Introducere în Socket-uri și IPC

1 Introducere – Socket-uri și IPC

Socket-urile sunt un mecanism de comunicare *inter-proces* (IPC) care poate fi folosit atât **local**, pe aceeași mașină, cât și **prin rețea**, între mașini diferite. Din perspectiva procesului, un socket este doar un **descriptor de fișier** (un întreg) prin care putem trimite și primi date, folosind apeluri de sistem precum `read`, `write`, `send`, `recv`.

Socket-urile mai sunt numite uneori și **pipe-uri bidirecționale**, deoarece permit atât **citirea**, cât și **scrierea** de date prin același canal. Socket-urile pot fi catalogate după **domeniul lor de adresare** și după **tipul de comunicare** pe care îl oferă.

2 Crearea unui `socket()`

În C, socket-urile sunt create cu apelul:

```
#include <sys/socket.h>

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

- `domain` – **domeniul de comunicare** (sau *address family*). Definește "spațiul de adrese" în care va exista socket-ul și, implicit, ce fel de adrese și convenții sunt folosite.
- `type` – **tipul socket-ului**, adică semantica de comunicare.
- `protocol` – **protocolul concret** folosit în cadrul combinației `(domain, type)`.
- **valoarea de return** – Apelul `socket()` întoarce un **descriptor de fișier** (un întreg nenul pozitiv) în caz de succes, sau `-1` în caz de eroare și setează variabila globală `errno`.

2.1 Domeniul (`domain`)

Argumentul `domain` specifică un **domeniu de comunicare**; acesta selectează familia de adrese ce va fi folosită.

Familiile uzuale sunt definite în `<sys/socket.h>`. Exemple:

- `AF_UNIX` – comunicare locală (UNIX domain sockets);
- `AF_LOCAL` – sinonim pentru `AF_UNIX`;
- `AF_INET` – protocoale Internet IPv4;
- `AF_INET6` – protocoale Internet IPv6.

2.2 Tipul (`type`)

Socket-ul are un anumit **tip**, care specifică semantica de comunicare. Tipurile importante pentru acest laborator sunt:

- `SOCK_STREAM`
  Oferă fluxuri de octeți **secvențiale**, **fiabile**, **bidirecționale**, bazate pe conexiune. Mesajele ajung în ordine, fără duplicări, sau conexiunea se întrerupe în caz de eroare gravă. De obicei este implementat folosind **TCP** pentru `AF_INET`/`AF_INET6`.
- `SOCK_DGRAM`
  Suportă datagrame (mesaje) **fără conexiune**, **nesigure**, de lungime maximă fixă. Mesajele pot fi pierdute, duplicat sau livrate în altă ordine. De obicei este implementat folosind **UDP**.

Există și alte tipuri (de exemplu `SOCK_RAW`, `SOCK_SEQPACKET`), dar în laborator ne vom concentra pe `SOCK_STREAM` și `SOCK_DGRAM`.

2.3 Protocolul (`protocol`)

Al treilea parametru al lui `socket()` este protocolul. De cele mai multe ori, se folosește valoarea `0`, iar kernelul alege protocolul implicit pentru combinația `(domain, type)`:

- `AF_INET + SOCK_STREAM + 0` → TCP,
- `AF_INET + SOCK_DGRAM + 0` → UDP.

Un protocol nenul (de exemplu `IPPROTO_TCP`, `IPPROTO_UDP`) se specifică atunci când există mai multe opțiuni și dorim să alegem explicit una.

3 Cazuri de folosire

În practică, alegerea dintre `SOCK_STREAM` (TCP), `SOCK_DGRAM` (UDP) și socket-urile locale `AF_UNIX/AF_LOCAL` depinde de nevoile aplicației. Tabelul de mai jos rezumă situațiile tipice de utilizare:

Tip socket	**Când îl folosim?**	**Exemple tipice**
Stream (TCP / `SOCK_STREAM`)	Avem nevoie de **fiabilitate** (nu vrem să pierdem date). Ordinea mesajelor este importantă. Putem accepta overhead mai mare și o latență ușor crescută.	HTTP, FTP, SSH Aplicații clasice client–server Protocoale text (SMTP etc.)
Datagram (UDP / `SOCK_DGRAM`)	Acceptăm **pierderi ocazionale** de date. Prioritizăm **latența mică** și overhead-ul redus. Mesajele sunt scurte, independente; logica de retransmisie se poate implementa în aplicație.	Jocuri online Streaming audio/video în timp real DNS, broadcast/multicast
Local (AF_UNIX / AF_LOCAL)	Toate procesele rulează pe **aceeași mașină**. Dorim **IPC rapid**, fără stack IP (fără TCP/UDP). Nu dorim să expunem serviciul în rețea (nu avem nevoie de port accesibil extern).	Daemoni de sistem Comunicare între componente ale aceluiași serviciu Servere locale cu socket în `/run` sau `/tmp`

4 Funcții folosite

Tabelul de mai jos rezumă pe scurt rolul principal al funcțiilor de sistem folosite în lucrul cu socket-uri:

Funcție	Descriere scurtă
`bind(2)`	Asociază un socket cu o adresă locală (IP + port sau path `AF_UNIX`).
`listen(2)`	Marchează un socket *stream* ca socket pasiv, care poate accepta conexiuni; setează dimensiunea cozii de conexiuni în așteptare.
`accept(2)`	Acceptă o nouă conexiune de la coada unui socket aflat în `listen` și întoarce un **nou** descriptor de socket conectat la client.
`connect(2)`	Inițiază o conexiune de la un socket client către o adresă de server (IP + port sau `AF_UNIX` path).
`read(2)`	Citește octeți dintr-un descriptor de fișier (inclusiv un socket) într-un buffer. Operație generică de I/O.
`write(2)`	Scrie octeți dintr-un buffer către un descriptor de fișier (inclusiv un socket). Operație generică de I/O.
`recv(2)`	Citește date de pe un socket (similar cu `read`, dar suportă **flag-uri** specifice rețelei, ex. `MSG_PEEK`).
`send(2)`	Trimite date pe un socket (similar cu `write`, dar suportă **flag-uri** specifice rețelei).
`shutdown(2)`	Închide parțial o conexiune TCP: dezactivează citirea, scrierea sau ambele, fără a elibera imediat descriptorul (spre deosebire de `close`).
`close(2)`	Închide un descriptor de fișier; pentru un socket, eliberează resursa și poate trimite FIN/RES la peer, în funcție de protocol.
`socketpair(2)`	Creează **două** socket-uri conectate direct între ele (de obicei `AF_UNIX`), folosite pentru IPC local (similar cu un *pipe* bidirecțional).

5 Sarcini de laborator

1. **Comunicare IPC locală**

   Folosind `socketpair(2)` (citiți manualul), creați doi socket-i locali `AF_UNIX` de tip `SOCK_STREAM` și realizați o comunicare bidirecțională între un proces părinte și procesul copil obținut prin `fork()`.

   Cerințe:

   - Procesul copil inițiază comunicarea: citește un mesaj arbitrar dintr-un fișier (ex.: `mesaj_copil.txt`) și îl trimite părintelui prin socket.
   - Procesul părinte așteaptă mesajul copilului, îl afișează, apoi citește propriul mesaj arbitrar dintr-un alt fișier (ex.: `mesaj_parinte.txt`)

și îl trimite înapoi copilului (s.a.m.d).

2. **Client TCP pentru serverul dat**

   În anexă aveți un program `server.c` care implementează un server TCP simplu (socket `AF_INET`, `SOCK_STREAM`) ce ascultă pe un port dat și trimite inapoi mesajele primite.

   Scrieți un program `client.c` care:

   - creează un socket TCP (`AF_INET`, `SOCK_STREAM`);
   - se conectează la serverul dat în anexă (ex.: `127.0.0.1`, portul indicat în codul serverului);
   - citește linii de la tastatură și le trimite serverului;
   - primește răspunsul de la server și îl afișează pe ecran (comportament de „echo client”);
   - tratează erorile pentru `socket()`, `connect()`, `read()/recv()`, `write()/send()` folosind `perror()`;
   - închide corect socket-ul la terminarea comunicării (`close()`).

   Restricție: nu modificați codul serverului din anexă; implementați doar clientul compatibil cu acesta.

6 Anexa

server.c

```
// server.c - simple TCP echo server
// Compile: gcc server.c -o server
// Run:     ./server

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT 5000
#define BUF_SIZE 1024

int main(void) {
    int server_fd, client_fd;
```

```
struct sockaddr_in addr, cli_addr;
socklen_t cli_len = sizeof(cli_addr);
char buffer[BUF_SIZE];

// 1. Create socket
server_fd = socket(AF_INET, SOCK_STREAM, 0);
if (server_fd < 0) {
    perror("socket");
    exit(EXIT_FAILURE);
}

// Allow reuse of the address (avoid "Address already in use" on restart)
int opt = 1;
if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) < 0) {
    perror("setsockopt");
    close(server_fd);
    exit(EXIT_FAILURE);
}

// 2. Bind to 0.0.0.0:PORT
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;  // listen on all interfaces
addr.sin_port = htons(PORT);

if (bind(server_fd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
    perror("bind");
    close(server_fd);
    exit(EXIT_FAILURE);
}

// 3. Listen
if (listen(server_fd, 5) < 0) {
    perror("listen");
    close(server_fd);
    exit(EXIT_FAILURE);
}

printf("Server listening on port %d...\n", PORT);

// Main accept loop
while (1) {
    client_fd = accept(server_fd, (struct sockaddr *)&cli_addr, &cli_len);
    if (client_fd < 0) {
```

```
            perror("accept");
            continue; // try accepting next client
        }

        char client_ip[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &cli_addr.sin_addr, client_ip, sizeof(client_ip));
        printf("Client connected from %s:%d\n",
               client_ip, ntohs(cli_addr.sin_port));

        // Echo loop for this client
        while (1) {
            ssize_t n = recv(client_fd, buffer, BUF_SIZE - 1, 0);
            if (n < 0) {
                perror("recv");
                break;
            } else if (n == 0) {
                // client closed connection
                printf("Client disconnected.\n");
                break;
            }

            buffer[n] = '\0';
            printf("Received: %s", buffer);

            // Echo back
            if (send(client_fd, buffer, n, 0) < 0) {
                perror("send");
                break;
            }
        }

        close(client_fd);
    }

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