430 Sockets

INF3173
Principes des systèmes d'exploitation

Jean Privat

Université du Québec à Montréal

Hiver 2021

Communication par sockets

- POSIX sockets alias BSD sockets alias Berkeley sockets
- Pour la communication **réseau** entre **processus** `socket(7)`
- API offerte par le système d'exploitation

Socket ?

- Point de communication **abstrait**
- Boîte d'émission et de réception

$\rightarrow$ Un socket est un **descripteur de fichier**

Ceci n'est pas un cours de réseau

- On fait juste communiquer des processus
- On implémente des protocoles de communication
- Et on expose des abstractions et services aux processus

$\rightarrow$ C'est la responsabilité du système d'exploitation

API des sockets

API commune

- Différents et nombreux protocoles
- Différents types de communication
- Y compris propriétaires ou désuets

API générale

- Abstractions et appels système communs
- Mais détails spécifiques à chaque protocole
- Et à chaque variante Unix
- API complexe avec défauts de conception historiques : berk!

Autre sockets

- « Socket » devenu un terme générique
- Autres langages et systèmes ont leur propre API de sockets
- API souvent proche (concepts et vocabulaire), parfois meilleure

Types de communication

- 3 dimensions principales
- Nombreuses variations spécifiques

Granularité

- Flux d'octets (*stream*)
- Messages (*datagram*, *packet*)

Connectivité

- Connecté et bidirectionnel : modèle client-serveur
- Non connecté : modèle pair à pair

Fiabilité (réseau principalement)

- Fiable : service garanti, obligation de résultat
  Risques de sacrifices : moins de débit et plus de latence
- Non fiable : service au mieux, obligation de moyen
  Risques de pertes de données, modifications du contenu, pertes de l'ordre, duplications

Petite sélection d'appels système

- `socket(2)`, `socketpair(2)` création de sockets
- `bind(2)`, `listen(2)`, `accept(2)` coté serveur
- `connect(2)` coté client
- `write(2)`, `send(2)`, `sendto(2)`, `sendmsg(2)` émission
- `read(2)`, `recv(2)`, `recvfrom(2)`, `recvmsg(2)` réception
- `close(2)`, `shutdown(2)` fermeture
- `getsockopt(2)`, `getsockopt(2)`, `ioctl(2)` configuration
- `getsockname(2)`, `getpeername(2)` identification
- `lseek(2)` bien évidement interdits (erreur `ESPIPE`)

Création de socket

```
socket(int domain, int type, int protocol)
```

Domaine = famille de protocoles

- `AF_INET` pour IPv4 (`ip(7)`) ou `AF_INET6` pour IPv6 (`ipv6(7)`)
- `AF_UNIX` (ou `AF_LOCAL`) pour socket Unix (on va y venir)
- plus de 20 chez Linux, AF = *address family*

Type = sémantique de la communication

- `SOCK_STREAM`: flux d'octets, connecté, fiable
  Exemple: TCP chez IP (`tcp(7)`). Analogie: téléphone
- `SOCK_DGRAM`: messages, non connecté, non fiable
  Exemple: UDP chez IP (`udp(7)`). Analogie: courrier postal
- `SOCK_SEQPACKET`: messages, connecté, fiable

Protocole

- Protocole particulier si plus d'un pour un domaine et un type
- 0 = protocole par défaut

Socket du domaine Unix

- `AF_UNIX` (ou `AF_LOCAL`). Voir `unix(7)`
- `SOCK_STREAM`, `SOCK_DGRAM` ou `SOCK_SEQPACKET`

Ressemblances avec les tubes

- Communication **efficace** via la mémoire
- Zones de mémoire gérées par le système d'exploitation
- Processus lisent/écrivent dans des descripteurs
- Anonymes ou nommés
- Synchronisation
  - Lecture si vide: bloquée ou 0 si aucun écrivain
  - Écrivain `SIGPIPE` si aucun lecteur ou bloqué si plein

Différence avec les tubes

- Utilise l'API des sockets POSIX
- Bidirectionnel
- Connecté ou non connecté
- Flux d'octets ou messages

Adresse de socket

- Désignation d'un socket existant ou potentiel
- Structures C semi-opaques, fragiles et contraignantes (berk!)
- Détails spécifiques à chaque domaine
  Exemple chez IP: adresse IP + numéro de port

Structures d'adresses

- `struct sockaddr`: structure abstraite
  - Utilisée dans les signatures des appels système
- `struct sockaddr_XXX`: une version spécifique à chaque domaine
  - Utilisées pour allouer et accéder aux champs
  - `struct sockaddr_in6` pour IPv6
  - `struct sockaddr_un` pour les sockets Unix
- `struct sockaddr_storage` structure **assez grande** pour stocker n'importe quelle structure spécifique

$\rightarrow$ On caste allègrement entre des pointeurs de ces types (berk!)

Sockaddr du domaine Unix

```
struct sockaddr_un {
  sa_family_t sun_family;     /* AF_UNIX */
  char        sun_path[108]; /* Chemin */
};
```

Attention, `sun_path` a une taille max (berk!) non portable (reberk!)

Fichier spécial socket

- Utilisé pour « nommer » les socket*
- Type « `s` » selon `ls -l`
- Créé par `bind(2)` (on y reviendra)
- Supprimé par `unlink(2)`
- `open(2)` échoue (`ENXIO`)

*Linux offre aussi des sockets avec des noms « abstraits » indépendants du système de fichiers (non portable).

Envoyer et recevoir

Envoyer

- `write(int fd, const void *buf, size_t len)`
- `send(int fd, const void *buf, size_t len, int flags)`
- `sendto(int fd, const void *buf, size_t len, int flags, const struct sockaddr *addr, socklen_t addrlen)`
- `sendto` = `send` + addr = `write` + flags + addr
- `sendmsg(int fd, const struct msghdr *msg, int flags)`

Recevoir

- `read(int fd, void *buf, size_t len)`
- `recv(int fd, void *buf, size_t len, int flags)`
- `recvfrom(int fd, void *buf, size_t len, int flags, struct sockaddr *addr, socklen_t *addrlen)`
- `recvfrom` = `recv` + addr = `read` + flags + addr
- `recvmsg(int fd, struct msghdr *msg, int flags)`

Note: ne pas préciser `addr` si connecté.

Mode connecté

Serveur

- `bind(int fd, const struct sockaddr *ad, socklen_t adlen)`
  Expose une « adresse » publique
- `listen(int fd, int backlog)`
  Prépare un serveur à recevoir des clients
  `backlog` est soumis au culte du cargo (berk!). `SOMAXCONN` est bien.
- `accept(int fd, struct sockaddr *ad, socklen_t *adlen)`
  Récupère ou attend le prochain client
  - Retourne un **nouveau** socket, connecté directement au client
  - On a donc un socket d'écoute + un socket par client connecté

Client

- `connect(int fd, const struct sockaddr *ad, socklen_t adlen)`
  - Se connecte à un serveur spécifique
  - Retourne 0 si réussi, `fd` est maintenant **connecté**

  → `read` et `write` fonctionnent !

Exemple de client socket unix

```c
#include "machins.h"
int main(int argc, char **argv)
{
  int sock = socket(AF_UNIX, SOCK_STREAM, 0);

  struct sockaddr_un addr;
  addr.sun_family = AF_UNIX;
  strncpy(addr.sun_path, "sock", sizeof(addr.sun_path)-1);

  int res = connect(sock, (struct sockaddr*)&addr, sizeof(addr));
  if(res==-1) { perror("connect"); exit(1); }

  write(sock, "Hello", 6);

  char buf[6];
  read(sock, buf, 6);
  printf("reçu: %s\n", buf);

  close(sock);
  return 0;
}
```

Exemple de serveur socket unix

```c
#include "machins.h"
int main(int argc, char **argv)
{
  int sock = socket(AF_UNIX, SOCK_STREAM, 0);

  struct sockaddr_un addr;
  addr.sun_family = AF_UNIX;
  strncpy(addr.sun_path, "sock", sizeof(addr.sun_path)-1);

  int res = bind(sock, (struct sockaddr *) &addr, sizeof(addr));
  if (res == -1) { perror("bind"); exit(1); }
  listen(sock, SOMAXCONN);

  int cli = accept(sock, NULL, NULL);
  write(cli, "World", 6);
  char buf[6];
  read(cli, buf, 6);
  printf("reçu: %s\n", buf);
  close(cli);

  close(sock);
  unlink("sock");
  return 0;
}
```

Modèles populaires de serveurs (1/2)

Un client après l'autre

- Boucle principale de `accept(1)`
- Traite chaque client entièrement, et dans l'ordre
- Problèmes
  - Traitements courts seulement
  - Un client peut bloquer les autres

Multiplexage

- Une liste de clients connectés
- Boucle principale avec un `select(2)` ou `poll(2)`
  - surveille le socket d'écoute + chacun des clients connectés
  - socket d'écoute bouge = on accepte un nouveau client
  - socket d'un client bouge = on traite sa demande
- Problèmes : messages courts seulement, pas adapté aux cas compliqués

Modèles populaires de serveurs (2/2)

Multithread

- Un thread principal écoute
- On lance un nouveau thread par client (ou pool de threads)
- Problèmes : programmation multithread

Multiprocessus

- Un processus principal écoute
- Un sous-processus (`fork(2)`) par client (ou pool de processus)
- Problème : lourd et isolation des clients
- Avantage : robuste et isolation des clients

Données auxiliaires

- Données spécifiques supplémentaires aux messages
- Alias « messages de contrôle » (*cmsg*)
- Contenu sémantique et spécifique :
  Contenu ont du sens pour le système d'exploitation
- Mais ce qui est possible est spécifique à chaque domaine
- `recvmsg(2)` et `sendmsg(2)` pour les utiliser
- `cmsg(3)` pour y accéder
- API horrible (berk!)

Descripteurs de fichiers auxiliaires

- Utilisable dans les sockets du domaine Unix
- `SCM_RIGHTS` passe des fichiers ouverts
- L'émetteur attache des **descripteurs de fichiers**
- Le système crée des descripteurs dans le processus récepteur
- C'est pas forcément les mêmes numéros de descripteur
- Mais c'est les mêmes fichiers ouverts

Paire de sockets

- `socketpair(2)` crée deux sockets connectés
- Ressemble fortement à `pipe(2)`
- Mais bidirectionnel !
- Messages possibles (pas seulement flux d'octets) !