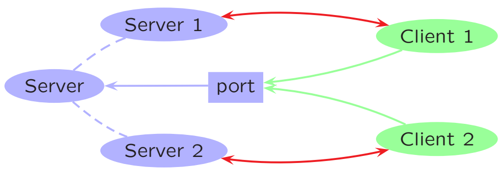
Communication inter-machines

- Les prises
- Connection à un service
- Wget
- Offre de service
- Serveur universel
- Primitives de haut-niveau
- Protocoles Clients-Serveurs

Le client-serveur (mode connecté)



Avec traitement concurrrent des requêtes

Le serveur est immédiatement disponible.

Insuffisance des tuyaux

- Communication locale à une machine.
- Pas bien adaptée au modèle client-serveur :
  - Le serveur accède seul à une resource partagée
  - Les clients accèdent à cette resource par l'intermédiaire d'une connection avec le serveur.
  - Le serveur sérialise et réglemente l'accès à cette ressource.
- On pourrait le faire avec des tuyaux nommés
  - le serveur crée un tuyau nommé à l'établissement du service et lit des demandes de connections sur celui-ci.
  - le client crée un tuyau nommé et fait une demande de connection au serveur en passant sur le tuyau de référence l'adresse de son propre tuyau.
  - problème de l'exclusion mutuelle entre les demandes de connection. Protocole réalisable mais complexe.

Les prises

Généralisation des tuyaux

- Elles abstraient le modèle client-serveur.
- Elles permettent la communication locale et distante (entre processus de différentes machines).
- Elles permettent différents protocoles de transmmissions de façon transparente.

Le protocole de branchement

- Le serveur propose un service sur un port dont l'adresse est publique et se met en écoute sur celui-ci.
- Le client fait une demande de branchement sur ce port. Si le branchement réussit, le client et le serveur reçoivent chacun une prise pour communiquer ensemble en privé.
- Plusieurs clients reçoivent des prises différentes pour communiquer avec le même serveur.

Schéma général

1. Création d'une prise

- On choisit le type de communication
- On choisit le domaine de la prise (locale *v.s.* distante)

Schéma général

1. Création d'une prise

2.a Branchement de la prise en mode serveur

1) On lie la prise à une adresse.
2) On se met en écoute sur la prise.
3) On accepte une connection, ce qui, à chaque connection, retourne une nouvelle prise connectée avec le client.
4) On traite la connection puis on reprend à l'étape précédente.
5) La fermeture de la prise ferme le service.

Schéma général

1. Création d'une prise

2.a Branchement de la prise en mode serveur

2.b Branchement de la prise en mode client

1) On demande à être connecté à une adresse.
2) On envoie les données et on écoute les réponses.
3) La fermeture de la prise ferme la connection.

Différents types de communication

Le type `socket_type` (voir cours réseau)

`SOCK_STREAM` :
Transmission fiable, par flot d'octets.
Le plus répandue. Transmission d'une suite d'octets sans structure particulière. Ex. `rsh`, `ssh`, `ftp` *etc.*

`SOCK_SEQPACKET` :
Transmission fiable, par paquets

`SOCK_DGRAM` :
Transmission non fiable, par paquets.
La plus proche du réseau, la plus économique : type internet.
Données sans importance.

`SOCK_RAW` :
Accès aux couches basses du réseau.

Les domaines de communication

Let type socket_domain

PF_UNIX
: Le domaine Unix : ne permet la communication qu'au sein d'une même machine.

PF_INET
: Le domaine Internet

Le prototole de communication

Définis dans /etc/protocols

- **i**nternet **p**rotocol, **u**ser **d**atagram **p**., **t**ransmission **c**ontrol **p**., *etc.*

Identification

- Représenté par un entier
- 0 laisse le système choisir le protocol (le plus fréquent).
  Il est déterminé en général à partir du numéro de port.
- On peut les consulter *de façon portable* par

```
getprotobyname : string -> protocol_entry
getprotobynumber : int -> protocol_entry
```

Exemple :

```
let tcp = getprotobyname "tcp";;
val tcp : Unix.protocol_entry =
  {p_name = "tcp"; p_aliases = [|"TCP"|]; p_proto = 6}
```

Création d'une prise (1.)

socket

```
socket : socket_domain -> socket_type -> int -> file_descr
```

Le troisième argument identifie le protocol (typiquement 0).

Une fois créée, une prise doit être « branchée » à une adresse.

socketpair (Dans le domaine PF_UNIX, pour info, guère utilisé)

```
socketpair : socket_domain -> socket_type -> int -> file_descr * file_descr
```

- socket_domain doit être PF_UNIX
- retourne une paire de prises « branchées » à une adresse.
- ce sont des « tuyaux jumelés » utilisables dans les deux sens.

```
let bipipe() = socketpair PF_UNIX SOCK_STREAM 0;;
```

Réglages d'une prise

Exemples de réglages

- Temporisation des émissions/réceptions,
- Synchronisation des opérations
- Tailles des buffeurs d'émission réception.
- Tailles minimal des opérations d'input/output (cf. select)

De nombreux réglages de types différents et un jeu de primitives

```
getsockopt : file_descr -> socket_bool_option -> bool
getsockopt_int : file_descr -> socket_int_option -> int
getsockopt_optint : file_descr -> socket_optint_option -> int option
getsockopt_float : file_descr -> socket_float_option -> float
```

Et leur variantes setsockopt....

(Voir la documentation—module Unix et man)

Réglages fréquents

Recyclage des prises La déconnection d'une prise TCP est négociée, ce qui prend un certain temps. Ce qui limite le risque qu'une nouvelle connection capture les paquets destinés à l'ancienne connection.

Deux réglages fréquents :

```
setsockopt_optint sock SO_LINGER (Some 5);
```

Alors close attendra (au plus 5s) que les données émisses soient effectivement transmisses avant de retourner.

Et en TD :

```
setsockopt sock SO_REUSEADDR true;
```

L'option SO_REUSEADDR dit à la commmande bind qu'une prise peut être réallouée à une adresse locale sur laquelle toutes les communications sont en cours de déconnection.

Les adresses

Type socketaddr

Constructeur	Type	Argument
ADDR_UNIX	string	nom de fichier
ADDR_INET	inet_addr * int	machine et numéro de port

Exemple

```
let local_socket_addr =
  ADDR_UNIX "/tmp/foo";;
let cours_server_addr =
  ADDR_INET (inet_addr_of_string "128.93.11.35", 80);;
```

Les adresses internet

Représentation

- Elles ont des noms symboliques de la forme

  pauillac.inria.fr

  www.enseignement.polytechnique.fr
- qui correspondent à des numéros de la forme $xxx.yyy.zzz.ttt$.
- la représentation interne est binaire : type abstrait inet_addr.

Les adresses internet

Représentation

Base de donnée

Associe des numéros à des noms symboliques.

- Fichier local /etc/hosts + « namerservers ».
- Accès transparent :

```
gethostbyname : string -> host_entry
gethostbyaddr : inet_addr -> host_entry
```

Exemple

```
let pauillac =
  gethostbyname "pauillac.inria.fr";;
```

```
val pauillac : Unix.host_entry =
  {h_name = "pauillac.inria.fr"; h_aliases = [|"pauillac"|];
   h_addrtype = PF_INET; h_addr_list = [|<abstr>|]}
```

Les adresses internet

Représentation

Base de donnée

Conversions

```
inet_addr_of_string : string -> inet_addr
string_of_inet_addr : inet_addr -> string
inet_addr_any : inet_addr
```

Pour voir la vraie adresse de `pauillac` :

```
string_of_inet_addr pauillac.h_addr_list.(0);;
 - : string = "128.93.11.35"
```

Définir l'adresse du réseau local :

```
let inet_addr_loopback = inet_addr_of_string "127.0.0.1"
```

Les ports

Un port représente un service

- Ce sont des numéros entiers sur 16 bits.
- `/etc/services` associe un port à un service et un protocol.

On y accède de façon abstraite par

```
getservbyname : string -> string -> service_entry
getservbyport : int -> string -> service_entry
```

```
let http = getservbyname "http" "tcp";;
```

```
val http : Unix.service_entry =
  {s_name = "http"; s_aliases = [|"www"; "www-http"|]; s_port = 80;
   s_proto = "tcp"}
```

Les ports

Un port représente un service

Choix d'un numéro pour offrir un service

Port	Catégorie	Droits, usage
0-1023	`Well-known`	Superuser
1024-49151	`Registered`	Utilisateur, conflits possibles
49152-65535	`Private`	Utilisateur, libres

Utiliser un service (2.b)

1) On lie la prise à l'adresse du service

```
connect : file_descr -> sockaddr -> unit
```

2) On peut alors parler au serveur

- en écrivant les données dans le descripteur (`write`)
- en lisant les réponses dans le descripteur (`read`)

Lectures et écritures se comportent comme sur un tuyau :

- `read` bloque s'il n'y a pas de données.
- `write` bloque s'il il a trop de données
- Si la connection a été fermée `read` renvoie 0 et `write` déclenche le signal `sigpipe` et échoue avec `EPIPE`.

3) On ferme la prise (ferme la connection)

Récupération d'une page WEB


```
val retransmit : file_descr -> file_descr -> unit
(** recopie le contenu du 1er descripteur dans le 2nd *)

let wget machine path =
  let port = (getservbyname "http" "tcp").s_port in
  let host = (gethostbyname machine).h_addr_list.(0) in
  let addr = ADDR_INET (host, port) in
  let sock = socket PF_INET SOCK_STREAM 0 in
  connect sock addr;
  let request = "GET " ^ path ^ "HTTP/1.0\r\n\r\n" in
  ignore (write sock request 0 (String.length request));
  retransmit sock stdout;
  close sock;;
```

Comme d'habitude... se protéger contre les erreurs qui ne sont pas fatales.

Offre de service (2.a)

1) Lier la prise à une adresse

```
bind : file_descr -> sockaddr -> unit
```

`bind` f s lie la prise f à l'adresse s.

Le descripteur f sera utilisé pour écouter et récupérer les connections.

- On fournit le service à une adresse particulière (de la machine)
  `gethostbyname(gethostname()).h_addr_list.(0)`
- ou bien à toutes les adresses de la machine en utilisant l'adresse spéciale `inet_addr_any`

Offre de service (2.a)

2) Autoriser les connections

```
listen : file_descr -> int -> unit
```

`listen` f n autorise les connections sur les adresses où la prise f a été branchée.

n est le nombre de connections qui peuvent être mises en attente avant leur ouverture, typiquement quelques dizaines : au délà `connect` par le client échouera.

Attention ! ce n'est pas le nombre maximale de connections en cours, mais de demandes d'ouvertures pas encore satisfaites.

Offre de service (2.a)

3) Accepter une connection

```
accept : file_descr -> file_descr * sockaddr
```

`accept` f attend une connection sur l'un des ports où le descripteur f à été branché et mis en écoute et retourne une nouvelle prise pour comuniquer avec le client.

4) Traiter la connection

- Soit séquentiellement par le serveur
  - les autres connections sont bloquées pendant ce temps.
- Soit en faisant traiter la connection par un processus fils
  - permet au serveur de traiter une autre connection
  - permet le traitement des connections en parallèles
    (le traitement peut être long, *e.g.* session `ftp` interactive)

Retour en 3)

Fermeture d'une connection

Par le client ou par le serveur

Abrupte en faisant `close` sur le descripteur.

- Ne permet pas de se dire «au revoir»
  (*e.g.* vider les buffeurs, envoyer un récepissé, *etc.*)

Douce par le client (en général) ou le serveur (possible)

```
shutdown : file_descr -> shutdown_command -> unit
```

Ferme l'autre bout de la prise.

SHUTDOWN_SEND	en lecture : `read` retournera `End_of_file`
SHUTDOWN_RECEIVE	en écriture : `write` provoquera `sigpipe`
SHUTDOWN_ALL	les deux combinés

L'autre peut encore lire ou écrire selon le cas, puis faire `close`.

5) Fermeture d'un service

- Un service est créé en liant une prise avec `bind` (rappel)
- La fermeture de la prise ferme le service et libère la prise, *i.e.* permet au système de la désallouer, mais la déallocation n'a pas nécessairement lieu immédiatement (utiliser `SO_REUSEADDR` pour permettre sa réutilisation immédiatement.)
- Un service créé dans le domaine Unix doit être fermé avec `unlink`.

Exemple : Un server universel


```
let server port cmd arg =
  let sock = socket PF_INET SOCK_STREAM 0 in
  bind sock (ADDR_INET(inet_addr_any, port));
  listen sock 10;
  while true do
    let (s, calleraddr) = accept sock in
      match fork() with
        0 -> if fork() <> 0 then exit 0;
             dup2 s stdin; dup2 s stdout; dup2 s stderr;
             close s;
             execvp cmd arg
      | k -> ignore(waitpid [] k);
             close s (* loguer la connection *)
  done;;
```

Exemple : Un server universel


```
let server port cmd arg =
  let sock = socket PF_INET SOCK_STREAM 0 in
  bind sock (ADDR_INET(inet_addr_any, port));
  listen sock 10;
  ignore (signal sigchld (Signal_handle free_children));
  while true do
    try
      let (s, calleraddr) = accept sock in
      match fork() with
        0 -> dup2 s stdin; dup2 s stdout; dup2 s stderr;
           close s; execvp cmd arg
      | _ -> close s (* loguer la connection *)
    with Unix_error(EINTR,_,_) -> ()
  done;;
```

Mise en place

Écrire un programme `server` qui «sert» une commande :

```
let main () =
  if Array.length Sys.argv > 2 then
  let port = int_of_string (Sys.argv.(1)) in
  let args = Array.sub 2 (Array.length Sys.argv - 2) in
  server port args.(0) args in
handle_unix_error main ();;
```

Puis exécuter sous le shell :

```
server 1024 date &
pid=$!
telnet localhost 1024
kill $pid
```

Commandes de haut niveau

Client

```
open_connection :
  sockaddr -> Pervasives.in_channel * Pervasives.out_channel
shutdown_connection : Pervasives.in_channel -> unit
```

Server

```
establish_server :
  (Pervasives.in_channel -> Pervasives.out_channel -> unit)
   -> sockaddr -> unit
```

Récupération d'une page WEB (revue)

On remplace (version bas-niveau + utilisation de retransmit) :

```
  let sock = socket PF_INET SOCK_STREAM 0 in
  ignore (connect sock addr);
  let request = "GET " ^ path ^ "HTTP/1.0\r\n\r\n" in
  ignore (write sock request 0 (String.length request));
  retransmit sock stdout;
  close sock;;
```

par (version haut-niveau + écritures tamporisés)

```
  let fd_in, fd_out = open_connection addr in
  let request = "GET " ^ path ^ "HTTP/1.0\r\n\r\n" in
  output_string fd_out request; flush fd_out;
  try while true do print_endline (input_line fd_in) done
  with End_of_file -> shutdown_connection fd_in;;
```

Le server (haut-niveau)

Un serveur qui capitalise ce qu'on lui donne...

```
let server port =
  let addr = ADDR_INET (inet_addr_any, port) in
  let treat_connection chan_in chan_out =
    try
      let l = input_line chan_in in
      if l = ".\r" || l = "." then raise End_of_file;
      output_string chan_out (String.uppercase l);
      output_char chan_out '\n'; flush chan_out;
    done with End_of_file -> () in
  establish_server treat_connection addr;;
```

Protocoles clients-serveurs

Le client et le serveur doivent s'entendre sur le protocole,
i.e. «parler le même langage».

Les protocoles binaires

- Transmission de données sous une forme compacte, proche de leur représentation en mémoire (*e.g.* X-Windows)
  - entiers, flotants : sur 4 ou 8 octets.
  - chaînes : longueurs, suivi du contenu.
  - tableau : longueur plus les éléments séquentiellement.
  - objets structurés à condition d'en connaître le type.
- En OCaml, le codage est automatique à partir du type :

```
output_value : 'a -> out_channel
input_value : in_channel -> 'a
```

  Suppose que lecteur et écrivain s'entendent sur le type `'a`.

Échange de valeurs structurées

Exemple : X Windows en OCaml

```
type request =
  | FillPolyReq of (int * int) array * drawable * ...
  | GetAtomNameReq of atom
  | ...
and reply =
  | GetAtomReply of string
  | ...
```

- on envoie une suite de valeurs du type `request` encodées.
- on reçoit une suite de valeurs du type `reply` encodées.
- Le codage/décodage appelé marshalling/unmarshalling
  est (presque) transparent en OCaml (fonction de librairie).
- *c.f.* le protocole e**X**ternal **D**ata **R**epresentation format.

C'est anodin (pas très profond), mais très pratique.

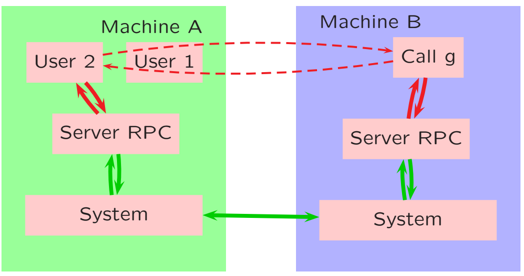
Échange de valeurs structurées

Exemple : Serveur de commandes

```
type request =
  | GetDate
  | GetFileBlock of string * int * int
  | Exec_to_string of string array
and reply =
  | Date of Unix.tm
  | FileBlock of string
  | Return of string array
```

Remote Procedure Call

Effectuer un calcul sur une machine distante



User 2 peut faire un appel à g sur B en faisant une connection locale à son serveur RCP.
Voir les services décrits dans `/etc/rpc` dont NFS.

Protocoles texte

Ce sont des petits langages de commandes

- Peuvent être interactifs, lus par des humains.
- La vitesse de transfert n'est pas cruciale.
- Certaines commandes peuvent passer en mode binaire le temps d'un transfert important

Exemples :

- ftp (fichiers), http (WEB), smtp (courrier), nntp (News)

Exemple des protocoles HTTP

HTTP/0.9 : fusil à un coup

- Le protocole
  - le client envoie une commande simple «`GET` *uri*»
  - le serveur répond puis ferme la connection.
- Problèmes :
  - Protocol simple, mais innefficace.
  - Le résultat est toujours une page HTML : les messages d'erreurs sont encodés à l'intérieur de la page.

Exemple des protocoles HTTP

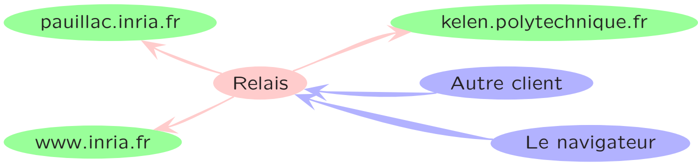
HTTP/0.9 : fusil à un coup

HTTP/1.1 : véritable dialogue

- Le protocole
  - Le client envoie une requête
  - La réponse du serveur contient
    - Une entête qui décrit si la requête s'est bien déroulée, la forme des données transmises, la façon dont elles sont transmises (par bloc ou par tronçon),*etc.*
    - éventuellement des données (corps de la réponse).
- Avantages :
  - Plusieurs requêtes par connection.
  - Requêtes conditionnelles (*Ex :* ne pas envoyer les données si elles n'ont pas changé depuis une certaine date)

Relais HTML

Connection relayée



- Un seul interlocuteur pour le client.
- Au passage, le relais peut ajouter des filtres (pub, contrôle d'accès), ou des caches (diminuer le transfert).
- Le relais peut partager le travail pour plusieurs clients.

Le mode déconnecté

Contrairement au protocole `tcp` (qui ne fonctionne qu'en mode connecté), le protocol `udp` ne fonctionne qu'en mode déconnecté. Pour établir un échange client-serveur :

1. Le serveur crée une prise dans le domaine UDP.
2. Il l'attache à un port (de type UDP)
3. La communication peut avoir lieu, directement par échange de (petits) paquets, par exemple taille $< 1k$.
   - Le serveur appelle `recvfrom` qui retournera un message et l'adresse du client, à laquelle le serveur peut répondre.
   - Le client envoie un message par `sendto`, puis (éventuellement) écoute la réponse du serveur.
   - La livraison des messages n'est pas garantie.

On peut quand même faire une connection, mais c'est une connection «virtuelle» qui mémorise l'adresse du destinataire dans la prise qui est utilisée à chaque message.

Sujets de projets en système

N'oubliez pas de choisir votre projet...

Voir information disponible en ligne :

- RCAML : exécuter des commandes sur une machine distante
- RSYNC : copie d'une hiérarchie sur une machine distante
- HTTP : Proxy HTML/1.1
- BTOR : Bit-Torrent
- ARCH : Serveur de sauvegarde.
- MODF : Modélisation du système de fichiers.
- PERSO : Projet personnel (avec accord, bien sûr).