

CHAPTER IX: SOCKETS

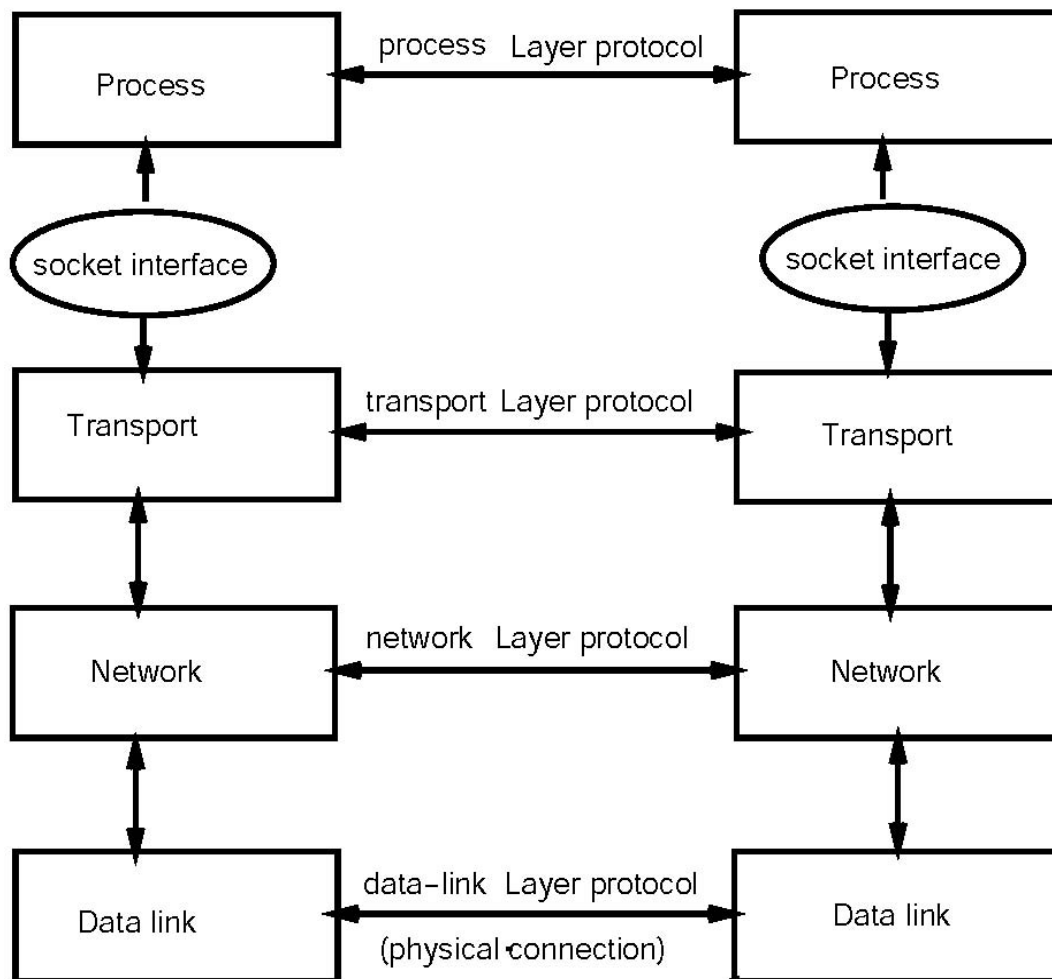
B. Boufama

UNIVERSITY OF WINDSOR

APPLICATION PROGRAM INTERFACE: THE SOCKETS

It is possible to use directly the TCP/IP protocol to implement network applications. However, it is difficult and time consuming (compare an assembler language with a high level one).

The socket interface acts between the user process and TCP/IP :



Sockets

Sockets are the traditional UNIX IPC mechanism that allows local/distant processes to talk to each other. IPC using sockets is based on the client/server paradigm.

A typical scenario can be described as follows.

- The server process creates a named socket, whose name is known by client processes, and listens on that sockets for requests from clients.
- A client process can talk to the server process by
 - creating an unnamed socket then,
 - requesting it to be connected to the server's named socket
- If succesful, one file descriptor is returned to the client and another one to the server. These file descriptors can be used for read and write allowing the server and client to communicate.

Note : Socket coonections are bidirectional.

Different kinds of sockets

Three attributes may differentiate between different kinds of sockets :

- The domain : **AF_INET** for internet and **AF_UNIX** for same machine IPC.
Note that **AF** stands for Address Family.
- The type of communication : **SOCK_STREAM**, reliable byte stream connection(TCP) and, **SOCK_DGRAM**,unreliable connectionless (UDP).
- The protocol : the low-level protocol used for communication. This parameter is usually set to 0 in system calls, which means “use the correct/default protocol”.

Socket Address Structure

Most socket functions require an argument pointer to a socket address structure, **which is protocol dependent**.

→ **Internet socket address structure**

The structure is called *sockaddr_in*, defined in *<netinet/in.h>*

Here is a version for **IPv4**:

```
typedef uint32_t in_addr_t;
typedef unsigned short sa_family_t;
struct in_addr {
    in_addr_t    s_addr;
};
struct sockaddr_in {
    uint8_t      sin_len; //length of structure(16)
    sa_family_t  sin_family; // AF_INET or AF_INET6
    in_port_t    sin_port; // 16-bit port number
    struct in_addr sin_addr; // 32-bit IP4 address
    char         sin_zero[8]; // unused
};
```

Here is a version for **IPv6**:

```
struct sockaddr_in6{
    sa_family_t      sin6_family;    // AF_INET6
    in_port_t        sin6_port;      // port number
    uint32_t          sin6_flowinfo; // IPv6 flow
                                                // information
    struct in6_addr   sin6_addr;      // IPv6 address
    uint32_t          sin6_scope_id;  // Scope ID
                                                // (new in 2.4)
};

struct in6_addr {
    unsigned char     s6_addr[16]; // IPv6 address
};
```


→ **Unix socket address structure**

The structure is called *sockaddr_un*, defined in
< *sys/un.h* > (< *linux/un.h* >)

Here is a version from **Solaris** :

```
struct sockaddr_un {  
sa_family_t sun_family;  // AF_UNIX  
char sun_path[108];  // path name  
};
```

→ **Generic Socket Address Structure**

Socket address structures are always passed by address when passed as a parameter.

Because there are several kinds of socket structures, socket functions prototypes take a pointer to the generic socket address structure, which represents any socket address structure parameter.

The generic address structure is called *sockaddr*, defined in `< sys/socket.h >`

Here is the definition of the structure :

```
struct    sockaddr{
    uint8_t      sa_len;
    sa_family_t   sin_family;
    char sa_data[14]; // protocol-specific address
};
```

Example: The *bind()* function prototype is

```
int bind(int, struct sockaddr *, socklen_t);
```

→ Any call to these function should cast the pointer to the protocol-specific socket address structure to be a point to a generic socket address structure.

For example :

```
struct sockaddr_in sv; // IPv4 socket, server
```

```
:
```

```
bind(sfd, (struct sockaddr*)&sv, sizeof(sv));
```

Creating endpoints for communication: `socket()`

Synopsis :

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

socket() creates an endpoint for communication and returns a file descriptor referencing the socket. In case of failure, *socket()* returns -1.

Calling *socket()* is the first thing a process must do in order to perform any network I/O operation.

Example:

`sd = socket(AF_INET, SOCK_STREAM, 0);`

When a reliable byte-stream connection is requested across the internet.

Notes: header files and libraries to be linked are

- Includes: `< sys/types.h >` and `< sys/socket.h >`

Initiating a connection on a socket : `connect()`

Synopsis :

`int connect(int s, struct sockaddr *srv, int len)`

Returns 0 when successful and -1 otherwise.

connect() is used by a *TCP* client to establish a connection with a *TCP* server.

The parameters have the following meanings :

- *s* is a socket descriptor that was returned by *socket()*.
- *srv* is a pointer to a socket address structure object, which must contain the IP address and port number of the server
- *len* is the size of the socket address structure.

connect() only returns when a connection is established or when an error occurs.

Binding a name to a socket : `bind()`

Synopsis :

`int bind(int s, struct sockaddr *sp, int len);`

Returns 0 when successful and -1 otherwise.

bind() assigns a local protocol address to a socket.

In case of the Internet, the protocol address consists of a 32-bit IPv4 address and a 16-bit port number.

Another description from the manual page :

When a socket is created with *socket()*, it exists in a name space (address family) but has no name assigned.

bind() requests that the name pointed to by *sp* be assigned to the socket specified by *s*.

bind() is usually called by a server to bind their local IP and a well-known port number to a socket.

Listening for connections on a socket : `listen()`

Synopsis : `int listen(int s, int backlog);`

Returns 0 when successful and -1 otherwise.

listen() is called only by a TCP server to accept connections from client sockets that will issue a *connect()*.

s is a file descriptor of a socket that has been already created.

backlog defines the maximum length the queue of pending connections may grow to.

listen() is normally called after the calls to *socket()* and *bind()*.

Accepting a connection on a socket : `accept()`

Synopsis :

```
int accept(int s, struct sockaddr *addr,  
           socklen_t *addrlen);
```

Returns a file descriptor for a new socket when successful and -1 otherwise.

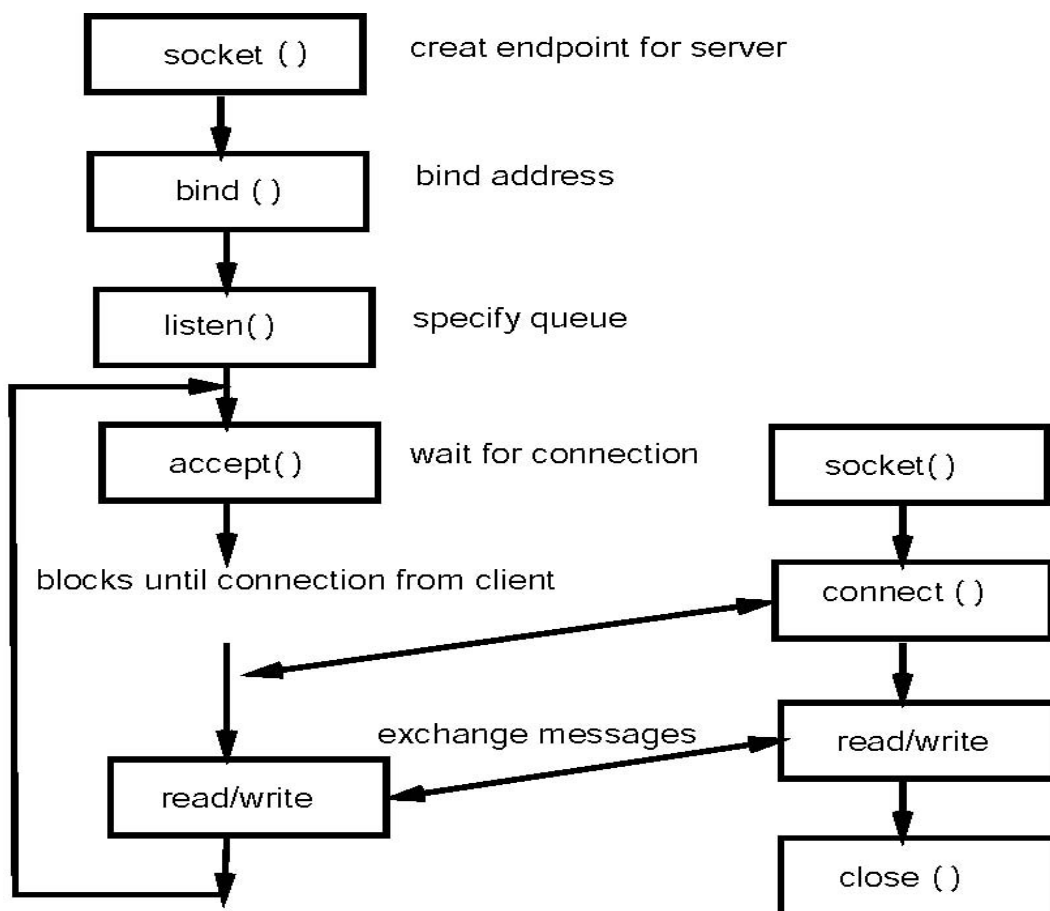
accept() is called by a TCP server to extract the first connection on the queue of pending connections, creates a new socket with the properties of *s*, and allocates a new file descriptor for the newly created socket.

If no pending connections are present on the queue, *accept()* blocks the caller until a connection is present.

Usually, the file descriptor *s* is called the listening socket while the returned value is called the connected socket.

Socket based client/server IPC

The following figure shows the typical scenario for a connection-oriented communication using sockets.



EXAMPLES : IMPLENTATION OF CLIENT/SERVER APPLICATION

Three examples are presented here:

- A date/time client/server, where the server sends its date/time to clients.
- A synchronized client/server message exchange(1 to 1).
- A texting client/server program.

A few useful functions :

- **inet_pton()**: converts an address from presentation to network format.
- **htonl()**, **htons()**, **ntohl()** and **ntohs()**: host to network or network to host conversions of multibyte integers for short and long respectively.

```
int main(int argc, char *argv[]){//E.g., 1, server
    char *myTime;
    time_t currentUnixTime; // time.h
    int sd, client, portNumber;
    socklen_t len;
    struct sockaddr_in servAdd;

    if(argc != 2){
        fprintf(stderr, "Call model: %s <Port#>\n", argv[0]);
        exit(0);
    }
    if((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
        fprintf(stderr, "Could not create socket\n");
        exit(1);
    }
    servAdd.sin_family = AF_INET;
    servAdd.sin_addr.s_addr = htonl(INADDR_ANY);
    sscanf(argv[1], "%d", &portNumber);
    servAdd.sin_port = htons((uint16_t)portNumber);
```

```
bind(sd, (struct sockaddr *) &servAdd,  
      sizeof(servAdd));  
listen(sd, 5);  
  
while(1){  
    client=accept(sd,(struct sockaddr*)NULL,NULL);  
    printf("Got a date/time request\n");  
    currentUnixTime = time(NULL);  
    myTime = ctime(&currentUnixTime);  
    write(client, myTime, strlen(myTime) + 1);  
    close(client);  
    printf("Date and Time sent\n");  
}  
}
```

```
int main(int argc, char *argv[]){//E.g., 1, client
    char message[100];
    int server, portNumber;
    socklen_t len;
    struct sockaddr_in servAdd;

    if(argc != 3){
        printf("Call model:%s <IP> <Port#>\n",argv[0]);
        exit(0);
    }
    if ((server=socket(AF_INET,SOCK_STREAM,0))<0){
        fprintf(stderr, "Cannot create socket\n");
        exit(1);
    }
    servAdd.sin_family = AF_INET;
    sscanf(argv[2], "%d", &portNumber);
    servAdd.sin_port = htons((uint16_t)portNumber);
```

```
if(inet_pton(AF_INET, argv[1],
            &servAdd.sin_addr) < 0){
    fprintf(stderr, " inet_pton() has failed\n");
    exit(2);
}
if(connect(server, (struct sockaddr *) &servAdd,
           sizeof(servAdd))<0){
    fprintf(stderr, "connect() failed, exiting\n");
    exit(3);
}
if (read(server, message, 100)<0){
    fprintf(stderr, "read() error\n");
    exit(3);
}
fprintf(stderr, "%s\n", message);
exit(0);
}
```

```
int main(int argc, char *argv[]){//E.g., 2: Server
    int sd, client, portNumber;
    socklen_t len;
    struct sockaddr_in servAdd;

    if(argc != 2){
        printf("Call model: %s <Port #>\n", argv[0]);
        exit(0);
    }
    if ((sd=socket(AF_INET,SOCK_STREAM,0))<0){
        fprintf(stderr, "Cannot create socket\n");
        exit(1);
    }
    servAdd.sin_family = AF_INET;
    servAdd.sin_addr.s_addr = htonl(INADDR_ANY);
    sscanf(argv[1], "%d", &portNumber);
    servAdd.sin_port = htons((uint16_t)portNumber);
    bind(sd,(struct sockaddr*)&servAdd,sizeof(servAdd));
    listen(sd, 5);
```

```
while(1){  
    client=accept(sd,(struct sockaddr*)NULL,NULL);  
    printf("Got a client\n");  
    if(!fork())  
        child(client);  
    close(client);  
}  
}
```



```
void child(int sd){
    char message[255];

    while(1){
        fprintf(stderr, "Enter line to send to client\n");
        fgets(message, 254, stdin);
        write(sd, message, strlen(message)+1);

        if(!read(sd, message, 255)){
            close(sd);
            fprintf(stderr, "Bye, client dead, wait for
                        a new client\n");
            exit(0);
        }
        fprintf(stderr, "Clent send back: %s\n", message);
    }
}
```

```
        // Example 2: Client
int main(int argc, char *argv[]){
    char message[255];
    int server, portNumber;
    socklen_t len;
    struct sockaddr_in servAdd;

    if(argc != 3){
        printf("Call model:%s <IP> <Port#>\n",argv[0]);
        exit(0);
    }
    if((server = socket(AF_INET, SOCK_STREAM, 0))<0){
        fprintf(stderr, "Cannot create socket\n");
        exit(1);
    }

    servAdd.sin_family = AF_INET;
    sscanf(argv[2], "%d", &portNumber);
    servAdd.sin_port = htons((uint16_t)portNumber);
```

```
if(inet_pton(AF_INET,argv[1],&servAdd.sin_addr)<0){  
    fprintf(stderr, " inet_pton() has failed\n");  
    exit(2);  
}  
if(connect(server,(struct sockaddr *)&servAdd,  
           sizeof(servAdd))<0){  
    fprintf(stderr, "connect() has failed, exiting\n");  
    exit(3);  
}
```

```
while(1){
    if(read(server, message, 255)<0){
        fprintf(stderr, "read() error\n");
        exit(3);
    }
    fprintf(stderr, "Server's messgae: %s\n",message);
    fprintf(stderr, "Enter the $ sign to quit or
                    a message for the server\n");
    fgets(message, 254, stdin);
    if(message[0] == '$'){
        close(server);
        exit(0);
    }
    write(server, message, strlen(message)+1);
}
}
```

```
int main(int argc, char *argv[]){// E.g. 3: Server
    int sd, client, portNumber, status;
    socklen_t len;
    struct sockaddr_in servAdd;

    if(argc != 2){
        printf("Call model: %s <Port #>\n", argv[0]);
        exit(0);
    }
    if((sd = socket(AF_INET, SOCK_STREAM, 0))<0){
        printf("Cannot create socket\n");
        exit(1);
    }
    servAdd.sin_family = AF_INET;
    servAdd.sin_addr.s_addr = htonl(INADDR_ANY);
    sscanf(argv[1], "%d", &portNumber);
    servAdd.sin_port = htons((uint16_t)portNumber);

    bind(sd, (struct sockaddr*)&servAdd, sizeof(servAdd));
    listen(sd, 5);
```

```
while(1){  
    client = accept(sd, (struct sockaddr *)NULL, NULL);  
    printf("Got a client\n");  
  
    if(!fork())  
        child(client);  
  
    close(client);  
    waitpid(0, &status, WNOHANG);  
}  
}
```

```
void child(int sd){
    char message[255];
    int n, pid;

    write(sd, "start chatting", 16);
    if(pid=fork()) /* reading client messages */
        while(1)
            if(n=read(sd, message, 255)){
                message[n]='\0';
                fprintf(stderr,"%s\n", message);
                if(!strcasecmp(message, "Bye\n")){
                    kill(pid, SIGTERM);
                    exit(0);
                }
            }
    else
```

```
while(1)// sending messages client
{
    if(n=read(0, message, 255)){
        message[n]='\0';
        write(sd, message, strlen(message)+1);
        if(!strcasecmp(message, "Bye\n")){
            kill(getppid(), SIGTERM);
            exit(0);
        }
    }
}
```



```
int main(int argc, char *argv[]){//E.g. 3 Client
    char message[255];
    int server, portNumber, pid, n;
    socklen_t len;
    struct sockaddr_in servAdd;

    if(argc != 3){
        printf("Call model: %s <IP> <Port #>\n", argv[0]);
        exit(0);
    }
    if((server=socket(AF_INET, SOCK_STREAM, 0))<0){
        fprintf(stderr, "Cannot create socket\n");
        exit(1);
    }
    servAdd.sin_family = AF_INET;
    sscanf(argv[2], "%d", &portNumber);
    servAdd.sin_port = htons((uint16_t)portNumber);
```

```
if(inet_pton(AF_INET, argv[1], &servAdd.sin_addr)<0){  
    fprintf(stderr, " inet_pton() has failed\n");  
    exit(2);  
}  
if(connect(server, (struct sockaddr *) &servAdd,  
           sizeof(servAdd))<0){  
    fprintf(stderr, "connect() has failed, exiting\n");  
    exit(3);  
}  
read(server, message, 255);  
fprintf(stderr, "message received: %s\n", message);  
  
pid=fork();
```

```
if(pid)
    while(1) /* reading server's messages */
        if(n=read(server, message, 255)){
            message[n]='\0';
            fprintf(stderr,"%s\n", message);
            if(!strcasecmp(message, "Bye\n")){
                kill(pid, SIGTERM);
                exit(0);
            }
        }
else while(1) // sending messages to server
    if(n=read(0, message, 255)){
        message[n]='\0';
        write(server, message, strlen(message)+1);
        if(!strcasecmp(message, "Bye\n")){
            kill(getppid(), SIGTERM);
            exit(0);
        }
    }
}
```