

# Programmation

Pierre David  
pda@unistra.fr

Université de Strasbourg – Master CSMI

2023 – 2024

# Plan

**L'API des sockets**

**Mode connecté**

**Mode non connecté**

**Diagnostics**

**Fonctions utilitaires**

**Intégration**

# Licence d'utilisation

©Pierre David

Disponible sur <https://gitlab.com/pdagog/ens>

Ces transparents de cours sont placés sous licence « Creative Commons Attribution – Pas d'Utilisation Commerciale 4.0 International »

Pour accéder à une copie de cette licence, merci de vous rendre à l'adresse <https://creativecommons.org/licenses/by-nc/4.0/>



# Plan

## L'API des sockets

Mode connecté

Mode non connecté

Diagnostics

Fonctions utilitaires

Intégration

# Présentation

1981 : financement de l'Université de Berkeley par la DARPA  
⇒ intégration des protocoles IP dans le noyau Unix

Initialement : 2 implémentations des protocoles IP

- ▶ Berkeley : adaptée aux réseaux locaux
- ▶ Bolt, Beranek et Newman (BBN) : adaptée aux réseaux longue distance

Choix de la DARPA : implémentation BBN avec interface de programmation Berkeley.

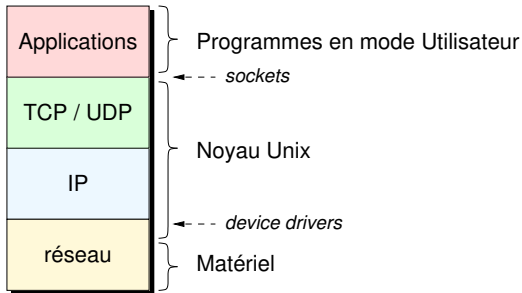
# Présentation

Interface de programmation bien intégrée au système :

- ▶ développement facile de nouvelles applications
- ▶ émergence de nouveaux services au cours du temps  
(Archie, Gopher, WAIS, WWW, NFS, P2P, etc.)

# Présentation – sockets

Interface de programmation = *socket*



# Présentation – sockets

Les sockets sont apparues en 1982 avec BSD 4.1

- ▶ Les sockets sont un des mécanismes de communication inter-processus disponibles sur Unix
- ▶ Les sockets ne sont pas liées à TCP/IP, et peuvent utiliser d'autres protocoles (AppleTalk, Xerox XNS, etc.)

Autres interfaces concurrentes (*Streams*, *XTI*, etc.) : obsolètes



# Présentation – sockets

Objectif des sockets : préserver la sémantique des opérations sur les fichiers (`open`, `read`, `write`...).

Problème : certains mécanismes rendent nécessaires l'ajout d'appels systèmes complémentaires

# Présentation – sockets

Utilisation des sockets :

1. création de la socket : `socket`
2. établissement de la communication : `bind`, `connect` ou `listen/accept`
3. échange de données : `read` et `write` (socket = descripteur de fichier) ou `sendto` et `recvfrom`
4. fermeture de la communication : `close` ou `shutdown`

L'enchaînement des opérations dépend du mode de connexion :

- ▶ mode connecté (exemple : TCP)
- ▶ mode non connecté (exemple : UDP)

# Présentation – familles de protocoles

- ▶ les sockets ne sont pas liées à une famille de protocoles donnée (IP, XNS, etc.)
- ▶ il existe certaines caractéristiques propres à chaque famille de protocoles
- ▶ une socket est donc rattachée à une famille :

PF_INET	Protocoles IPv4
PF_INET6	Protocoles IPv6
PF_UNIX	Tubes nommés
PF_NS	Xerox NS
...	...

# Présentation – familles d'adresses

Les adresses sont propres à chaque famille :

AF_INET	adresse IPv4 + numéro de port TCP ou UDP
AF_INET6	adresse IPv6 + numéro de port TCP ou UDP
AF_UNIX	nom dans l'arborescence Unix
...	...

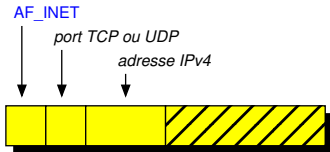
⇒ structure modèle pour les différentes familles :

```
struct sockaddr {  
    u_short  sa_family;  
    char     sa_data[14];  
};
```

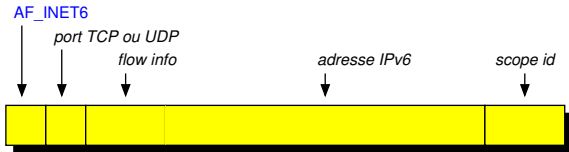
# Présentation – familles d'adresses



struct sockaddr (générique)



struct sockaddr\_in (IPv4)



struct sockaddr\_in6 (IPv6)

# Présentation

La structure du champ `sa_data` dépend de la famille de protocoles utilisée :

► Famille `PF_INET`

```
struct in_addr {  
    uint32_t s_addr;  
};  
  
struct sockaddr_in {  
    uint16_t      sin_family;           // AF_INET  
    uint16_t      sin_port;            // Port  
    struct in_addr sin_addr;           // Adresse IP  
    char          sin_zero[8];         // Bourrage  
};
```

# Présentation

## ► Famille PF\_INET6 (cf `netinet/in.h`)

```
struct in6_addr {  
    // 16 octets dans une union  
};  
  
struct sockaddr_in6 {  
    uint16_t      sin6_family;    // AF_INET6  
    uint16_t      sin6_port;      // Port  
    uint32_t      sin6_flowinfo;  // Flux  
    struct in6_addr sin6_addr;     // Adresse IPv6  
    uint32_t      sin6_scope_id;  
};
```

# Présentation

## ► Famille PF\_UNIX (cf `sys/un.h`)

```
struct sockaddr_un {  
    uint16_t  sun_family;           // AF_UNIX  
    char      sun_path[108];       // chemin  
};
```



# Plan

L'API des sockets

**Mode connecté**

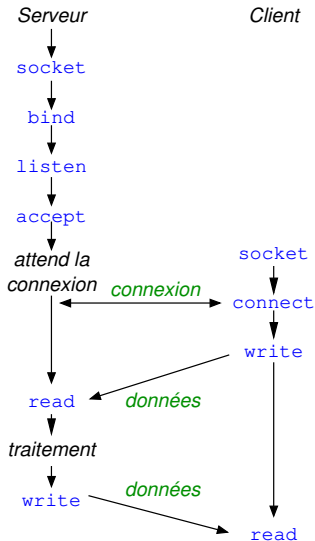
Mode non connecté

Diagnostics

Fonctions utilitaires

Intégration

# Mode connecté – Principe



# Mode connecté – socket

**Rôle** : crée la socket côté client et serveur

**Syntaxe** :

```
int socket (int famille, int type, int proto)
```

`socket` renvoie un descripteur de socket ( $\Leftrightarrow$  fichier) utilisable avec les autres primitives

► `type` = type de connexion :

type		famille	
		PF_UNIX	PF_INET/INET6
SOCK_STREAM	(connecté)	Oui	TCP
SOCK_DGRAM	(non connecté)	Oui	UDP
SOCK_RAW	(brut)		IP

► `protocole` = numéro du protocole (IP, TCP, UDP, etc.)  
(0  $\Rightarrow$  choix par le système)

# Mode connecté – bind

**Rôle** : attache une adresse (IP + port) à une socket

**Syntaxe** :

```
int bind (int s, struct sockaddr *adr, int lg)
```

Numéro de port = 0  $\Rightarrow$  choix laissé au système

# Mode connecté – listen

**Rôle** : place la socket en mode « ouverture passive » et définit la longueur de la file d'attente des connexions.

(ouverture passive  $\Rightarrow$  pour le serveur exclusivement)

**Syntaxe** :

```
int listen (int s, int longueur)
```

## Mode connecté – accept

**Rôle** : accepte une connexion en provenance d'un client  
(pour le serveur exclusivement)

**Syntaxe** :

```
int accept (int s, struct sockaddr *adr, int *lg)
```

Après `accept` :

- ▶ l'appel système retourne un nouveau descripteur de socket utilisé pour le dialogue avec le client
- ▶ `adr` : adresse du client connecté
- ▶ `lg` : longueur de l'adresse

# Mode connecté – connect

**Rôle** : connexion au serveur.  
(pour le client exclusivement)

**Syntaxe** :

```
int connect (int s, struct sockaddr *adr, int lg)
```

- ▶ `adr` : adresse de l'application (IP + port)
- ▶ `lg` : longueur de l'adresse

# Mode connecté – Serveur IPv4

```
main ()
{
    int s_cnx, s_dial, cli_len ;
    struct sockaddr_in serv_addr, cli_addr ;

    s_cnx = socket (PF_INET, SOCK_STREAM, 0) ;

    bzero ((char *) &serv_addr, sizeof serv_addr) ;
    serv_addr.sin_family = AF_INET ;
    serv_addr.sin_addr.s_addr = htonl (INADDR_ANY) ;
    serv_addr.sin_port = htons (5000) ;

    bind (s_cnx, &serv_addr, sizeof serv_addr) ;
    listen (s_cnx, 5) ;

    for (;;)
    {
        cli_len = sizeof cli_addr ;
        s_dial = accept (s_cnx, &cli_addr, &cli_len) ;
        serveur_tcp (s_dial) ;
        close (s_dial) ;
    }
}
```



# Mode connecté – Serveur IPv4

```
void serveur_tcp (int sock)
{
    while (read (sock, ...) > 0)
    {
        // traiter l'envoi et préparer la réponse
        write (sock, ...) ;
    }
}
```

## Mode connecté – Client IPv4

```
main ()
{
    int s_cli ;
    struct sockaddr_in serv_addr ;

    s_cli = socket (PF_INET, SOCK_STREAM, 0) ;

    bzero ((char *) &serv_addr, sizeof serv_addr) ;
    serv_addr.sin_family = AF_INET ;
    serv_addr.sin_addr.s_addr = htonl (0x824fc805) ; // 130.79.200.5
    serv_addr.sin_port = htons (5000) ;

    connect (s_cli, &serv_addr, sizeof serv_addr) ;

    client_tcp (s_cli) ;

    close (s_cli) ;
    exit (0) ;
}
```

# Mode connecté – Client IPv4

```
void client_tcp (int sock)
{
    while (...)
    {
        write (s_cli,...) ;
        // attendre la réponse
        read (s_cli,...) ;
    }
}
```

# Plan

L'API des sockets

Mode connecté

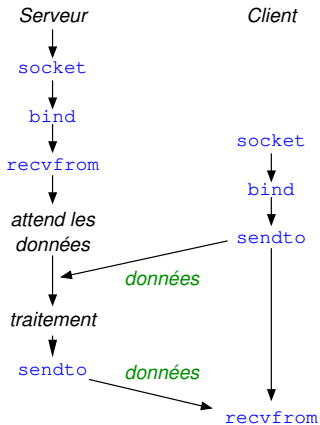
**Mode non connecté**

Diagnostics

Fonctions utilitaires

Intégration

# Mode non connecté – Principe



# Mode non connecté

1. le serveur et les clients créent chacun une socket (`SOCK_DGRAM`)
2. le serveur utilise `bind` avec un numéro de port unique
3. chaque client doit utiliser une adresse (IP + port) différente
4. le dialogue se fait par `sendto` et `rcvfrom`

# Mode non connecté – sendto

**Rôle** : envoyer un message en mode non connecté

**Syntaxe** :

```
sendto (int s, char *buf, int nb, int flags,  
        struct sockaddr *to, int len)
```

- ▶ `sendto` retourne le nombre de caractères envoyés
- ▶ `s`, `buf` et `nb` : idem `write`
- ▶ `flag` peut prendre les valeurs :

0	envoi messages normaux
MSG_OOB	envoi messages spéciaux
MSG_DONTROUTE	pas de routage

- ▶ `to` : adresse et port auxquels on envoie le message

# Mode non connecté – rcvfrom

**Rôle** : recevoir un message en mode non connecté.

**Syntaxe** :

```
rcvfrom (int s, char *buf, int nb, int flags,  
         struct sockaddr *from, int *len)
```

- ▶ `rcvfrom` retourne le nombre de caractères reçus
- ▶ `s`, `buf` et `nb` : idem `read`
- ▶ `flag` peut prendre les valeurs :

0	réception messages normaux
MSG_OOB	réception messages spéciaux
MSG_PEEK	regarder le message

- ▶ `from` : adresse et port d'où vient le message



## Mode non connecté – Serveur IPv4

```
main ()
{
    int sock ;
    struct sockaddr_in serv_addr ;

    sock = socket (PF_INET, SOCK_DGRAM, 0) ;

    bzero ((char *) &serv_addr, sizeof serv_addr) ;
    serv_addr.sin_family = AF_INET ;
    serv_addr.sin_addr.s_addr = htonl (INADDR_ANY) ;
    serv_addr.sin_port = htons (5000) ;

    bind (sock, &serv_addr, sizeof serv_addr) ;

    for (;;)
        serveur_udp (sock) ;
}
```

## Mode non connecté – Serveur IPv4

```
#define MAXMSG 1024

void serveur_udp (int sock)
{
    struct sockaddr_in cli_addr ;
    int n, cli_len ;
    char buf [MAXMSG] ;

    cli_len = sizeof cli_addr ;
    n = recvfrom (sock, buf, MAXMSG, 0, &cli_addr, &cli_len) ;
    sendto (sock, buf, n, 0, &cli_addr, cli_len) ;
}
```

## Mode non connecté – Client IPv4

```
main ()
{
    int sock ;
    struct sockaddr_in serv_addr, cli_addr ;

    sock = socket (PF_INET, SOCK_DGRAM, 0) ;

    bzero ((char *) &cli_addr, sizeof cli_addr) ;
    cli_addr.sin_family = AF_INET ;
    cli_addr.sin_addr.s_addr = htonl (INADDR_ANY) ;
    bind (sock, &cli_addr, sizeof cli_addr) ;

    bzero ((char *) &serv_addr, sizeof serv_addr) ;
    serv_addr.sin_family = AF_INET ;
    serv_addr.sin_addr.s_addr = htonl (0x824fc805) ; // 130.79.200.5
    serv_addr.sin_port = htons (5000) ;

    client_udp (sock, &serv_addr) ;

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

## Mode non connecté – Client IPv4

```
#define MAXMSG 1024

void client_udp (int sock, struct sockaddr_in *serv_addr)
{
    char buf [MAXMSG] ;
    int n ;

    sendto (sock, buf, n, 0, serv_addr, sizeof *serv_addr) ;
    n = recvfrom (sock, buf, MAXMSG, 0, (struct sockaddr *) 0,
                  (int *) 0) ;
}
```

# Plan

L'API des sockets

Mode connecté

Mode non connecté

**Diagnostics**

Fonctions utilitaires

Intégration

# Diagnostics

En cas d'erreur, les primitives systèmes renvoient -1 en code de retour, et un numéro d'erreur dans la variable globale `errno`  
De nouvelles définitions sont nécessaires pour gérer les protocoles IP.

# Diagnostics

EADDRINUSE	adresse déjà utilisée
EADDRNOTAVAIL	l'adresse ne peut être affectée
EAFNOSUPPORT	famille d'adresses non supportée
ECONNABORTED	connexion rompue
ECONNREFUSED	connexion refusée
ECONNRESET	connexion rompue par l'autre extrémité
EDESTADDRREQ	adresse de destination requise
EHOSTDOWN	machine ne répondant pas
EHOSTUNREACH	aucune route trouvée
EINPROGRESS	opération en cours
EISCONN	socket déjà connectée
ENET	erreur du logiciel ou du matériel réseau
ENETDOWN	réseau hors service
ENETRESET	connexion coupée par le réseau
ENETUNREACH	pas de route vers le réseau
ENOPROTOPT	protocole non disponible
ENOTCONN	socket non connectée
ENOTSOCK	opération sur une socket
EPROTONOSUPPORT	protocole non supporté
EPROTOTYPE	mauvais type pour la socket
ESHUTDOWN	transmission après un shutdown
ESOCKTNOSUPPORT	type de socket non supporté
ETIMEDOUT	temps d'attente dépassé

# Plan

L'API des sockets

Mode connecté

Mode non connecté

Diagnostics

**Fonctions utilitaires**

Intégration



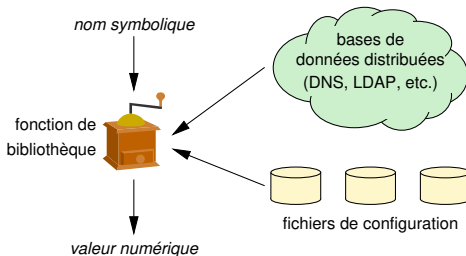
# Présentation

Les primitives systèmes utilisent :

- ▶ des adresses IPv4 sur 32 ou IPv6 sur 128 bits
- ▶ des numéros de port sur 16 bits
- ▶ des numéros de protocole

# Présentation

Utiliser des noms plutôt que des valeurs numériques ?



- ▶ machines : `www.unistra.fr`, `mailhost.u-strasbg.fr`...
- ▶ ports : `smtp/tcp`, `domain/udp`...
- ▶ protocoles : IP, TCP...

# Numéros de port

Fichier `/etc/services` : traduction nom  $\leftrightarrow$  numéro de port

Format :

`service port/protocole [synonymes...]`

<code>service</code>	nom officiel du service
<code>port/protocole</code>	numéro de port et protocole
<code>synonymes</code>	liste de synonymes

Exemple :

```
ftp      21/tcp
telnet   23/tcp
smtp     25/tcp  mail
domain   53/tcp  nameserver
domain   53/udp  nameserver
tftp     69/udp
```

# Numéros de port – getservbyname

**Rôle** : obtenir le numéro d'un port à partir de son nom

**Syntaxe** :

```
struct servent *getservbyname (char *nom, char *prot)
```

La structure `servent` est définie comme :

```
struct  servent {  
    char *s_name;           // nom officiel  
    char **s_aliases;       // liste de synonymes  
    int  s_port;            // numéro de port  
    char *s_proto;          // protocole  
}
```

Si `proto` est spécifié, le protocole doit être défini dans le fichier `/etc/protocols`

# Numéros de protocole

Fichier `/etc/protocols` : traduction nom  $\leftrightarrow$  numéro de protocole

Format :

`protocole numéro [synonymes...]`

<code>protocole</code>	nom officiel du protocole
<code>numéro</code>	numéro de protocole
<code>synonymes</code>	liste de synonymes

Exemple :

<code>ip</code>	<code>0</code>	<code>IP</code>
<code>icmp</code>	<code>1</code>	<code>ICMP</code>
<code>tcp</code>	<code>6</code>	<code>TCP</code>
<code>udp</code>	<code>17</code>	<code>UDP</code>

# Numéros de port – getprotobyname

**Rôle** : obtenir le numéro du protocole à partir de son nom.

**Syntaxe** :

```
struct protoent *getprotobyname (char *nom)
```

La structure `protoent` est définie comme :

```
struct  protoent {  
    char *p_name;           // nom officiel  
    char **p_aliases;       // liste des synonymes  
    int  p_proto;           // numéro de protocole  
}
```

# Adresses IP – /etc/hosts

Fichier `/etc/hosts` : utilisé quand le serveur de noms n'est pas actif (ou en complément)

Format :

`adresse nom-officiel [synonymes...]`

<code>adresse</code>	adresse IPv4 ou IPv6
<code>nom-officiel</code>	nom de la machine
<code>synonymes</code>	synonymes facultatifs

Exemple :

```
130.79.201.195    www.unistra.fr    www
130.79.200.5      ftp.u-strasbg.fr  ftp
2001:660:2402::6  ftp.u-strasbg.fr  ftp
```

# Adresses IP – DNS

Fichier `/etc/resolv.conf` : utilisé pour identifier le serveur de noms mandataire et le ou les critères de *domain completion*

Exemple :

```
search u-strasbg.fr
nameserver 130.79.200.200
nameserver 2001:660:2402::200
```



# Informations de connexion – getaddrinfo

**Rôle** obtenir les informations de connexion (adresses IP et numéro de port)

**Syntaxe :**

```
int getaddrinfo (char *nom, char *service,  
                struct addrinfo *indic, struct addrinfo **res0)
```

La structure `addrinfo` est définie comme :

```
struct addrinfo  
{  
    int ai_flags;           // pour le paramètre indic  
    int ai_family;         // famille d'adresse pour socket  
    int ai_socktype;       // SOCK_STREAM ou SOCK_DGRAM  
    int ai_protocol;       // protocole trouvé  
    socklen_t ai_addrlen;  // taille de l'adresse trouvée  
    struct sockaddr *ai_addr; // adresse trouvée  
    char *ai_canonname;     // nom canonique trouvé  
    struct addrinfo *ai_next; // suivant dans la liste  
};
```

# Informations de connexion – getaddrinfo

Utilisation par un client pour se connecter à un serveur :

```
struct addrinfo indic, *res0 ;
```

ai_flags	0
ai_family	AF_UNSPEC
ai_socktype	SOCK_STREAM
ai_protocol	0
ai_addr	NULL
ai_canonname	NULL
ai_next	NULL

```
getaddrinfo ("ftp.u-strasbg.fr", "ftp", &indic, &res0) ;
```

ai_flags	0
ai_family	AF_INET6
ai_socktype	SOCK_STREAM
ai_protocol	IPPROTO_TCP
ai_addr	2001:660:2402::6 (21)
ai_canonname	NULL
ai_next	_____

ai_flags	0
ai_family	AF_INET
ai_socktype	SOCK_STREAM
ai_protocol	IPPROTO_TCP
ai_addr	130.79.200.5 (21)
ai_canonname	NULL
ai_next	NULL

## Informations de connexion – getaddrinfo

Code correspondant :

- ▶ le client doit parcourir la liste pointée par `res0` et tenter une connexion sur chaque adresse
- ▶ il faut s'arrêter à la première connexion réussie
- ▶ ne pas oublier d'utiliser `freeaddrinfo` pour désallouer la liste

## Informations de connexion – getaddrinfo

```
int s = -1, r ; char *cause ;
struct addrinfo indic, *res, *res0 ;


memset (&indic, 0, sizeof indic) ;
indic.ai_family = PF_UNSPEC ;
indic.ai_socktype = SOCK_STREAM ;
if (getaddrinfo (host, serv, &indic, &res0) != 0) {
    fprintf (stderr, "getaddrinfo:_%s\n", gai_strerror (r)) ;
    exit (1) ;
}
for (res = res0 ; res != NULL ; res = res->ai_next) {
    s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    if (s == -1) cause = "socket" ;
    else {
        if (connect (s, res->ai_addr, res->ai_addrlen) == -1) {
            cause = "connect" ;
            close (s) ;
            s = -1 ;
        }
        else break ;
    }
}
if (s == -1) perror (cause) ;
freeaddrinfo (res0) ;
```

# Informations de connexion – getaddrinfo

Utilisation par un serveur pour ouvrir la socket :

```
struct addrinfo indic, *res0 ;
```

ai_flags	AI_PASSIVE
ai_family	AF_UNSPEC
ai_socktype	SOCK_STREAM
ai_protocol	0
ai_addr	NULL
ai_canonname	NULL
ai_next	NULL



```
getaddrinfo (NULL, "ftp", &indic, &res0) ;
```



ai_flags	AI_PASSIVE
ai_family	AF_INET6
ai_socktype	SOCK_STREAM
ai_protocol	IPPROTO_TCP
ai_addr	:: (21)
ai_canonname	NULL
ai_next	_____

ai_flags	AI_PASSIVE
ai_family	AF_INET
ai_socktype	SOCK_STREAM
ai_protocol	IPPROTO_TCP
ai_addr	0.0.0.0 (21)
ai_canonname	NULL
ai_next	NULL

## Informations de connexion – getaddrinfo

Code correspondant :

- ▶ le serveur doit créer une nouvelle socket de connexion passive pour chaque élément de la liste
- ▶ le serveur doit détecter une tentative sur n'importe laquelle des sockets  
⇒ primitive système `select`
- ▶ ne pas oublier d'utiliser `freeaddrinfo` pour désallouer la liste après avoir créé les sockets

## Informations de connexion – getaddrinfo

```
memset (&indic, 0, sizeof indic) ;
indic.ai_family = PF_UNSPEC ;
indic.ai_socktype = SOCK_STREAM ;
indic.ai_flags = AI_PASSIVE ;
if ((r = getaddrinfo (NULL, serv, &indic, &res0)) != 0) {
    fprintf (stderr, "getaddrinfo:_%s\n", gai_strerror (r)) ;
    exit (1) ;
}

ns = 0 ;
for (res = res0; res && ns < MAXSOCK; res = res->ai_next) {
    s[ns]=socket(res->ai_family,res->ai_socktype,res->ai_protocol);
    if (s [ns] == -1) cause = "socket" ;
    else {
        setsockopt(s[ns],SOL_SOCKET,SO_REUSEADDR,&opt,sizeof opt) ;
        r = bind (s [ns], res->ai_addr, res->ai_addrlen) ;
        if (r == -1) {
            cause = "bind" ;
            close (s [ns]) ;
        }
        else {
            listen (s [ns], 5) ;
            ns++ ;
        }
    }
}
```

# Informations de connexion – getaddrinfo

```
for (;;) {
    fd_set readfds ;
    int i, max = 0 ;

    FD_ZERO (&readfds) ;
    for (i = 0 ; i < ns ; i++) {
        FD_SET (s [i], &readfds) ;
        if (s [i] > max) max = s [i] ;
    }
    if (select (max+1, &readfds, NULL, NULL, NULL) == -1)
        raler_log ("select") ;

    for (i = 0 ; i < ns ; i++) {
        struct sockaddr_storage sonadr ;
        socklen_t salong ;
        if (FD_ISSET (i, &readfds)) {
            salong = sizeof sonadr ;
            sd = accept (s[i],(struct sockaddr *)&sonadr,&salong) ;
            if (fork () == 0) {
                serveur (sd) ;
                exit (0) ;
            }
            close (sd) ;
        }
    }
}
```



# Macros de test

```
int IN6_IS_ADDR_UNSPECIFIED (struct in6_addr *) ;  
int IN6_IS_ADDR_LOOPBACK   (struct in6_addr *) ;  
int IN6_IS_ADDR_MULTICAST  (struct in6_addr *) ;  
int IN6_IS_ADDR_V4COMPAT   (struct in6_addr *) ;  
int IN6_IS_ADDR_V4COMPAT   (struct in6_addr *) ;  
...
```

# Plan

L'API des sockets

Mode connecté

Mode non connecté

Diagnostics

Fonctions utilitaires

**Intégration**

# Démons

Les serveurs sont réalisés sous Unix grâce à des *démons*

Processus utilisateurs ordinaires, mais :

- ▶ fonctionnement en arrière plan (non lié à un terminal)
- ▶ souvent avec les droits du super-utilisateur
- ▶ lancés en général au démarrage du système
- ▶ ne s'arrêtent jamais

# Démons

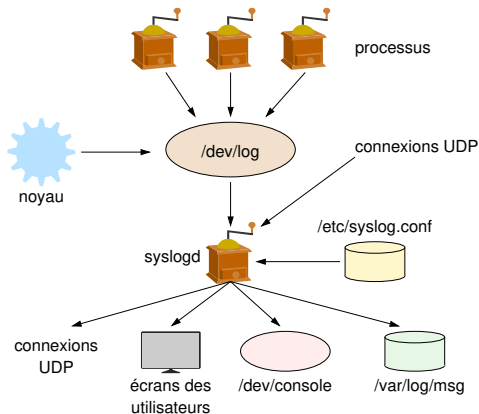
Les démons doivent suivre certaines règles de conception pour répondre aux contraintes :

1. faire un `fork`, terminer le père et continuer dans le fils  
⇒ fils « orphelin » ⇒ nouveau père = processus 1 (`init`)
2. appeler `setsid`  
⇒ créer une nouvelle session, non rattachée à un terminal
3. changer le répertoire courant pour un répertoire qui existe toujours (p. ex. : `/`)
4. modifier le masque de création des fichiers (`umask = 0`)  
⇒ expliciter les permissions des fichiers créés
5. fermer les descripteurs de fichiers inutiles

# Démons

Problème : plus de terminal pour les messages

Solution : envoi des messages par le démon **syslogd**



# Démons

```
main ()
{
    int pid ;

    pid = fork () ;
    switch (pid)
    {
        case -1 :           // erreur de fork
            exit (1) ;
        case 0 :           // le démon proprement dit
            setsid () ;     // session propre au démon
            chdir ("/") ;   // répertoire courant
            umask (0) ;
            close (0) ; close (1) ; close (2) ;
            openlog ("exemple", LOG_PID, LOG_DAEMON) ;
            /* ... */
            syslog (LOG_WARN, "attention_(%f)", 3.14) ;
            /* ... */
        default :          // terminaison du père
            exit (0) ;
    }
}
```

# Internet Super Server

## Problèmes :

- ▶ chaque démon doit avoir une séquence de code quasiment toujours identique pour accepter les connexions
- ▶ les démons sont des processus qui utilisent des ressources même s'ils sont inactifs

## Solution :

- ▶ remplacer tous les démons réseau correspondant aux divers services par un seul
- ▶  $\Rightarrow$  diminuer le nombre de processus
- ▶  $\Rightarrow$  simplifier l'écriture des démons
- ▶ implémentation « historique » : `inetd`
- ▶ autres implémentations : `xinetd`, `launchd`, `systemd`, etc.

# Internet Super Server – inetd

Implémentation :

- ▶ Le fichier `/etc/inetd.conf` liste les démons gérés par `inetd`
- ▶ `inetd` utilise `bind/listen/connect` pour chaque service
- ▶ Après une connexion d'un client à un de ces services, `accept` et `exec` du serveur



# Internet Super Server – inetd

Le fichier `inetd.conf` contient pour chaque service :

nom du service	présent dans <code>/etc/services</code>
type	<code>stream</code> ou <code>dgram</code>
protocole	dans <code>/etc/protocols</code>
<code>wait / nowait</code>	<code>wait</code> (ou non) en mode stream
user	en général : root
nom du serveur	nom complet
arguments	au plus 5

Exemple :

```
ftp      stream tcp nowait root /etc/ftpd      ftpd -l
telnet   stream tcp nowait root /etc/telnetd  telnetd
ntalk    dgram  udp wait   root /etc/ntalkd  ntalkd
```

# Internet Super Server

Un serveur (exemple pour TCP) avec `inetd` est réduit à :

```
main ()
{
    serveur_tcp (0, 1) ;           // entrée et sortie standard
}

void serveur_tcp (int sock_in, int sock_out)
{
    while (read (sock_in, ...) > 0)
    {
        // traiter l'envoi et préparer la réponse
        write (sock_out, ...) ;
    }
}
```

# Internet Super Server

Aujourd'hui, l'Internet Super Server (`inetd` ou équivalent) n'est plus réellement utilisé :

- ▶ `inetd` provoque un appel à `fork` et à `exec` à chaque connexion  
⇒ peu efficace pour des connexions de courte durée (ex : HTTP)
  - ▶ la plupart des services conçus avec `inetd` sont obsolètes car :
    - ▶ services non authentifiés pour la plupart
    - ▶ services sans chiffrement
    - ▶ volonté de réduire la surface d'attaque
  - ▶ la tendance est de faire des *micro-services*
    - ▶ services élémentaires
    - ▶ durée de connexion très courte
    - ▶ utilisant HTTP comme protocole support
- ⇒ utilisation de serveurs spécialisés dans ce type de traitement