

Application Programmation Interface

Yves Legrandg rard

ylg@pps.jussieu.fr

Sockets (1)

(introduction)

- The most widespread API is the socket interface
- Sockets constitute a generic device allowing communications between processes not residing on the same host
- A socket can be seen like one of the ends of a communication device. Under UNIX, one or more processes can read or write in this one via a I/O descriptor provided by the socket

Sockets (2)

(socket's domain)

- A socket, when it wants to communicate with another socket, uses a logical address having a specific format. Such a whole of same addresses format is called a domain (warning: domain \neq protocol)
- Two important domains:
 - internet domain IPv4 (AF_INET)
 - internet domain IPv6 (AF_INET6)

Sockets (3)

(socket's domain)

- There are also many of other domains:
 - UNIX domain (AF_UNIX or AF_LOCAL). This last has only one historical interest and allows only the communication between sockets residing on the same host
 - ATM domain (AF_ATM)
 - ISO domain (AF_ISO), ISO protocols
 - etc...

Sockets (4)

(socket's type)

- Each socket has a type which determines its functionalities. There are 3 principal types:
 - Type `SOCK_DGRAM`: unreliable communication in non connected mode. In the Internet domain, it is protocol UDP
 - Type `SOCK_STREAM`: reliable communication in connected mode. In the Internet domain, it is protocol TCP

Sockets (5)

(socket's type)

- Type `SOCK_RAW`: allows access to the protocols of lower level like IP and ICMP. Under UNIX, this type is accessible only to the “super-user” and is used in particular by developers
- Other types exist but are used little. For example type `SOCK_SEQPACKET`, similar to the type `SOCK_STREAM`, but which meets only in the domain Xerox (`AF_NS`)

Sockets (6)

(structures corresponding to the various domains: UNIX BSD 4.4)

```
/* generic structure */  
/* /usr/include/sys/socket.h */  
  
struct sockaddr {  
    u_char          sa_len;      /* total length */  
    sa_family_t     sa_family; /* domain */  
    char            sa_data[14]; /* address */  
};
```

Sockets (7)

(structures corresponding to the various domains: UNIX BSD 4.4)

```
/* IPv4 structure */  
/* /usr/include/netinet/in.h */  
struct sockaddr_in {  
    u_char    sin_len;           /* total length */  
    u_char    sin_family;       /* AF_INET */  
    u_short   sin_port;         /* port number */  
    struct    in_addr sin_addr; /* IPv4 @ */  
    char      sin_zero[8];  
};
```

Sockets (8)

(structures corresponding to the various domains: UNIX BSD 4.4)

```
/* IPv6 structure */  
/* /usr/include/netinet/in.h */  
struct sockaddr_in6 {  
    u_int8_t sin6_len;          /* total length */  
    u_int8_t sin6_family;      /* AF_INET6 */  
    u_int16_t sin6_port;       /* port number */  
    u_int32_t sin6_flowinfo;   /* flow id */  
    struct in6_addr sin6_addr; /* IPv6 @ */  
    u_int32_t sin6_scope_id;    /* scope */  
};
```

Sockets (9)

(structures corresponding to the various domains: UNIX BSD 4.4)

```
/* IPv4 address (/usr/include/netinet/in.h) */  
struct in_addr {  
    u_int32_t    s_addr;        /* 32 bits */  
}; /* It's a structure for historical reasons */
```

```
/* IPv6 address (/usr/include/netinet/in.h) */  
struct in6_addr {  
    u_int8_t    s6_addr[16];   /* 128 bits */  
};
```

Sockets (10)

- The size of the structure *sockaddr_in6* is higher than the size of the generic structure *sockaddr*
- The structure *sockaddr_storage* is of sufficient size to take into account all the protocols while managing the problems of alignment. An example of use:

```
struct sockaddr_storage ss;  
struct sockaddr_in *s = (struct sockaddr_in *) &ss;  
struct sockaddr_in6 *s6 = (struct sockaddr_in6 *) &ss;
```

socket system call (1)

- It is the primitive which allows the creation of a socket. It has 3 arguments and returns either the I/O descriptor associated with the socket, or - 1 in the event of error

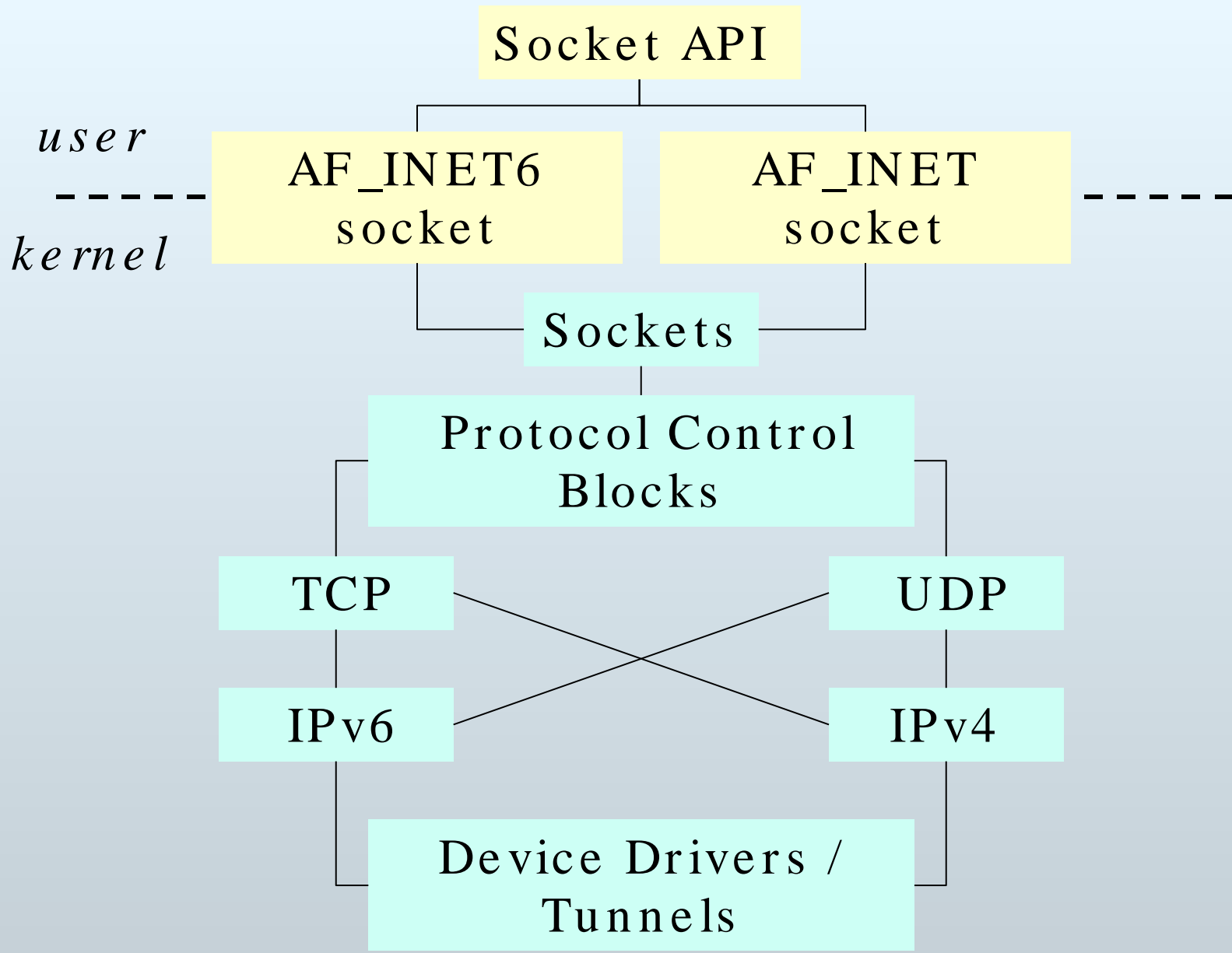
```
int socket (int domain, int type, int protocol);
```

- The last argument is the protocol chosen by the user. When it is 0, the default protocol suggested by the system is selected (usual case)

socket system call (2)

- An example: protocol TCP in Internet domain (IPv6)

```
int sock;  
sock = socket (AF_INET6, SOCK_STREAM, 0);  
if (sock == -1)  
    perror ("socket");  
else  
    ...
```



IPv4 / IPv6 Dual Stack

DNS primitives (1)

- They are the primitives (dual one of the other) *getaddrinfo* and *getnameinfo*. They are independent of the family of addresses and standardized by the IETF
- They replace the old primitives:
 - *gethostbyname*, *gethostbyaddr*
 - *getservbyname*, *getservbyport*
- They make it possible in particular to take into account the new family of addresses AF_INET6, which did not allow to do, for example, the primitive *gethostbyname*

DNS primitives (2)

```
int getaddrinfo (const char *nodename,  
                const char *servname,  
                const struct addrinfo *hints,  
                struct addrinfo **res);
```

```
void freeaddrinfo (struct addrinfo *res);
```

```
const char *gai_strerror(int errcode);
```

DNS primitives (3)

```
struct addrinfo {  
    int ai_flags; /* AI_PASSIVE, AI_CANONNAME, ... */  
    int ai_family; /* AF_XXX */  
    int ai_socktype; /* SOCK_XXX */  
    int ai_protocol; /* 0 or IPPROTO_XXX */  
    size_t ai_addrlen; /* address size (ai_addr) */  
    char *ai_canonname; /* fully qualified domain name */  
    struct sockaddr *ai_addr; /* binary address */  
    struct addrinfo *ai_next; /* pointer to next structure */  
};
```

DNS primitives (4)

(an example of use of *getaddrinfo*)

```
int ret;  
char *host;  
struct addrinfo *res, *ptr;  
struct addrinfo hints = {  
    AI_CANONNAME, /* FQDN */  
    AF_UNSPEC, /* IPv4 and IPv6 */  
    SOCK_STREAM, /* TCP only */  
    0, 0, NULL, NULL, NULL /* must be 0 */  
};                                /* or NULL */
```

DNS primitives (5)

(an example of use of *getaddrinfo*)

```
ret = getaddrinfo (host, NULL, &hints, &res);
if (ret) {
    fprintf (stderr, "%s\n", gai_strerror(ret));
    /* more error processing here */ }
for (ptr = res; ptr; ptr = ptr->ai_next)
    switch(ptr->ai_family) {
        case AF_INET:          /* IPv4 address */
        case AF_INET6:       /* IPv6 address */
    }
freeaddrinfo(res);
```

DNS primitives (6)

- The dual primitive `getnameinfo` carries out the translation of an address towards a name

```
int getnameinfo (const struct sockaddr *sa,  
                 socklen_t salen,  
                 char *host, size_t hostlen,  
                 char *serv, size_t servlen,  
                 int flags);
```

- The field `flags` makes it possible to modify the answer (bits:
`NI_NUMERICHOST`, `NI_NUMERICSERV`,
`NI_NAMEREQD`, `NI_DGRAM`)

Address conversion primitives (1)

- The function *inet_pton* converts a textual address into its binary format. It returns *1* in the event of success, *0* if the provided textual address is not valid and *-1* if the address family is not recognized (AF_INET and AF_INET6 only)

```
int inet_pton(int af, /* AF_INET or AF_INET6 */  
              const char *src, /* address to convert */  
              void *dst); /* address converted */
```

Address conversion primitives (2)

- The function `inet_ntop` converts a binary address into its textual form. It turns over the null pointer in the event of failure (address family not supported or insufficient buffer size)

```
char *inet_ntop(int af, /* AF_INET ou AF_INET6 */  
                const void *src, /* address to convert */  
                char *dst, /* address converted */  
                size_t size); /* buffer size */
```

Address conversion primitives (3)

```
char dst[INET6_ADDRSTRLEN];  
/* the constant INET6_ADDRSTRLEN is defined in  
* netinet/in.h and is equal to 46. For IPv4  
* addresses, a constant INET_ADDRSTRLEN  
* is also defined and setted to 16 */  
  
struct in6_addr *src;  
if (!inet_ntop (AF_INET6, (const void *) src, dst,  
    sizeof(dst)) {  
    /* error processing */  
}
```

Client / Server, connected mode (1)

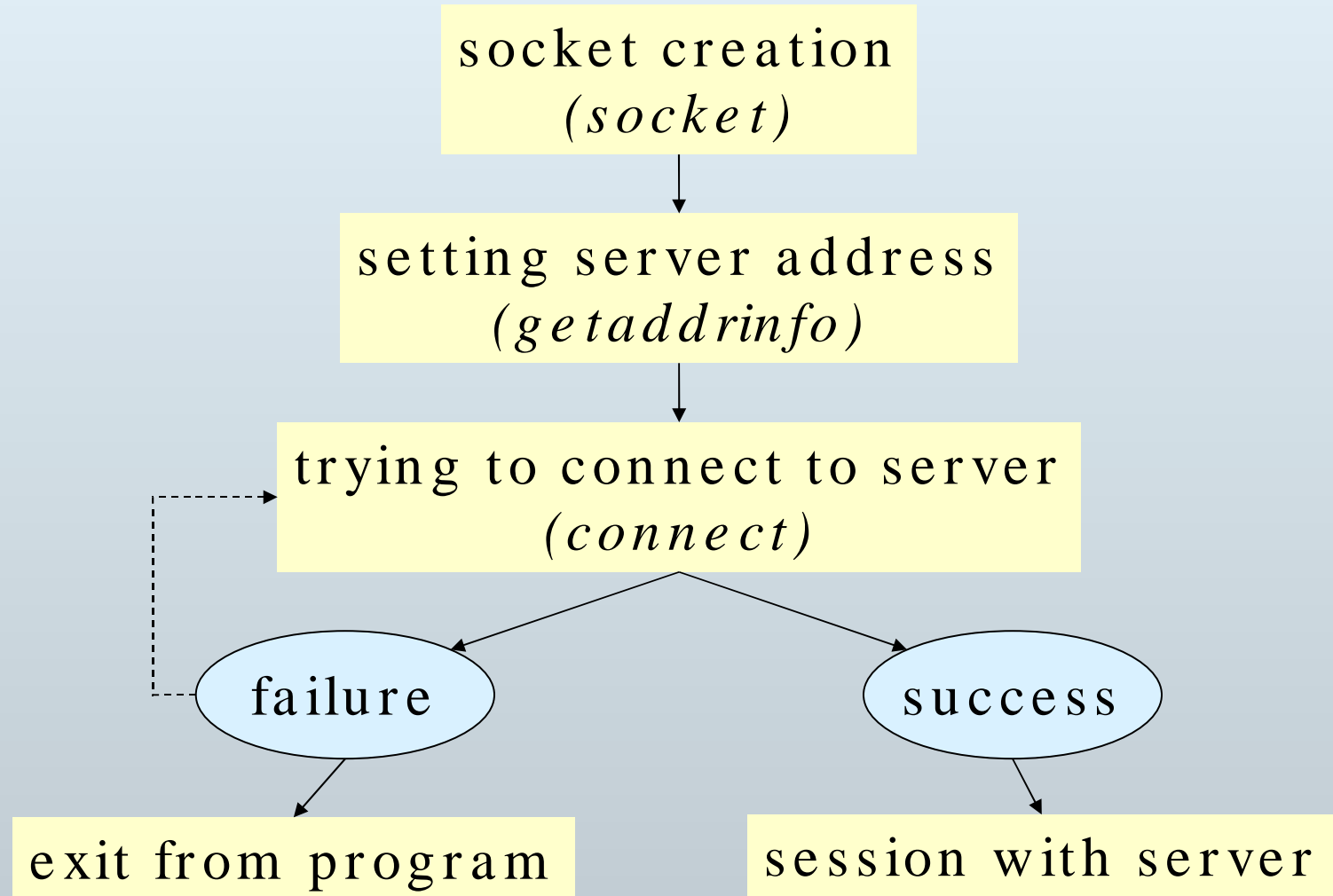
(the client side)

- In connected mode, the client creates a socket, then built the address of the server and finally establishes a connection with this server (virtual circuit) using the primitive *connect*

```
int connect (int sock, const struct sockaddr *sa,  
            socklen_t length);
```

- *connect* returns *-1* in the event of error *0* if not
- In Internet domain, the corresponding protocol connected mode is of course TCP

Client / Server, connected mode (2) (the client side)



Client / Server, connected mode (3) (the client side, source code)

```
# include <stdio.h>
# include <unistd.h>
# include <sys/socket.h>
# include <netdb.h>

int open_conn (char *host, char *serv)
{
    int sock, ecode;
    struct addrinfo *res;
    struct addrinfo *hints = {
        0, AF_UNSPEC, SOCK_STREAM,
        0, 0, NULL, NULL, NULL
    };
};
```

1. The function *open_conn* takes as its 1st argument the name of the server on which one will establish a TCP connection and as its 2nd argument the name of the service requested on this server
2. It returns the I/O descriptor associated with the socket carrying out connection or *-1* in the event of error

Client / Server, connected mode (4) (the client side, source code)

```
ecode = getaddrinfo (host, serv, &hints, &res);
if (ecode) {
    fprintf (stderr, "getaddrinfo :%s\n", gai_strerror (ecode));
    return -1;
}
sock = socket (res -> ai_family, res -> ai_socktype,
               res -> ai_protocol);
if (sock < 0) {
    freeaddrinfo (res);
    perror ("socket");
    return -1;
}
```

Client / Server, connected mode (5) (the client side, source code)

```
if (connect (sock, res -> ai_addr, res -> ai_addrlen) < 0) {  
    freeaddrinfo (res);  
    perror ("connect");  
    return -1;  
}  
freeaddrinfo (res);  
return sock;  
}
```

Client / Server, connected mode (6) (the server side)

- The client/server model is **asymmetrical**. Contrary to the client, the server is passive, it awaits the requests of the clients
- On UNIX, the server is a “daemon” process. With each request for connection, it creates a process which treats the request of the client
- At the time of its launching, the server creates a socket whose I/O descriptor is known only by processes having inherited from it (after a *fork*). So that other processes have access to this socket, this one is attached at an address. This attachment is carried out by the primitive *bind*

Client / Server, connected mode (7) (the server side)

```
int bind (int sock, const struct sockaddr *sa,  
          socklen_t length);
```

- When the port number provided to the primitive *bind* (via the parameter *sa*) is 0, it is then allocated dynamically by the system. One can get it with the primitive *getsockname*

```
int getsockname (int sock, struct sockaddr *sa,  
                 socklen_t *length);
```

Client / Server, connected mode (8) (the server side)

- Once the socket created and attached at an address, the server must inform the system he is ready to accept the requests for connections. It is the object of the primitive *listen*

```
int listen (int sock, int max);
```

- The second argument is the maximum number of requests for hanging connections. It is a constant defined in the file: */usr/include/sys/socket.h*

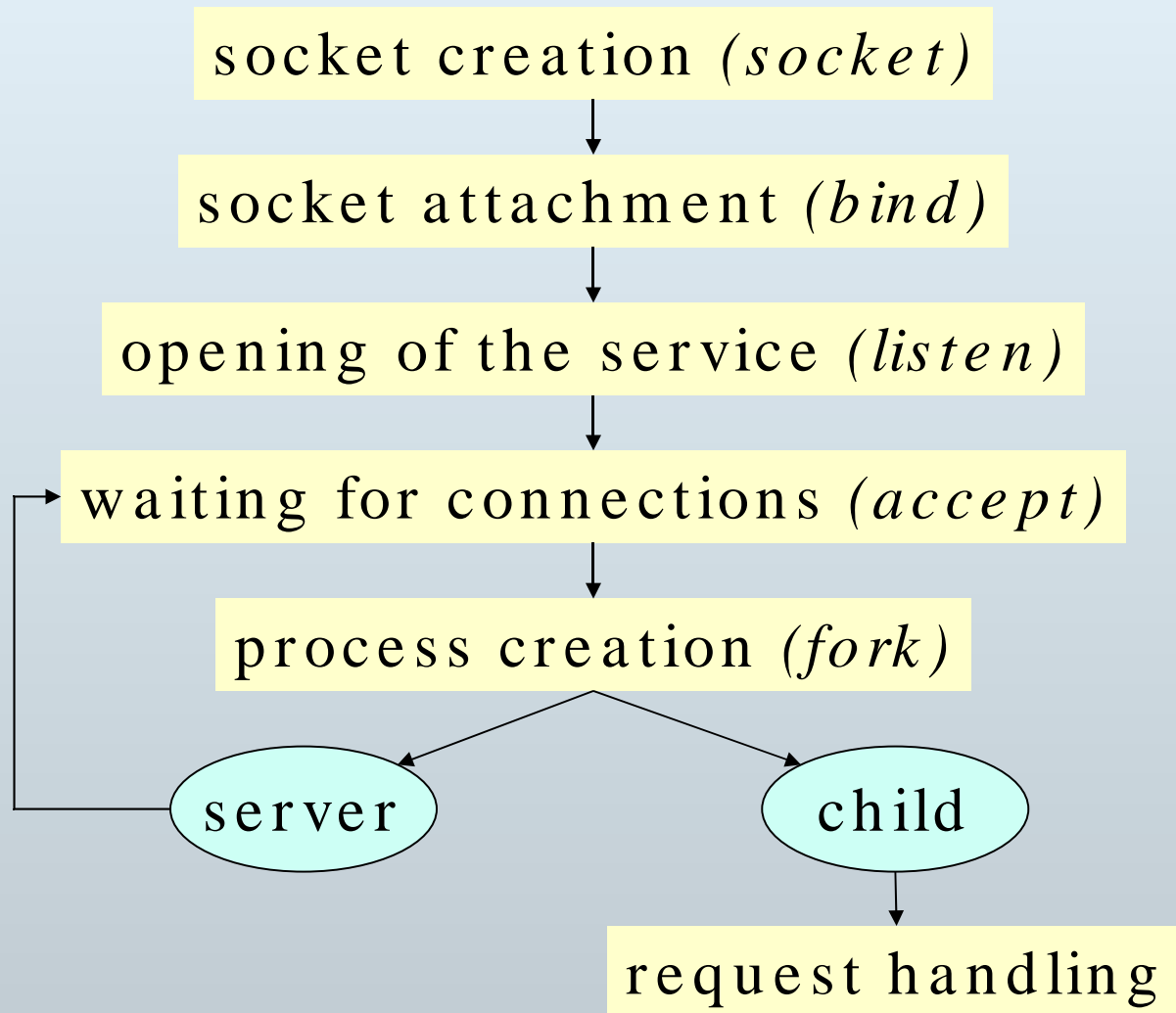
```
#define SOMAXCONN 128
```

Client / Server, connected mode (9) (the server side)

- Finally each hanging connection in the queue associated with the socket is extracted by the server and the virtual circuit with the socket of the client is established via the creation of a new socket.
- This is carried out by the primitive *accept* whose return value is the I/O descriptor of the socket lately created

```
int accept (int sock, struct sockaddr *sa,  
            socklen_t *length);
```

Client / Server, connected mode (10) (the server side)



Client / Server, connected mode (11) (the server side: source code)

```
# include <stdio.h>
# include <stdlib.h>
# include <unistd.h>
# include <errno.h>
# include <sys/socket.h>
# include <netdb.h>
# include <syslog.h>
# include <signal.h>

static void reap_child (int);
void serv_daemon (int family, char *serv,
                 void (*serv_func)(int),
                 char *serv_logname)
{
```

1. *serv_daemon* takes as 1st argument an address family: IPv4 or IPv6
2. The 2nd argument is the name of the service
3. The 3rd argument is a pointer towards the function of service
4. The last argument is the name passed to *syslogd* (it is the name which will appear in log files)

Client / Server, connected mode (12) (the server side: source code)

```
int sock, ecode;
struct addrinfo *res;
struct addrinfo *hints = {
    AI_PASSIVE, 0, SOCK_STREAM,
    0, 0, NULL, NULL, NULL
};
hints.ai_family = family;
ecode = getaddrinfo(NULL, serv, &hints, &res);
if (ecode) {
    fprintf(stderr, "getaddrinfo :%s\n", gai_strerror(ecode));
    exit(1);
}
sock = socket(res->ai_family, res->ai_socktype,
              res->ai_protocol);
```

Client / Server, connected mode (13) (the server side: source code)

```
if (sock < 0) {  
    freeaddrinfo (res);  
    perror ("socket");  
    exit(1);  
}  
if (bind (sock, res -> ai_addr, res -> ai_addrlen) < 0) {  
    freeaddrinfo (res);  
    perror ("bind");  
    exit(1);  
}  
freeaddrinfo (res);
```

Client / Server, connected mode (14) (the server side: source code)

```
# ifndef DEBUG
    if (daemon (0, 0) < 0) { /* becomes a «daemon» process */
        perror ("daemon");
        exit(1);
    }
# endif
    signal (SIGCHLD, reap_child); /* signal SIGCHLD is caught */
    listen (sock, SOMAXCONN);
    openlog (serv_logname, LOG_PID | LOG_CONS, LOG_USER);
    for (;;) {
        int a, f, len;
        struct sockaddr_storage from;

        a = accept (sock, (struct sockaddr *) &from, &len);
```

Client / Server, connected mode (15) (the server side: source code)

```
if (a < 0) {
    if (errno != EINTR)
        syslog (LOG_ERR, "%s: accept: %m ", serv);
    continue;
}
f = fork ();
if (f < 0) {
    syslog (LOG_ERR, "%s: fork: %m ", serv);
    continue;
}
if (f == 0) { /* child process */
    close(sock);
    serv_func(a);
    exit(0);
}
```

Client / Server, connected mode (16) (the server side: source code)

```
        close(a); /* server and child */  
    } /* for loop ends here */  
}  
  
static void reap_child (int sig)  
{  
    int status;  
  
    while (wait3 (&status, WNOHANG, NULL) > 0);  
}
```

«wildcard» address (1)

- A TCP server when binding its (listening) socket, normally leaves the system to compute the source address
- In IPv4, the field *sin_addr.s_addr* of the structure *sockaddr_in*, passed to the primitive *bind*, has for value the constant *INADDR_ANY* defined in the file *netinet/in.h*
- In IPv6, there are two ways of proceeding because of the rules of the C language on initializations and assignments of structures

«wildcard» address (2)

- 1st method:

```
struct in6_addr any_addr = IN6ADDR_ANY_INIT;
```

- 2nd method:

```
extern const struct in6_addr in6addr_any;  
struct sockaddr_in6 sin6;  
sin6.sin6_addr = in6addr_any;
```

- Warning: the following C code is invalid

```
struct sockaddr_in6 sin6;  
sin6.sin6_addr = IN6ADDR_ANY_INIT;
```

«wildcard» address (3)

- The constants *IN6ADDR_ANY_INIT* and *in6addr_any* are defined in the file *netinet/in.h*

```
struct addrinfo *res;  
struct addrinfo *hints = {  
    AI_PASSIVE, 0, SOCK_STREAM,  
    0, 0, NULL, NULL, NULL  
};  
hints.ai_family = family;  
ecode = getaddrinfo (NULL, serv, &hints, &res);
```

- Field *res->ai_addr* is worth *INADDR_ANY* or *IN6ADDR_ANY_INIT* according to whether *family* is worth *AF_INET* or *AF_INET6*

read / write system calls (1)

- The primitive *recv*, which should be used only with connected sockets, (i.e. of type *SOCK_STREAM*) is identical in its use to the primitive *read* except the presence of an additional argument

```
ssize_t recv (int s, void *buf, size_t len, int flags);
```

- The additional argument *flags* can have the following values:
MSG_OOB, *MSG_PEEK*, *MSG_WAITALL*

read / write system calls (2)

- The primitive *send*, which, like *recv*, should be used only with connected sockets, is identical in its use to the *write* primitive except the presence of an additional argument

```
ssize_t send (int s, const void *buf, size_t len, int flags);
```

- The additional argument *flags* can have the following values:
MSG_OOB, MSG_PEEK, MSG_DONTROUTE,
MSG_ERR, MSG_EOF

read / write system calls (3)

- With non-connected mode (the type of the socket is then: *SOCK_DGRAM*), one must use the primitives *recvfrom* and *sendto* because the destination (or source) address is not implicit any more, from where the presence of 2 additional arguments:

```
ssize_t recvfrom (int s, void *buf, size_t len, int flags,  
                 struct sockaddr *from,  
                 socklen_t *fromlen);
```

read / write system calls (4)

```
ssize_t sendto (int s, const void *buf, size_t len, int flags,  
                const struct sockaddr *to,  
                socklen_t tolen);
```

- The first four parameters of *recvfrom* (resp. *sendto*) is identical to those of *recv* (resp. *send*)
- These two primitives can be also used in connected mode. In particular, if the argument *from* of the primitive *recvfrom* is the null pointer, then *recvfrom* is equivalent to *recv*

Socket's options (1)

```
int getsockopt (int s, int level, int optname, void *optval,  
                socklen_t *optlen);
```

- For example, to get the default value of the “hop limit” of the (unicast) IPv6 datagrams:

```
int hoplimit;  
socklen_t len = sizeof (hoplimit);  
if (getsockopt(s, IPPROTO_IPV6, IPV6_UNICAST_HOPS,  
              (void *) &hoplimit, &len) < 0) {  
    /* error processing */  
}
```

Socket's options (2)

```
int setsockopt (int s, int level, int optname,  
               const void *optval, socklen_t optlen);
```

- For example to change the value of the “hop limit” of the (unicast) IPv6 datagrams sent via the socket whose I/O descriptor is *s*:

```
int hoplimit = 16;  
if (setsockopt (s, IPPROTO_IPV6, IPV6_UNICAST_HOPS,  
              (void *) &hoplimit, sizeof (hoplimit)) < 0) {  
    /* error processing */  
}
```

Client / Server: a full example (1)

```
$ nbus <hostname>
```

```
5 user(s) logged on <hostname>
```

- *nbus.c*: client source code
- *nbusd.c*: server source code
- *nbus.h*: common header to files *nbus.c* and *nbusd.c*

Client / Server: a full example (2)

```
/* nbus.h */  
# include <stdio.h>  
# include <unistd.h>  
  
# define SERVICE      "nbus"  
/* You must declare nbus in file /etc/services. For example,  
* insert the following line:  
* nbus      20000/tcp  
*/  
# define UTMP         "/var/run/utmp" /* FreeBSD/NetBSD */
```

Client / Server: a full example (3)

```
/* nbus.c (part 1/2) */  
# include "nbus"  
  
extern int open_conn(char*, char*);  
  
int main (int argc, char **argv);  
{  
    int fd;  
    short nbus; /* number of users logged on remote host */  
  
    if (argc != 2) {  
        fprintf (stderr, "%s: usage: %s <host>.\n ", argv[0], argv[0]);  
        exit(1);  
    }  
    if ((fd = open_conn (argv[1], SERVICE)) == -1)  
        exit(1);
```

Client / Server: a full example (4)

```
/* nbus.c (part 2/2) */
```

```
recv (fd, (void *) &nbus, sizeof (nbus), 0);
```

```
nbus = ntohs (nbus); /* host byte order */
```

```
if (nbus == -1) {
```

```
    fprintf (stderr, "Can't open %s on %s.\n", UTMP, argv[1]);
```

```
    exit(1);
```

```
}
```

```
fprintf (stdout, "%d user(s) logged on %s.\n", nbus, argv[1]);
```

```
exit(0);
```

```
}
```

Client / Server: a full example (5)

```
/* nbusd.c (part 1/1) */  
# include "nbus"  
# ifdef IPV6  
# define FAMILY      AF_INET6  
# else  
# define FAMILY      AF_INET  
# endif  
  
extern int serv_daemon (int, char *, void (*)(int), char *);  
void nbus(int);  
  
int main (void);  
{  
    serv_daemon (FAMILY, SERVICE, nbus, NULL);  
}
```

Client / Server: a full example (6)

```
/* nbusd.c (part 2/2) */
void nbus (int sock)
{
    int fd;
    short nu = -1;
    struct utmp ut;
    if ((fd = open (UTMP, O_RDONLY)) > 0) {
        nu = 0;
        while (read (fd, (char *) &ut, sizeof (ut)) == sizeof (ut))
            if (ut.ut_name[0])
                nu++;
    }
    close(fd);
    nu = htons (nu);
    send (sock, (void *) &nu, sizeof (nu), 0);
}
```