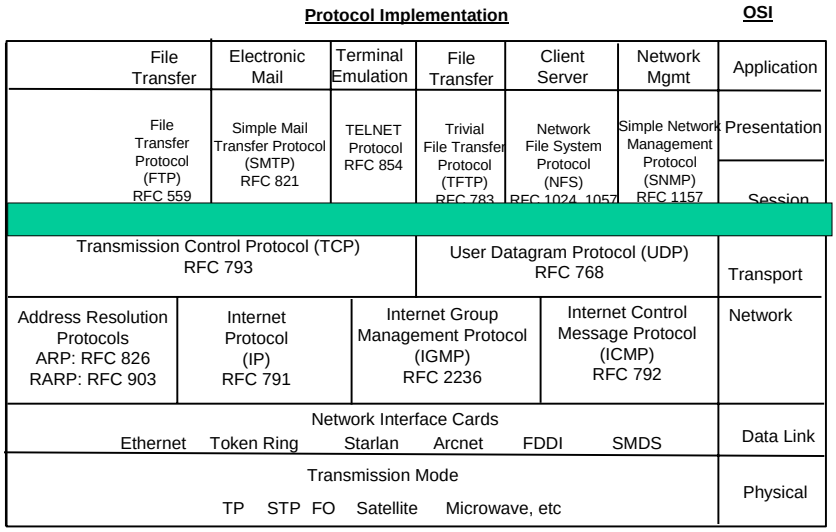
Cours 4: TCP/IP Programmation

Cours 4: Plan

4.1 INTRODUCTION
4.2 LE MODELE
4.3 LA CONCEPTION
4.4 LES SOCKETS
- Introduction
- les primitives
- la structure d'adressage
- les serveurs multi-protocoles
- les serveurs multiservices

4.5 LES RPC

Les couches de protocoles : TCP/IP et le modèle OSI



3

Introduction

- L'architecture Client/Serveur est l'aboutissement d'un ensemble d'évolutions technologiques survenues dans les vingt dernières années:
  - capacités mémoires,
  - performances des processeurs et des réseaux,
  - évolutions des logiciels : interfaces graphiques, multimédia, des interfaces de communications.
- L'intérêt du modèle = les fonctionnalités nouvelles de l'informatique distribuée:
  - applications *peer to peer*
  - aspect économique: applications client/serveur avec les ordinateurs personnels,
- Adapté à l'organisation des sociétés modernes
  - ==> structurées en entités de moindre taille (filialisations)
  - ==> nouveaux besoins de communication (applications distribuées).

4

Introduction

- Architecture d'abord utilisée dans les systèmes «Time Sharing»
- S'étend de plus en plus vers tous les domaines d'activités :
  - gestion de base de données,
  - les systèmes transactionnels,
  - les systèmes de messagerie, Web, Intranet,
  - les systèmes de partage des données,
  - le calcul scientifique
  - etc.
- Les freins
  - difficulté de concevoir des applications distribuées,
  - manque de cohérence entre les applications clientes et serveurs,
  - manque d'outils d'administration des serveurs au niveau des services et des réseaux.
  - réticences des responsables pour des raisons de sécurité, de dispersion des données jugées sensibles,
  - incompatibilité avec les systèmes existants.

5

Plan

6

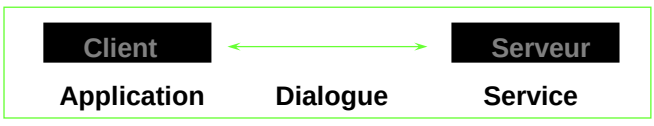
Le Modèle

- Repose sur une communication d'égal à égal entre les applications.
- Communication réalisée par dialogue entre processus deux à deux.
- Un processus est le client, l'autre le serveur;
- les processus ne sont pas identiques mais forment plutôt un système coopératif.

7

Le Modèle

- Le résultat de cette coopération se traduit par un échange de données, le client réceptionne les résultats finaux délivrés par le serveur.
- Le client initie l'échange,
- Le serveur est à l'écoute d'une requête cliente éventuelle.



Le service rendu = traitement effectué par le serveur, modèle client/serveur ==> répartition des services plutôt que l'application elle-même.

8

Le modèle

- Un service réparti est typiquement un service nécessitant beaucoup de ressources machine (CPU, mémoire résidente, mémoire secondaire, etc)
- ==> le service est exécuté sur une machine spécialisée appelée serveur.
- Le modèle client/serveur constitue un système coopératif sans distinction à priori entre les différents membres du réseau, ==> chacun des membres :
  - peut être indifféremment client et/ou serveur,
  - demander un service auprès d'un autre membre,
  - réaliser un service donné pour un ou plusieurs autres membres du réseau.

9

Plan

10

La Conception

- But : structurer les applications en clients et serveurs.
- Une application informatique est représentée selon un modèle en trois couches:
  - la couche présentation (interface Homme/Machine):
    - ✓ gestion de l'affichage (exemple Windows, X-windows, etc.),
    - ✓ logique de l'affichage, partie intrinsèque de l'applicatif qui transmet à la gestion de l'affichage, les éléments de présentation.
  - la couche traitements qui constitue la fonctionnalité intrinsèque de l'application :
    - ✓ la logique des traitements : l'ossature algorithmique de l'application,
    - ✓ la gestion des traitements déclenchés par la logique de traitements qui réalise la manipulation des données de l'applicatif (ex: procédures SQL).

11

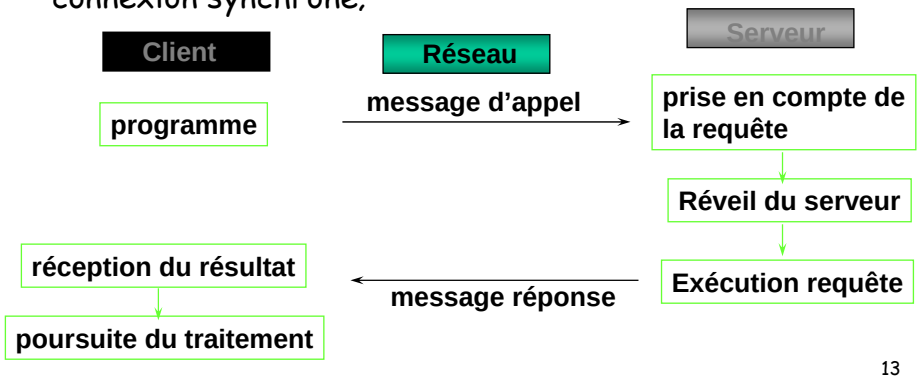
La Conception

  - la couche données qui assure la gestion des données applicatives:
    - ✓ la logique des données constituant les règles régissant les objets de la base de données,
    - ✓ la gestion des données (consultation et mise à jour des enregistrements). Un système de type SGBDR, habituellement, est responsable de cette tâche.
- Le découpage permet de structurer une application en mode client/serveur;
- exemple:
  - le module de gestion des données peut être hébergé par un serveur distant,
  - le module de gestion de l'affichage peut également être géré par un serveur distant (un Terminal X par exemple).

12

La conception

- Le mode de communication
- Mode non connecté, l'arrivée des données + ordonnancement + non duplication ne sont pas garantis par le protocole; ==> à gérer par l'application.
- l'approche non connecté implique généralement une connexion synchrone;



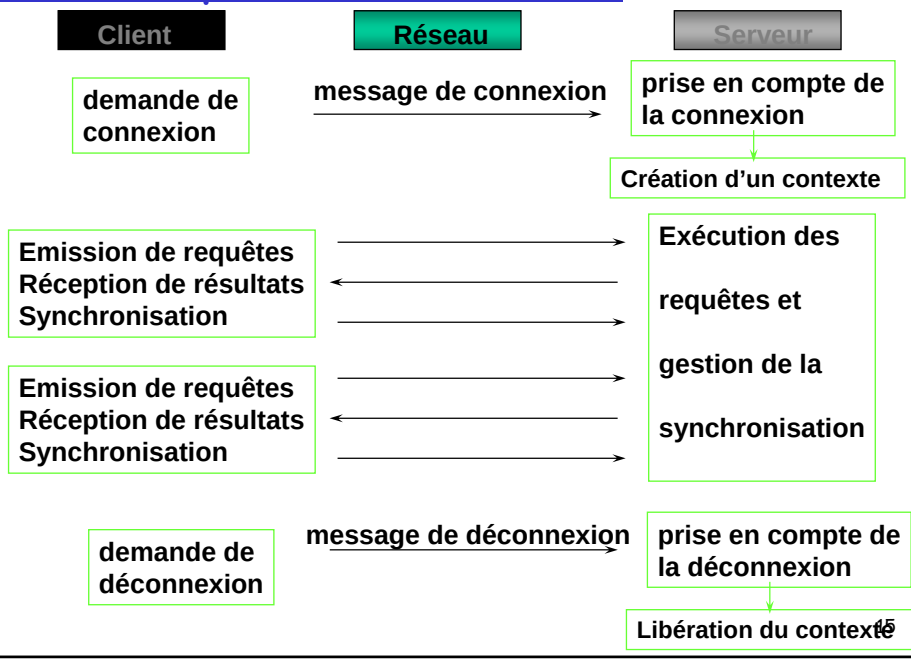
13

La conception

- le mode connecté implique une diminution des performances par rapport au mode non connecté: ceci est une contrainte pour certaines applications.
- le mode connecté permet une implémentation asynchrone des échanges, plus complexe mais plus performantes que le mode non connecté.

14

La conception (connexion)



15

La conception : architecture cliente

- Une application cliente est moins complexe que son homologue serveur car:
  - la plupart des applications clientes ne gèrent pas d'interactions avec plusieurs serveurs,
  - la plupart des applications clientes sont traitées comme un processus conventionnel; au contraire, un serveur nécessite des accès privilégiés de connexion
  - la plupart des applications clientes ne nécessitent pas de protection supplémentaires, le système d'exploitation assurant les protections élémentaires suffisantes.

16

Conception : architecture serveur

- Processus serveur:
  - Offre une connexion sur le réseau,
  - Entre indéfiniment dans un processus d'attente de requêtes clientes,
  - Lorsqu'une requête arrive, le serveur déclenche les processus associés à cette requête, puis émet la ou les réponses vers le client.
  - Problème : gérer plusieurs clients simultanément.
  - Les types de serveurs
    - serveurs itératifs: ne gèrent qu'un seul client à la fois
    - serveurs parallèles : fonctionnent « en mode concurrent ».

17

Conception : architecture serveur

- les serveurs itératifs en mode non connecté:
  - offrent une interface de communication sur le réseau en mode non connecté,
  - indéfiniment : réceptionnent une requête client, formulent une réponse, et renvoient le message réponse vers le client selon le protocole applicatif défini.
- les serveurs itératifs en mode connecté:
  - offrent une connexion sur le réseau en mode connecté,
  - réceptionnent une connexion client,
  - offrent une nouvelle connexion sur le réseau,
  - répétitivement : réceptionnent une requête pour cette connexion, formulent une réponse, et renvoient le message réponse vers le client, lorsque le traitement pour ce client est terminé .

18

Conception : architecture serveur

- les serveurs parallèles en mode non connecté:
  - offrent une interface de communication en mode non connecté,
  - répétitivement : réceptionnent la requête client; offrent une nouvelle interface de communication sur le réseau, et créent un processus secondaire (PR. S.) chargé de traiter la requête courante.
  - (PR. S.) : formule une réponse à la requête client, et renvoient le message,
  - (PR. S.) : lorsque le traitement est terminé, libère la communication.

19

Conception : architecture serveur

- les serveurs concurrents en mode connecté:
  - offrent une connexion sur le réseau en mode connecté,
  - répétitivement : réceptionnent une connexion client, offrent une nouvelle connexion sur le réseau, créent un PR. S. chargé de traiter la connexion courante.
  - (PR. S.) : répétitivement : réceptionne une requête pour cette connexion, formule une réponse, et renvoie le message réponse vers le client selon le protocole applicatif défini,
  - (PR. S.) : lorsque le traitement est terminé (propre au protocole applicatif), libère la connexion.

20

Conception : architecture serveur

Quel type de serveur utiliser ?

- serveurs itératifs en mode non connecté : services qui nécessitent très peu de traitement par requête (pas de concurrence).
- serveurs itératifs en mode connecté : services qui nécessitent très peu de traitement par requête mais requièrent un transport fiable de type TCP. Peu utilisé.
- serveurs concurrents en mode non connecté : pas utilisé sauf si :
  - temps de création d'un processus extrêmement faible (dépend du système d'exploitation hôte) par rapport au temps de traitement d'une requête,
  - les requêtes nécessitent des accès périphériques importants (dans ce cas, la solution itérative est, en effet, inacceptable).

21

Conception : architecture serveur

- serveurs concurrents en mode connecté : offre un transport fiable et est capable de gérer plusieurs requêtes de différents clients simultanément; implémentation:
  - multi instanciation de processus avec un processus primaire et des processus secondaires traitant les connexions clientes,
  - avec un seul processus gérant les multiples connexions par l'intermédiaire de requêtes asynchrones et primitive d'attente d'évènements multiples.

22

Plan

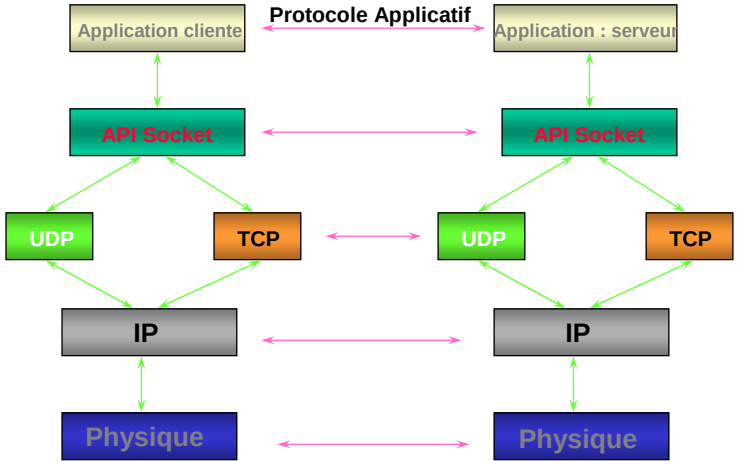
23

Les sockets

- Les sockets : interface client/serveur utilisée à l'origine dans le monde UNIX et TCP/IP.
- Etendue aujourd'hui du micro (Cf Winsock) au Mainframe.
- fournit les primitives pour le support des communications reposant sur toute suite de protocoles; les protocoles TCP/IP sont à l'origine des développements.
- Les applications cliente et serveur ne voient les couches de communication qu'à travers l'API socket
- Une socket est la porte entre le processus d'application et le protocole de transport de bout en bout

24

Les sockets



25

Sockets

- comme un descripteur de fichier dans le système UNIX,
- associe un descripteur à un socket;
- le concepteur d'application utilise ce descripteur pour référencer la communication client/serveur sous-jacente.
- une structure de données «socket» est créée à l' ouverture de socket;

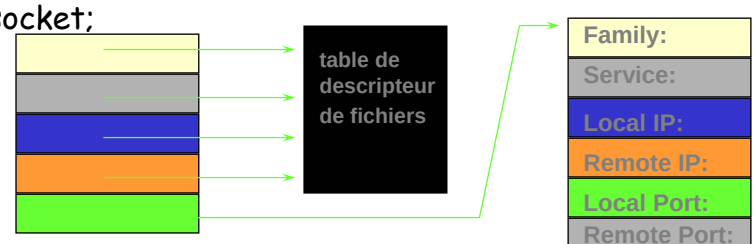


Table de descripteurs de processus

Structure Socket

La primitive *socket* permet l'ouverture de cette socket; initialement, après l'appel à cette fonction, la structure de données associée au socket est principalement vide, les appels à d'autres primitives de l'interface socket renseigneront ces champs vides.

26

Les Sockets : primitives

- Elles permettent d'établir un lien de communication en mode connecté ou non connecté sur un réseau,
- Structurent une application
  - soit en mode client ,
  - soit en mode serveur,
- Permettent d'échanger des données entre ces applications.
- **La primitive *socket: création***
  - point d'encrage qui permet à l'application d'obtenir un lien de communication vers la suite de protocole qui servira d'échange,
  - définit le mode de communication utilisé (connecté ou non connecté).
  - int socket (int famille, int type, int proto);
    famille: PF_INET pour IP
    type: SOCK_STREAM ou SOCK_DGRAM
    proto: protocole = 0 pour laisser le choix en fonction de type (TCP/UDP)

27

Les Sockets : primitives

- **La primitive *bind*:** nommage le socket

  permet de spécifier le point de terminaison local (essentiellement le port TCP/UDP dans l'environnement TCP/IP).nécessaire si le socket sera utilisé comme répondeur (serveur par exemple)

  int bind (int socket, struct sockaddr *myaddr, int addrlen);
- **la primitive *connect:***
  - permet à un client d'établir une communication active avec un serveur,
  - le point de terminaison distant (adresse IP + port TCP/UDP dans l'environnement TCP/IP) est spécifié lors de cet appel.

  int connect (int sock, struct sockaddr *correspondant, int addrlen);

  Si le socket n'est pas nommé le sysstème le nomme avec un numéro de port non utilisé

  Avec un socket STREAM, connect ne peut être utilisé q'une seule fois, sur socket DGRAM à volonté. Après connect dans une association sans connexion plus nécessaire de spécifier le destinataire de chaque datagramme et si aucun processus n'écoute le port spécifié sur la machine distante, indique une erreur

28

Les Sockets : primitives

- la primitive *listen* :
  - permet à un serveur d'entrer dans un mode d'écoute de communication ,
  - dés lors le serveur est « connectable » par un client,
- la primitive *accept* :
  - permet à un serveur de recevoir la communication entrante (client),
  - crée un nouveau socket et retourne le descripteur associé à l'application.
  - le serveur utilise ce descripteur pour gérer la communication entrante
  - le serveur utilise le descripteur de socket précédent pour traiter la prochaine communication à venir.
  - le processus est bloqué jusqu'à l'arrivée d'une communication entrante.

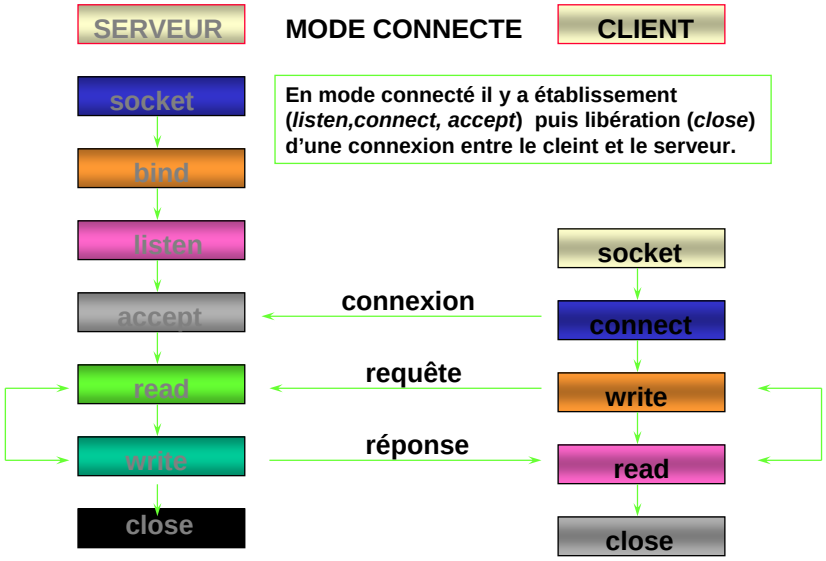
29

Les Sockets : primitives

- les primitives *read* et *write:*
  - Lorsque la communication est établie, client et serveur échangent des données afin d'obtenir (client) et transmettre (serveur) le service désiré.
  - En mode connecté, clients et serveurs utilisent *read* et *write*; en mode non connecté, ils utilisent les primitives *recvfrom* et *sendto*.
- la primitive *close* : termine la connexion et libère le socket associé.
- Le reste voir le document annexe

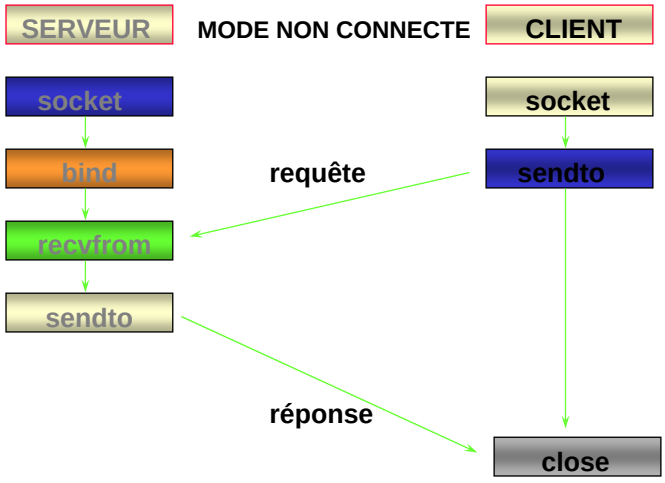
30

Les Sockets : Mode connecté



31

Les Sockets : Mode non connecté



32

Socket : Mode non connecté

En mode non connecté:

- le client n'établit pas de connexion avec le serveur mais émet un datagramme (sendto) vers le serveur.
- Le serveur n'accepte pas de connexion, mais attend un datagramme d'un client par recvfrom qui transmet le datagramme à l'application ainsi que l'adresse client.

- Les sockets en mode non connecté peuvent utiliser la primitive connect pour associer un socket à une destination précise. ==> send peut être utilisée à la place de la sendto,
- De même, si l'adresse de l'émetteur d'un datagramme n'intéresse pas un processus la primitive recv peut être utilisée à la place de la primitive recvfrom.

33

Socket : exemple de serveur itératif

```
int sockfd, newsockfd ;
if ( ( sockfd = socket (.....)) < 0 ) err_sys(«erreur de socket«) ;
if ( bind ( sockfd, ....) < 0 )              err_sys («erreur de
   bind»)
if ( listen ( sockfd , 5) < 0 ) ;            err_sys (« erreur de
   listen» ) ;

for ( ; ; ) {
   newsockfd = accept ( sockfd, .....) ;
   if ( newsockfd < 0)
        err_sys( «erreur de accept») ;

   execute_la_demande( newsockfd ) ;
   close ( newsockfd ) ;
}
```

34

Socket : exemple de serveur parallèle

```
int sockfd, newsockfd ;

if ( ( sockfd = socket (.....)) < 0 )          err_sys(«erreur de socket«) ;
if ( bind ( sockfd, ....) < 0 )        err_sys («erreur de bind») 
if ( listen ( sockfd , 5) < 0 ) ;    err_sys (« erreur de listen» ) ;

for ( ; ; ) {
    newsockfd = accept ( sockfd, .....) ;
    if ( newsockfd < 0)   err_sys( «erreur de accept») ;
    if ( fork() == 0 ) {
          close ( sockfd ) ;
          execute_la_demande( newsockfd ) ;
          exit (1) ;
          }
    close ( newsockfd ) ;
}
```

35

Sockets : gestion de noms

- Les primitives gethostname et sethostname
  - Dans le monde UNIX, la primitive *gethostname* permet aux processus utilisateurs d'accéder au nom de la machine locale.
  - D'autre part, la primitive *sethostname* permet à des processus privilégiés de définir le nom de la machine locale.
- La primitive getpeername
  - Cette primitive est utilisée afin de connaître le point de terminaison du distant.
  - Habituellement, un client connaît le point de terminaison (couple port/adresse IP) puisqu'il se connecte à ce serveur distant; cependant, un serveur qui utilise la primitive *accept* pour obtenir une connexion, a la possibilité d'interroger le socket afin de déterminer l'adresse du distant.
- La primitive getsockname
  - Cette primitive rend le nom associé au socket qui est spécifié en paramètre.

36

Sockets : gestion de noms

- Lorsque ces fonctions sont exécutées sur des machines ayant accès à un serveur de noms de domaines, elles fonctionnent elles-mêmes en mode client/serveur en émettant une requête vers le serveur de nom de domaines et attendent la réponse.
- Lorsqu'elles sont utilisées sur des machines qui n'ont pas accès à un serveur de noms, elles obtiennent les informations à partir d'une base de données (simple fichier) locale.
- *gethostbyname* spécifie un nom de domaine et retourne un pointeur vers une structure *hostent* qui contient les informations propres à ce nom de domaine.
- *gethostbyaddr* permet d'obtenir les mêmes informations à partir de l'adresse spécifiée.
- *getnetbyname* spécifie un nom de réseau et retourne une structure *« netent »en* renseignant les caractéristiques du réseau.
- *getnetbyaddr* spécifie une adresse réseau et renseigne la

37

Sockets : fonctions de service

- Les fonctions getprotobyname et getprotobynumber
  - Dans la base de données des protocoles disponibles sur la machine, chaque protocole a un nom officiel, des alias officiels et un numéro de protocole officiel.
  - La fonction *getprotobyname* permet d'obtenir des informations sur un protocole donné en spécifiant son nom; renseigne la structure *protoent*.
  - La fonction *getprotobynumber* permet d'obtenir les mêmes informations en spécifiant le numéro de protocole.
- La fonction getservbyname
  - Certains numéros de ports sont réservés pour les services s'exécutant au-dessus des protocoles TCP et UDP.
  - *getservbyname* retourne les informations relatives à un service donné en spécifiant le numéro du port et le protocole utilisé; renseigne la structure *servent.*

38

Sockets : Byte ordering

- TCP/IP spécifie une représentation normalisée pour les entiers utilisés dans les protocoles. Cette représentation, appelée *network byte order*, représente les entiers avec le MSB en premier.
- Une application doit renseigner certaines informations du protocole et par conséquent, doit respecter le *network* informations *byte order;* Exemple le numéro de port.
- Pour que les applications fonctionnent correctement, elles doivent translater la représentation des données de la machine locale vers le *network byte order* :
  - *htonl* : host to network long : convertit une valeur sur 32 bits de la représentation machine vers la représentation réseau.
  - *htons* : host to network short : convertit une valeur sur 16 bits de la représentation machine vers la représentation réseau.
  - *ntohl* : network to host long : convertit une valeur sur 32 bits de la représentation réseau vers la représentation machine.
  - *ntohs* : network to host short : convertit une valeur sur 16 bits de la représentation réseau vers la représentation machine.

39

Sockets : les options

- Une application peut contrôler certains aspects du fonctionnement des sockets:
  - configurer les valeurs des temporisations,
  - l'allocation de la mémoire tampon,
  - vérifier si le socket autorise la diffusion ou la gestion des données hors bande.
- La primitive getsockopt
- Permet à une application d'obtenir les informations relatives au socket. Le système d'exploitation exploite les structures de données internes relatives au socket et renseigne l'application appelante.

40

Sockets : serveur multi-protocoles

- Certains serveurs offrent leurs services sur plusieurs protocoles simultanément afin de satisfaire les clients qui nécessitent des transports, soit en mode connecté, soit en mode non connecté.
  - Exemple : DAYTIME port 13 sur UDP et sur TCP.
- Les services réalisés sur l'une ou l'autre interface fonctionnent différemment :
  - la version TCP utilise la connexion entrante du client pour déclencher la réponse (à une requête donc implicite): le client n'émet aucune requête.
  - la version UDP de DAYTIME requiert une requête du client. Cette requête consiste en un datagramme arbitraire nécessité pour déclencher l'émission de la donnée côté serveur. Ce datagramme est ensuite rejeté par le serveur.
- Dans de nombreux cas, un serveur fournit un service pour un protocole donné; par exemple, le service DAYTIME est réalisé par deux serveurs différents, l'un servant les requêtes TCP, l'autre les requêtes UDP.

41

Serveurs multi-protocoles

- Avantage à utiliser des serveurs différents réside dans le contrôle des protocoles et des services qu'offrent un système (certains systèmes, par exemple, ferment tout accès à UDP pour des raisons de sécurité)
- L'avantage à utiliser un serveur commun ou multi-protocoles :
  - non duplication des ressources associées au service, (corps du serveur),
  - cohérence dans la gestion des versions
- Fonctionnement
  - Un seul processus utilisant des opérations d'entrée/sortie asynchrones de manière à gérer les communications à la fois en mode connecté et en mode non connecté
  - Deux implémentations possibles : en mode itératif et en mode concurrent

42

Serveurs multi-protocoles

- En mode itératif
  - le serveur ouvre un socket UDP et un socket TCP,
  - Lorsqu'une requête TCP arrive, le serveur utilise *accept* provoquant la création d'un nouveau socket servant la communication avec le client
  - Lorsque la communication avec le client est terminée, le serveur ferme le troisième socket et réitère son attente sur les deux sockets initiales
  - Si une requête UDP arrive, le serveur reçoit et émet des messages avec le client (il n'y a pas d'*accept*); lorsque les échanges sont terminés, le serveur réitère son attente sur les deux sockets initiales
- Le mode concurrent
  - Création d' un nouveau processus pour toute nouvelle connexion TCP et traitement de manière itérative des requêtes UDP.
  - Automate gérant les événement asynchrones : optimisation maximale des ressources machines, puisque un seul processus traite toutes les variantes protocolaires des serveurs (TCP et UDP) et toutes les instances de services seront également traitées par le même processus.

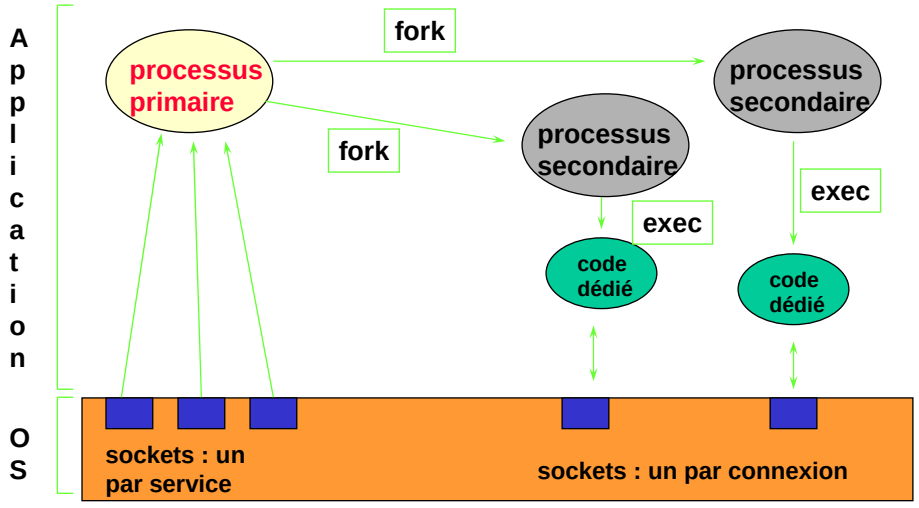
43

Sockets : les serveurs multi-services

- Problème lié à la multiplication des serveurs : le nombre de processus nécessaires et les ressources consommées qui sont associées.
- La consolidation de plusieurs services en un seul serveur améliore le fonctionnement:
- La forme la plus rationnelle de serveur multi-services consiste à déclencher des programmes différents selon la requête entrante : le fonctionnement d'un tel serveur en mode connecté est le suivant:
  - le serveur ouvre un socket par service offert,
  - le serveur attend une connexion entrante sur l'ensemble des sockets ouverts,
  - lorsqu'une connexion arrive, le serveur crée un processus secondaire (*fork* sous système UNIX), qui prend en compte la connexion,
  - le processus secondaire exécute (via *exec* sous système UNIX) un programme dédié réalisant le service demandé.

44

Sockets : serveurs multi-services



45

Sockets : Serveurs multi-services

AVANTAGES

- le code réalisant les services n'est présent que lorsqu'il est nécessaire,
- la maintenance se fait sur la base du service et non du serveur : l'administrateur peut gérer le serveur (modifie, archiver, ...) par service au lieu de le gérer globalement.
- Ce schéma est retenu en standard : le « super serveur » (*inetd* en BSD) consistant en un processus multi-services multi-protocoles offrant une interface de configuration (fichier systèmes) permettant à l'administrateur système d'ajouter de nouveaux services alors qu'aucun processus supplémentaire n'est nécessaire.

46

Plan

4.1 INTRODUCTION
4.2 LE MODELE
4.3 LA CONCEPTION
4.4 LES SOCKETS
- Introduction
- les primitives
- la structure d'adressage
- les serveurs multi-protocoles
- les serveurs multiservices

4.5 LES RPC

47

Les RPC

- Remote Procedure Call (RPC) : technologie permettant l'exécution de procédures situées dans des environnements distants.
- 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

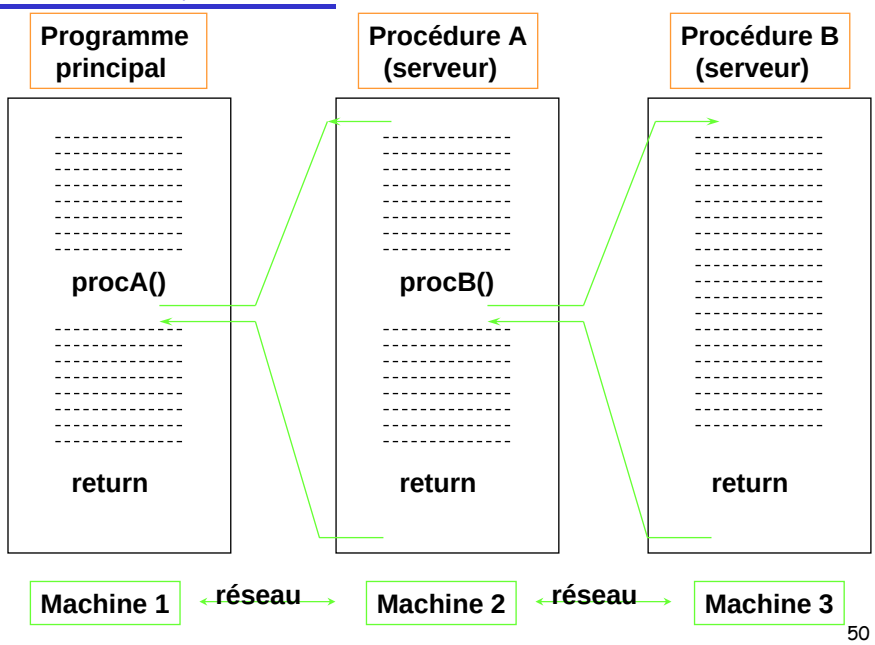
48

RPC : le modèle

- Le modèle RPC utilise l'approche «conception orientée application» et permet l'exécution de procédure sur des sites distants
- L'appel de la procédure distante constitue la requête cliente, le retour de la procédure constitue la réponse serveur
- but : conserver le plus possible la sémantique associée à un appel de procédure conventionnel, alors qu'il est mis en oeuvre dans un environnement totalement différent
- 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

49

RPC : le modèle



50

RPC : le modèle

Différences entre procédures distantes et procédures locales

- les temps de délais dus au réseau peuvent engendrer des temps d'exécution considérablement plus long,
- Un appel de procédure distant ne peut contenir d'argument de type pointeur,
- Les descripteurs d'entrée/sortie ne sont pas accessibles au procédures distantes, leur interdisant l'utilisation de périphériques locaux (exemple écriture de messages d'erreur impossible).

51