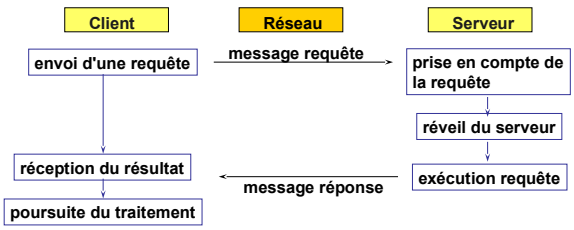
4-C : Programmation socket

Olivier GLÜCK
Université LYON 1/Département Informatique
Olivier.Gluck@univ-lyon1.fr
http://perso.univ-lyon1.fr/olivier.gluck

Copyright

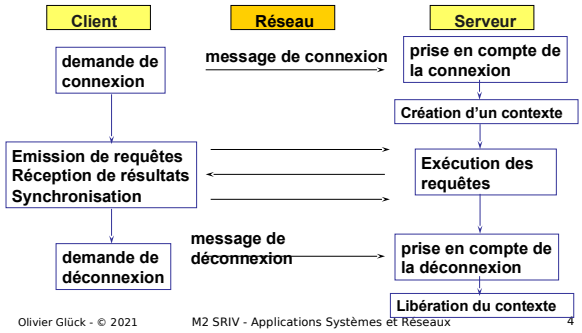
Les modes de communication

- Communication en mode non connecté



Les modes de communication

- Communication en mode connecté



Serveur itératif ou concurrent

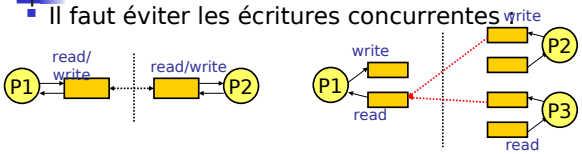
- Serveur itératif
  - traite séquentiellement les requêtes
  - adapté aux requêtes qui peuvent s'exécuter rapidement
  - souvent utilisé en mode non connecté (recherche de la performance)
- Serveur concurrent
  - le serveur accepte les requêtes puis les "délègue" à un processus fils (traitement de plusieurs clients)
  - adapté aux requêtes qui demandent un certain traitement (le coût du traitement est suffisamment important pour que la création du processus fils ne soit pas pénalisante)
  - souvent utilisé en mode connecté

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)

Les sockets

Lyon 1

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

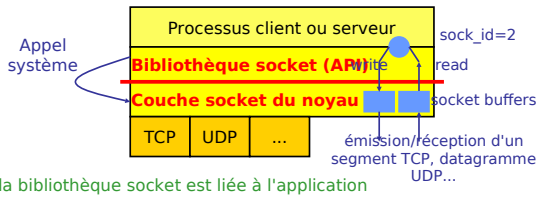
- Le serveur doit utiliser un numéro de port fixe vers lequel les requêtes clientes sont dirigées
- Les ports inférieurs à 1024 sont réservés :
  - "*well-known ports*"
  - ils permettent d'identifier les serveurs d'applications connues
  - ils sont attribués par l'IANA
- Les clients n'ont pas besoin 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

- Une socket est un fichier virtuel avec les opérations d'ouverture, fermeture, écriture, lecture, ...
- Ces opérations sont des appels système
- Il existe différents types de socket associés aux différents services de transport :
  - *stream sockets (connection-oriented)* - SOCK_STREAM
    - utilise TCP qui fournit un service de transport d'octets fiable, dans l'ordre, entre le client et le serveur
  - *datagram sockets (connectionless)* - SOCK_DGRAM
    - utilise UDP (transport non fiable de datagrammes)
  - *raw sockets* - SOCK_RAW
    - utilise directement IP ou ICMP (ex. *ping*)

Les sockets en pratique

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

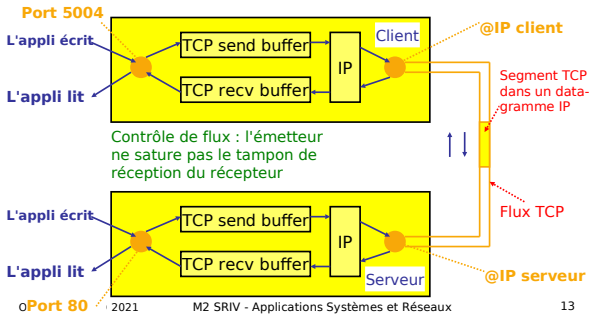


- la bibliothèque socket est liée à l'application

- la couche socket du noyau réalise l'adaptation au protocole de transport utilisé

Rappel - une connexion TCP

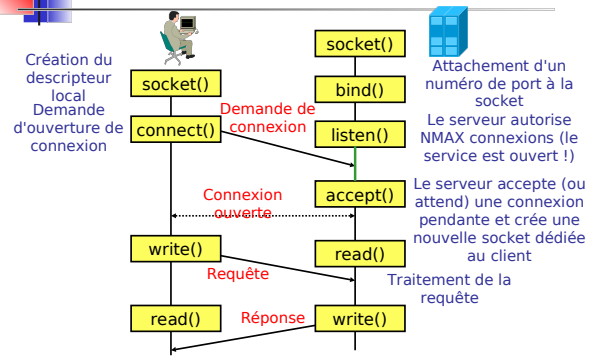
- Une connexion = **(proto, @IP_src, port_src, @IP_dest, port_dest)**



En mode connecté...

- Pour que le client puisse contacter le serveur
  - le processus serveur doit déjà tourner
  - le serveur doit avoir créé au préalable une socket pour recevoir les demandes de connexion des clients
- Le client contacte le serveur
  - en créant une socket locale au client
  - en spécifiant une adresse IP et un numéro de port pour joindre le processus serveur
- Le client demande alors l'établissement d'une connexion avec le serveur
- Si le serveur accepte la demande de connexion
  - il crée une nouvelle socket permettant le dialogue avec ce client
  - permet au serveur de dialoguer avec plusieurs clients

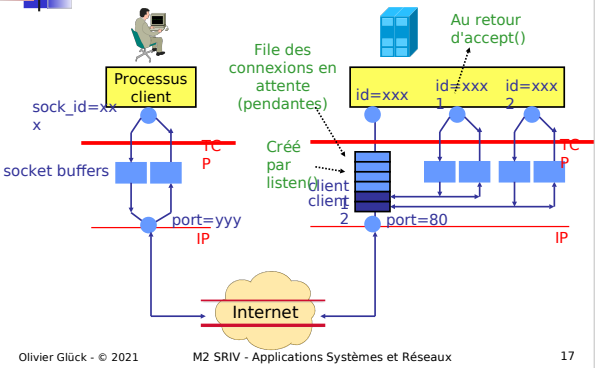
En mode connecté...



En mode connecté...

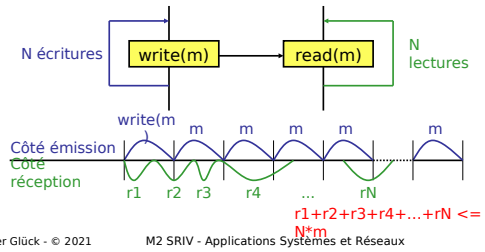
	Paramètres en entrée	Paramètres en sortie
socket()	type, domaine, protocole	sock_id
bind()	sock_id, port	
listen()	sock_id, NMAX	
connect()	sock_id, @sock_dest	
accept()	sock_id	@sock_src, client_sock_id
read()	client_sock_id, @recv_buf, lg	read_lg
write()	client_sock_id, @send_buf, lg	write_lg

En mode connecté...



En mode connecté...

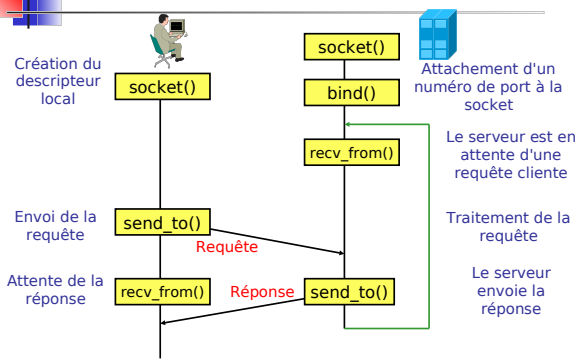
- Attention : les émissions/réceptions ne sont pas synchrones
  - *read(m)* : lecture **d'au plus** m caractères
  - *write(m)* : écriture de m caractères



En mode non connecté...

- Pour que le client puisse contacter le serveur
  - il doit connaître l'adresse de la socket du serveur
  - le serveur doit avoir créé la socket de réception
- Le client envoie sa requête en précisant, lors de chaque envoi, l'adresse de la socket destinataire
- Le datagramme envoyé par le client contient l'adresse de la socket émettrice (port, @IP)
- Le serveur traite la requête et répond au client en utilisant l'adresse de la socket émettrice de la requête

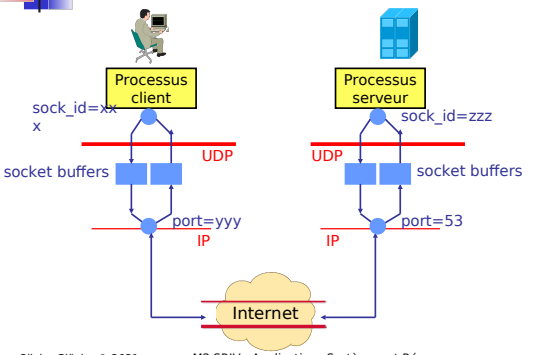
En mode non connecté...



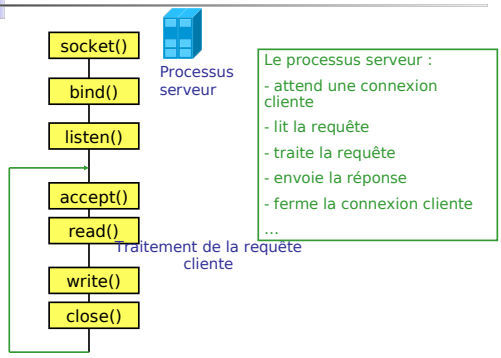
En mode non connecté...

	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

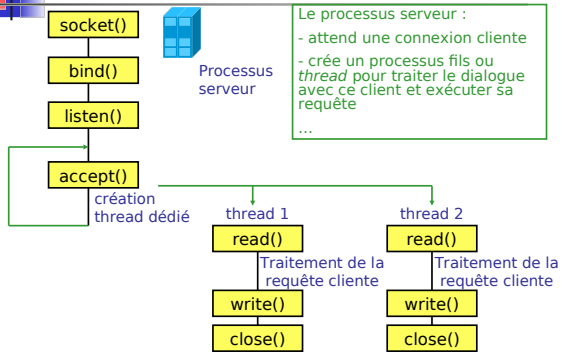
En mode non connecté...



Serveur itératif en mode connecté



Serveur concurrent en mode 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)

Opérations bloquantes/non bloquantes

- Comportement vis à vis de l'acceptation des connexions en mode non bloquant
  - s'il n'y a pas de connexion pendante, retourne -1 ... (l'application doit réessayer plus tard)
- Comportement vis à vis des demandes de connexions en mode non bloquant
  - la primitive `connect()` retourne immédiatement mais la demande de connexion n'est pas abandonnée au niveau TCP...

Paramétrage des sockets

- Les sockets sont paramétrables
  - fonctions `setsockopt()` et `getsockopt()`
  - options booléennes et non booléennes
- Exemples d'options booléennes
  - diffusion (dgram uniquement ; remplace l'@IP destinataire par l'@ de diffusion de l'interface)
  - `keepalive` : teste régulièrement la connexion (stream)
  - `tcpnodelay` : force l'envoi des segments au fur et à mesure des écritures dans le tampon
- Exemples d'options non booléennes
  - taille du tampon d'émission, taille du tampon de réception, type de la socket

Les serveurs multi-protocoles

- Un serveur qui offre le même service en mode connecté et non connecté
  - exemple : DAYTIME (RFC 867) port 13 sur UDP et sur TCP qui permet de lire la date et l'heure sur le serveur
  - 13/TCP : la demande de connexion du client déclenche la réponse (à une requête donc implicite) : le client n'émet aucune requête
  - 13/UDP : la version UDP de DAYTIME requiert une requête du client : cette requête consiste en un datagramme arbitraire qui n'est pas lu par le serveur mais qui déclenche l'émission de la donnée côté serveur
- Le serveur écoute sur 2 sockets distinctes pour rendre le même service

Les serveurs multi-protocoles

- Pourquoi un serveur multi-protocoles ?
  - certains systèmes ferment tout accès à UDP pour des raisons de sécurité (pare-feu)
  - non duplication des ressources associées au service (corps du serveur)
- Fonctionnement
  - un seul processus utilisant des opérations non bloquantes 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

Les serveurs multi-protocoles

- En mode itératif
  - le serveur ouvre la socket UDP et la socket TCP puis boucle sur des appels non bloquants à `accept()` et `recv_from()` sur chacune des sockets
  - si une requête TCP arrive
    - le serveur utilise `accept()` provoquant la création d'une nouvelle socket servant la communication avec le client
    - lorsque la communication avec le client est terminée, le serveur ferme la socket "cliente" et réitère son attente sur les deux sockets initiales
  - si une requête UDP arrive
    - le serveur reçoit et émet des messages avec le client
    - lorsque les échanges sont terminés, le serveur réitère son attente sur les deux sockets initiales

Les serveurs multi-protocoles

- En mode concurrent
  - un automate gère l'arrivée des requêtes (primitives non bloquantes)
  - création d'un nouveau processus fils pour toute nouvelle connexion TCP
  - traitement de manière itérative des requêtes UDP
    - elles sont traitées en priorité
    - pendant ce temps, les demandes de connexion sont mises en attente

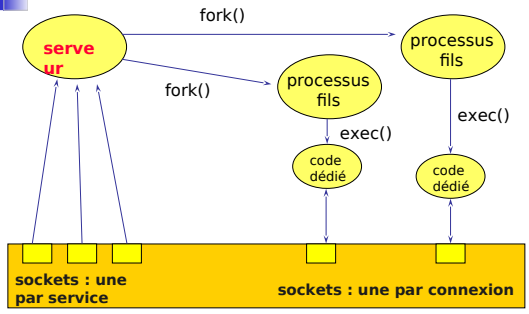
Les serveurs multi-services

- Un serveur qui répond à plusieurs services (une socket par service)
- Pourquoi un serveur multi-services ?
  - problème lié à la multiplication des serveurs : le nombre de processus nécessaires et les ressources consommées qui y sont associées
- 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 facilement activer ou désactiver un service

Les serveurs multi-services

- Fonctionnement : lancement d'un programme différent selon la requête entrante
  - le serveur ouvre une socket par service offert, attend une connexion entrante sur l'ensemble des sockets ouvertes
  - lorsqu'une demande de connexion arrive, le serveur crée un processus fils qui prend en compte la connexion
  - le processus fils exécute (via `exec()` sur système UNIX) un programme dédié réalisant le service demandé

Les serveurs multi-services



Les processus démons

- L'invocation d'un service Internet standard (FTP, TELNET, RLOGIN, SSH, ...) nécessite la présence côté serveur d'un processus serveur
  - qui tourne en permanence
  - qui est en attente des requêtes clientes
- On parle de démon
- A priori, il faudrait un démon par service
- Problème : multiplication des services --> multiplication du nombre de démons
- Sous UNIX, un super-démon : `inetd`

Le démon *inetd*

- Un "super serveur"
  - un processus multi-services multi-protocoles
  - un serveur unique qui reçoit les requêtes
  - activation des services à la demande
  - permet d'éviter d'avoir un processus par service, en attente de requêtes
  - une interface de configuration (fichier `inetd.conf`) permettant à l'administrateur système d'ajouter ou retirer de nouveaux services sans lancer ou arrêter un nouveau processus
- Le processus `inetd` attend les requêtes à l'aide de la primitive `select()` et crée un nouveau processus pour chaque service demandé (excepté certains services UDP qu'il traite lui-même)

Le fichier `/etc/inetd.conf`

```
#  Internet services syntax :
#  <service_name> <socket_type> <proto> <flags> <user> <server_pathname> <args>
#  wait : pour un service donné, un seul serveur peut exister à un instant donné
# donc le serveur traite l'ensemble des requêtes à ce service
# stream --> nowait : un serveur par connexion
ftp     stream          tcp    nowait  root   /etc/ftpd           ftpd -l
tftp    dgram           udp    wait    root   /etc/tftpd          tftpd
shell   stream          tcp    nowait  root   /etc/rshd           rshd
pop3    stream tcp      nowait  root    /usr/local/lib/popper   popper -s -d -t /var/log/poplog
# internal services :
# => service réalisé par inetd directement
time    stream          tcp     nowait          root    internal
time    dgram           udp     nowait          root    internal
```


La scrutation de plusieurs sockets

- Scrutation : mécanisme permettant l'attente d'un événement (lecture, connexion, ...) sur plusieurs points de communication
  - nécessaire dans le cas des serveurs multi-services ou multi-protocoles
- Problème lié aux caractères bloquants des primitives
  - exemple : une attente de connexion (`accept`) sur une des sockets empêche l'acceptation sur les autres...
- Première solution
  - rendre les primitives non bloquantes à l'ouverture de la socket
  - inconvénient : attente active (dans une boucle)

La scrutation de plusieurs sockets

- Deuxième solution
  - créer un fils par socket pour la scrutation d'un service
  - inconvénient : lourd, gaspillage de ressources
  - mais avantage conservé d'activation à la demande
- Troisième solution : la primitive `select()`
  - permet de réaliser un multiplexage d'opérations bloquantes (scrutation) sur des ensembles de descripteurs passés en argument :
    - descripteurs sur lesquels réaliser une lecture
    - descripteurs sur lesquels réaliser une écriture
    - descripteurs sur lesquels réaliser un test de condition exceptionnelle (arrivée d'un caractère urgent)
  - un argument permet de fixer un temps maximal d'attente avant que l'une des opérations souhaitées ne soit possible

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()`)

Exercices

Lecture bloquante/non bloquante

- Une application (client ou serveur) veut lire exactement 100 caractères sur une socket (mode connecté)
- Décrire l'algorithme correspondant et donnez les avantages/inconvénients
  - dans le cas d'une lecture complétement bloquante (read retourne quand tout est lu)
  - dans le cas standard des sockets (« au moins 1 »)
  - dans le cas d'une lecture non bloquante (-1 si EWOULDBLOCK)

Exemple de programmation C/S

- 1. Quel est le service proposé par cette application client/serveur ? Combien d'arguments sont nécessaires au lancement du client ? Quels sont-ils ?
- 2. Quel port utilise le serveur ? Aurait-on pu choisir une autre valeur ? Quel port utilise le client ? Comment est-il attribué et par quelle primitive ? S'agit-il d'une connexion en mode connecté ou non et est-ce justifié ?
- 3. A quoi correspondent les constants BUF_SIZE et QUEUE_SIZE ?
- 4. Quand est-ce que le serveur s'arrête ? Que fait le serveur une fois les initialisations terminées (décrire le cas où il y a des connexions pendantes et le cas inverse) ?
- 5. Que se passe t-il si le client est lancé avant que le serveur n'ait démarré ?
- 6. Quand est-ce que le client s'arrête si la connexion a réussi ? Que fait le client une fois la connexion établie ?
- 7. Que pensez-vous de la structure actuelle du serveur ? Peut-il satisfaire un grand nombre de connexions ? Expliquez. Proposez une solution plus adaptée.

Un mini-inetd

- Voici la page man du programme `mini-inetd` ainsi que son code.
- 1. Complétez la partie `DESCRIPTION` de la page man. Représentez à l'aide d'un schéma/diagramme la structure algorithmique du programme.
- 2. Dans le code ci-après, le code de la fonction `tcp_listen()` a volontairement été omis. Quelles sont les paramètres et la valeur de retour de cette fonction ? Quelles sont les opérations qui doivent y être réalisées et où les paramètres interviennent-ils ?
- 3. Commentez le nom du programme. Quelles sont les différences et similitudes entre `mini-inetd` et `inetd` ?
- 4. Comment modifieriez vous la structure donnée à la question 1 pour que `mini-inetd` puisse traiter plusieurs couples (port, program) passés en arguments ?