

Programarea in retea (I)

Lenuta Alboaie
adria@info.uaic.ro

Cuprins

- Modelul client/server
- API pentru programarea in retea
- *Socket*-uri BSD
 - Caracterizare
 - Creare
 - Primitive
- Modelul client/server TCP

Paradigme ale comunicarii in retea

- Modelul **client/server**
- Apelul procedurilor la distanta (**RPC**)
- Mecanismul **peer-to-peer** (**P2P**) –
comunicare punct-la-punct

Modelul client/server

- Proces **server**
 - Oferă servicii în rețea
 - Acceptă cereri de la un proces client
 - Realizează un anumit serviciu și returnează rezultatul
- Proces **client**
 - Inițializează comunicarea cu serverul
 - Solicită un serviciu, apoi așteaptă răspunsul serverului



[Primul server Web]

Modelul client/server

- Moduri de interacțiune
 - **Orientat-conexiune** – bazat pe TCP
 - **Neorientat-conexiune** – bazat pe UDP

Modelul client/server

- Implementare:
 - **iterativa** – fiecare client e tratat pe rind, secvential

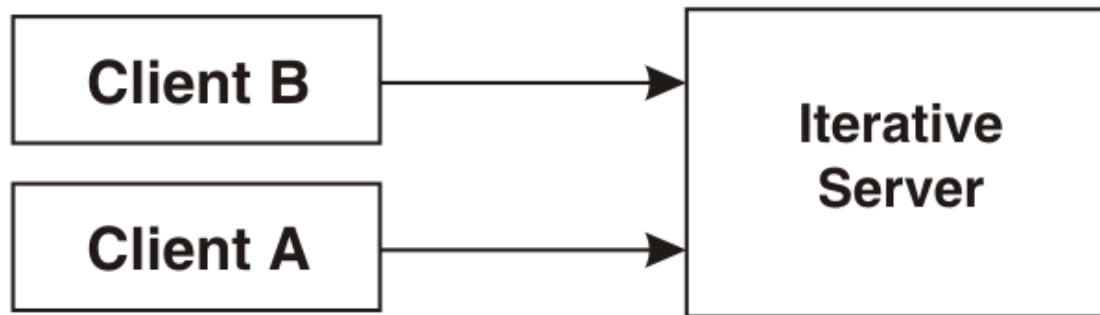


Figura: Exemplu de server iterativ

Modelul client/server

- Implementare:

- **concurrenta** – cererile sunt procesate concurrent

Procese copil pentru fiecare cerere de procesat

Multiplexarea conexiunii

Tehnici combinate

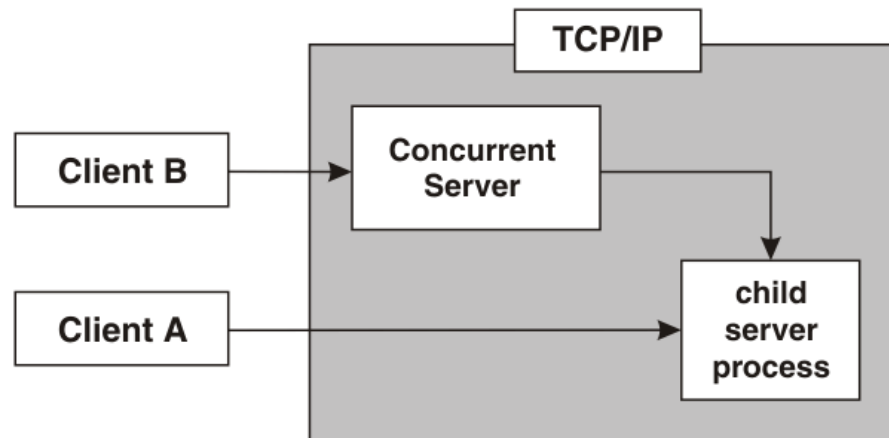


Figura: Exemplu de server concurrent

[<http://publib.boulder.ibm.com>]

API pentru programarea in retea

- Necesitate:
 - Interfata generica pentru programare
 - Independenta de hardware si de sistemul de operare
 - Suport pentru diferite protocoale de comunicatie
 - Suport pentru comunicatii orientate-conexiune si prin mesaje
 - Independenta in reprezentarea adreselor
 - Compatibilitatea cu serviciile I/O comune

API pentru programarea in retea

- Se pot utiliza mai multe API-uri pentru programarea aplicatiilor Internet
 - **Socket-uri BSD** (*Berkeley System Distribution*)
 - TLI (*Transport Layer Interface*) – AT&T, XTI (Solaris)
 - Winsock
 - MacTCP
- Functii oferite:
specificarea de puncte terminale locale si la distanta,
initierea si acceptarea de conexiuni, trimitere si receptare
de date, terminarea conexiunii, tratare erori
- TCP/IP nu include definirea unui API

...

Interfata de programare a aplicatiilor bazata pe *soket-uri* BSD

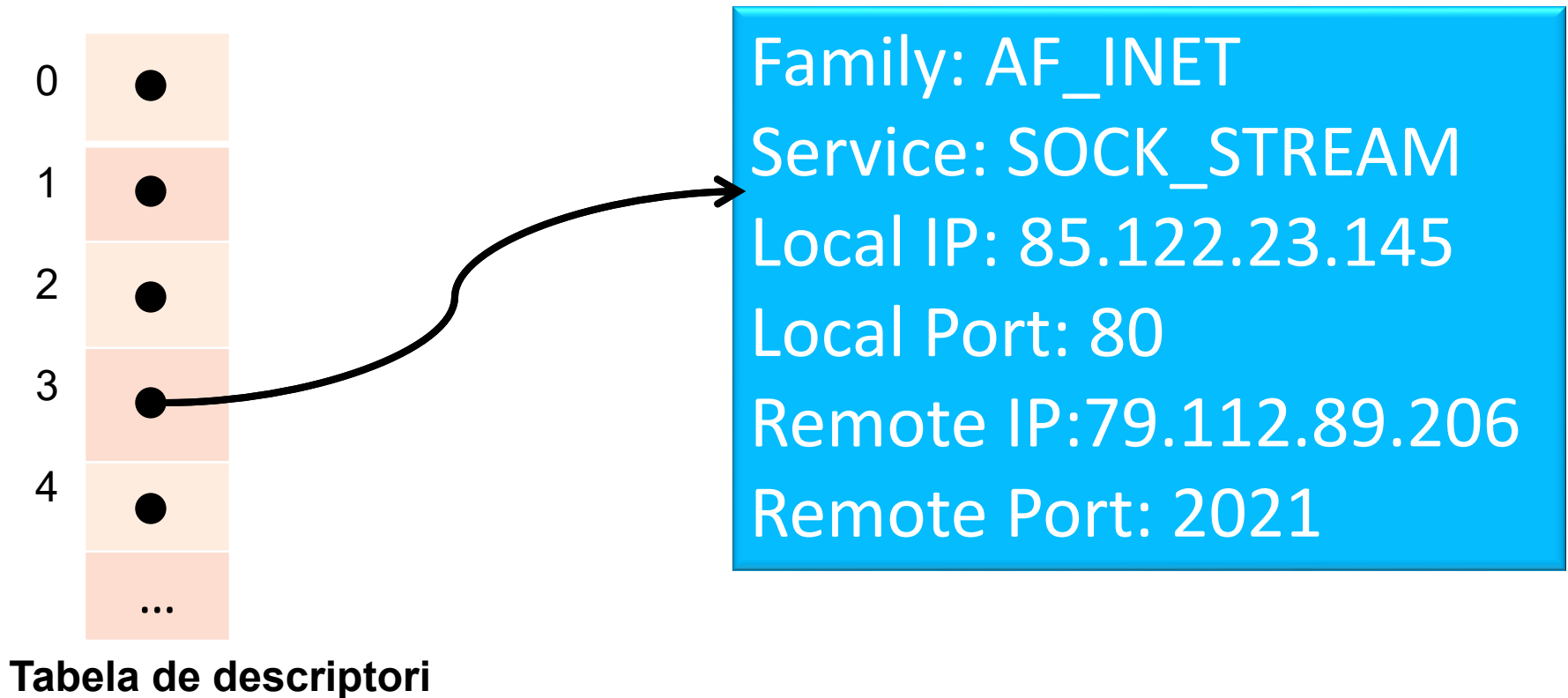
Socket

- Facilitate generala, independenta de arhitectura hardware, de protocol si de tipul de transmisiune a datelor, pentru comunicare intre procese aflate pe masini diferite, in retea
- Oferă suport pentru familii multiple de protocoale
 - Protocolul domeniului UNIX – folosit pentru comunicatii locale
 - Protocolul domeniului Internet folosind TCP/IP
 - Altele: XNS Xerox,...
- Abstracțiune a unui punct terminal (*end-point*) la nivelul transport

Socket

- Utilizeaza interfata de programare I/O existenta (similar fisierelor, *pipe*-urilor, *FIFO*-urilor etc.)
- Poate fi asociat cu unul/mai multe procese, existind in cadrul unui domeniu de comunicatie
- Oferă un API pentru programarea in retea, avind implementari multiple
- Din punctul de vedere al programatorului, un *socket* este similar unui descriptor de fisier; diferente apar la creare si la diferite optiuni de control al *socket*-urilor

Socket



Interfata de programare a aplicatiilor bazata pe *socket*-uri BSD

Primitive de baza:

- **socket()** – creaza un nou punct terminal al conexiunii
- **bind()** ataseaza o adresa locala la un *socket*
- **listen()** permite unui *socket* sa accepte conexiuni
- **accept()** blocheaza apelantul pina la sosirea unei cereri de conectare(utilizata de serverul TCP)
- **connect()** tentativa (activa) de stabilire a conexiunii (folosita de clientul TCP)
- **send()** trimitere de date via *socket*
- **recv()** receptarea de date via *socket*
- **close()** elibereaza conexiunea (inchide un *socket*)
- **shutdown()** inchide directional un *socket*

Interfata de programare a aplicatiilor bazata pe *socket*-uri BSD

Alte primitive:

- Citire de date
 - `read()` / `readv()` / `recvfrom()` / `recvmsg()`
- Trimitere de date
 - `write()` / `writew()` / `sendto()` / `sendmsg()`
- Multiplexare I/O
 - `select()`
- Administrarea conexiunii
 - `fcntl()` / `ioctl()` / `setsockopt()` / `getsockopt()` /
`getsockname()` / `getpeername()`

Socket-uri | creare

Apelul de sistem **socket()**

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

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

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

Domeniul de comunicare:
**AF_UNIX, AF_INET,
AF_INET6, ...**

Protocolul utilizat pentru
transmitere
(uzual: 0 pentru nivelul
transport)

Tipul socketului (modalitatea de realizare a
comunicarii): **SOCK_STREAM, SOCK_DGRAM,
SOCK_RAW**

Socket-uri | creare

Apelul de sistem **socket()**

Valoarea de retur

- Succes: descriptorul de *socket* creat
- Eroare: -1

- Raportarea erorii se realizeaza *via* variabilei **errno**

- **EACCES**
- **EAFNOSUPPORT**
- **ENFILE**
- **ENOBUFS** sau **ENOMEM**
- **EPROTONOSUPPORT**
- ...



Constante definite
in **errno.h**

Socket-uri

Exemplu de combinatii posibile pentru cele trei argumente ale primitivei socket():

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

Domeniu	Tip	Protocol
AF_INET	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_RAW	IP
AF_INET6	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_RAW	IPv6
AF_LOCAL	SOCK_STREAM	Mecanism intern de comunicare
	SOCK_DGRAM	

Observatie: AF_LOCAL=AF_UNIX (din motive istorice)

Socket-uri

Observatii

- Primitiva *socket()* alocă resursele necesare unui punct terminal de comunicare, dar nu stabilește modul de adresare
- *Socket*-urile oferă un mod generic de adresare; pentru TCP/IP trebuie specificate (*adresa IP, port*)
- Alte suite de protocoale pot folosi alte scheme de adresare

Tipuri POSIX

int8_t, uint8_t, int16_t, uint16_t, int32_t, uint32_t,
u_char, u_short, u_int, u_long

Socket-uri

- Tipuri POSIX folosite de socket-uri:
 - `sa_family_t` – familia de adrese
 - `socklen_t` – lungimea structurii de memorare
 - `in_addr_t` – adresa IP (v4)
 - `in_port_t` – numarul de port

- Specificarea adreselor generice

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

Familia de adrese:
AF_INET, AF_ISO,...

14 bytes - adresa folosita

Socket-uri

- Pentru IPv4 AF_INET vom avea nevoie de o structura speciala: `sockaddr_in`

```
struct sockaddr_in {  
    short int sin_family;  
    unsigned short int sin_port;  
    struct in_addr sin_addr;  
    unsigned char sin_zero[8];  
}
```

Familia de adrese:
AF_INET

Portul (2 octeti)

Bytes neutilizati

```
struct in_addr{  
    in_addr_t s_addr;  
}
```

4 bytes ai adresei IP

Socket-uri

sockaddr

sa_family

Permite
oricare tip
de adresare

sockaddr_in

AF_INET

sin_addr

Socket-uri

- Toate valorile stocate in `sokaddr_in` vor respecta ordinea de codificare a rețelei (*network byte order*)
- Functii de conversie (`netinet/in.h`)
 - **uint16_t htons (uint16_t)** – conversie a unui intreg scurt (2 octeti) de la gazda la retea;
 - **uint16_t ntohs (uint16_t);**
 - **uint32_t htonl (uint32_t)** – conversie a unui intreg lung (4 octeti) de la retea la gazda;
 - **uint32_t htonl (unit32_t);**

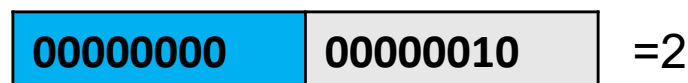
Discutii | Ordinea octetilor

Ordinea octetilor dintr-un cuvint (*word* – 2 octeti) se poate realiza in doua moduri:

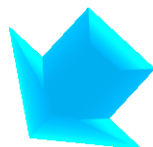
- **Big-Endian** – cel mai semnificativ octet este primul
- **Little-Endian** – cel mai semnificativ octet este al doilea

Exemplu:

Masina BigEndian trimite
(e.g. procesor Motorola)



Masina LittleEndian va interpreta:
(e.g. procesor Intel)



Drept conventie, se considera ordinea retelei
(*network byte order*) - *BigEndian*

Socket-uri

- Pentru IPv6 AF_INET6 vom avea nevoie de o structura `sockaddr_in6`:

```
struct sockaddr_in6 {  
    u_int16_t sin6_family; /* AF_INET6 */  
    u_int16_t sin6_port;  
    u_int32_t sin6_flowinfo;  
    struct in6_addr sin6_addr;  
    u_int32_t sin6_scope_id;  
}
```

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

Socket-uri

Exemplu:

// IPv4:

```
struct sockaddr_in ip4addr; int s;  
ip4addr.sin_family = AF_INET;  
ip4addr.sin_port = htons(2510);  
inet_pton(AF_INET, "10.0.0.1", &ip4addr.sin_addr);  
s = socket(PF_INET, SOCK_STREAM, 0);  
bind(s, (struct sockaddr*)&ip4addr, sizeof (ip4addr));
```

Converteste adrese IPv4 si IPv6 din sir de caractere (x.x.x.x) in ordinea de codificare a retelei (#include <arpa/inet.h>)

// IPv6:

```
struct sockaddr_in6 ip6addr; int s;  
ip6addr.sin6_family = AF_INET6;  
ip6addr.sin6_port = htons(2610);  
inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &ip6addr.sin6_addr);  
s = socket(PF_INET6, SOCK_STREAM, 0);  
bind(s, (struct sockaddr*)&ip6addr, sizeof (ip6addr));
```

? (urmatorul slide)



Socket-uri (slide 19)



Observatii

- Primitiva *socket()* alocă resursele necesare unui punct terminal de comunicare, dar nu stabilește modul de adresare
- *Socket*-urile oferă un mod generic de adresare; pentru TCP/IP trebuie specificate (*adresa IP, port*)
- Alte suite de protocoale pot folosi alte scheme de adresare



Socket-uri | asignarea unei adrese

- Asignarea unei adrese la un socket existent se realizeaza cu **bind()**

```
int bind ( int sockfd,  
          const struct sockaddr *addr,  
          int addrlen );
```

- Se returneaza: 0 in caz de succes, -1 eroare
variabila errno va contine codul de eroare
corespunzator: **EACCES** , **EADDRINUSE**, **EBADF**, **EINVAL**,
ENOTSOCK,...

Socket-uri | asignarea unei adrese

Exemplu:

```
#define PORT 2021
struct sockaddr_in adresa;
int sd;

sd = socket (AF_INET, SOCK_STREAM, 0) // TCP
adresa.sin_family = AF_INET; // stabilirea familiei de socket-uri
adresa.sin_addr.s_addr = htonl (adresaIP); //adresa IP
adresa.sin_port = htons (PORT); //portul

if (bind (sd, (struct sockaddr *) &adresa, sizeof (adresa)) == -1)
{
    perror ("Eroare la bind().\n");
    ...
}
```

Socket-uri | asignarea unei adrese

- Utilizari ale lui `bind()`:
 - Serverul doreste sa ataseze un socket la un port prestabilit (pentru a oferi servicii via acel port)
 - Clientul vrea sa ataseze un socket la un port specificat
 - Clientul cere sistemului de operare sa asigneze orice port disponibil
- In mod normal, clientul nu necesita atasarea la un port specificat
- Alegerea oricarui port liber:
`adresa.sin_port = htons(0);`

Socket-uri | asignarea unei adrese

- Alegerea adresei IP la `bind()`
 - Daca gazda are asignate mai multe adrese IP?
 - Cum se rezolva independenta de platforma?



Pentru a atasa un socket la adresa IP locala, se va utiliza in locul unei adrese IP constanta

`INADDR_ANY`

Socket-uri | asignarea unei adrese

- Conversia adreselor IP:

`int inet_aton (const char *cp, struct in_addr *inp);`

ASCII “x.x.x.x” -> reprezentare interna pe 32 biti
(*network byte order*)

`char *inet_ntoa(struct in_addr in);`

reprezentare pe 32 biti (*network byte order*)->
ASCII “x.x.x.x”

Obs: `[@fenrir ~]$ man inet_addr`

Socket-uri | asignarea unei adrese

- Observatii:

- Pentru IPv6 in locul constantei INADDR_ANY se va folosi (vezi antetul `netinet/in.h`):

- `serv.sin6_addr = in6addr_any;`

- Functiile de conversie pentru IPv6 (merg si pentru IPv4) sunt:

- `inet_pton()`

- `inet_ntop()`

Socket-uri | listen()

- Stabilirea modului pasiv de interacțiune
 - Nucleul sistemului va aștepta cereri de conectare directionate la adresa la care este atasat socketul

3-way handshake

- Conexiunile multiple receptionate vor fi plasate într-o coadă de așteptare

```
int listen(int sockfd, int backlog);
```

Numarul de
conexiuni din coada
de așteptare

- Se returneaza: 0 – succes, -1 - eroare

Socket TCP atasat
unei adrese

Socket-uri | listen()

- Observatii:
 - Alegerea valorii *backlog* depinde de aplicatie
 - Serverele HTTP trebuie sa specifice o valoare *backlog* cit mai mare (din cauza incarcarii cu cereri multiple)

Socket-uri | accept()

- Acceptarea propriu-zisa a conexiunilor din partea clientilor
 - Cand aplicatia este pregatita pentru a trata o noua conexiune, va trebui sa interogam sistemul asupra unei alte conexiuni cu un client

```
int accept (int sockfd,  
            struct sockaddr *cliaddr,  
            socklen_t *addrlen);
```

Socket TCP
(mod pasiv)

- Trebuie initial sa fie egal cu lungimea structurii **cliaddr**
- Se va returna numarul de bytes folositi in **cliaddr**

Se returneaza descriptorul de socket corespunzator punctului terminal al clientului sau -1 in caz de eroare

Socket-uri | connect()

- Incercarea de a stabili o conexiune cu serverul
 - *3-way handshake*

```
int connect (int sockfd,  
             const struct sockaddr *serv_addr,  
             socklen_t addrlen);
```

Socket TCP

- Nu necesita atasarea cu bind();
sistemul de operare va asigna o adresa
locala (IP, port)

Contine adresa serverului
(IP, port)

Se returneaza: succes -> 0, eroare -> -1

I/O TCP | read()

```
int read(int sockfd, void *buf, int max);
```

- Apelul este blocant in mod normal, `read()` returneaza doar cind exista date disponibile
- Citirea de la un socket TCP poate returna mai putini octeti decat numarul maxim dorit
 - Trebuie sa fim pregatiti sa citim cate 1 byte la un moment dat (vezi cursul anterior)
- Daca partenerul a inchis conexiunea si nu mai sunt date de primit, se returneaza `0` (EOF)
- Erori: `EINTR` – un semnal a intrerupt citirea, `EIO` – eroare I/O, `EWouldBlock` – *socket*-ul nu are date intr-o citire neblocanta

I/O TCP | write()

```
int write(int sockfd, const void *buf, int count);
```

- Apelul este blocant in mod normal
- Erori:
 - **EPIPE** – scriere la un socket neconectat
 - **EWOULDBLOCK** – nu se pot accepta date fara blocare, insa operatiunea este setata ca fiind blocanta

I/O TCP | Exemplu

```
#define MAXBUF 127 /* lungime buffer citire*/
...
char *cerere= "da-mi ceva";
char buf [MAXBUF]; /* buffer pentru raspuns*/
char *pbuf= buf; /* pointer la buffer */
int n, lung = MAXBUF; /* nr. bytes cititi, nr. bytes liberi in buffer */
...
/* trimite cererea*/
write(sd, cerere, strlen(cerere));

/* asteapta raspunsul*/
while ((n = read (sd, pbuf, lung)) > 0) {
    pbuf+= n;
    lung -= n;
}
```

Exemplu de comunicarea dintre
client si sever

Inchiderea conexiunii | `close()`

`int close(int sockfd)`

- Efect:
 - terminarea conexiunii;
 - dealocarea memoriei alocate *socket*-ului
 - pentru procese care partajeaza acelasi *socket*, se decrementeaza numarul de referinte la acel socket; cind ajunge la 0 *socket*-ul este dealocat
- Probleme:
 - serverul nu poate termina conexiunea, nu stie daca si cind clientul va mai trimite si alte cereri
 - clientul nu poate sti daca datele au ajuns la server

Inchiderea conexiunii | `shutdown()`

- Inchidere unidirectionala
 - Cind un client termina de trimis cererile, poate apela `shutdown()` pentru a specifica faptul ca nu va mai trimite date pe *socket*, fara a dealoca *socket*-ul
 - Serverul va primi EOF si, dupa expedierea catre client a ultimului raspuns, va putea inchide conexiunea

```
#include <sys/socket.h>  
int shutdown (int sockfd, int how);
```

0 – viitoare citiri de pe socket nu vor mai fi permise (SHUT_RD);
1 – viitoarele scrieri pe socket nu vor mai fi permise (SHUT_WR);
2 - citirile/scrierile nu vor mai fi permise (SHUT_RDWR)

Metaphor for Good Relationships

Copyright Dr. Laura's Network Programming Corp.

To succeed in relationships...

- you need to establish your own identity. *bind()*
- you need to be open & accepting. *accept()*
- you need to establish contacts. *connect()*
- you need to take things as they come, not as you expect them. *read might return 1 byte*
- you need to handle problems as they arise.

check for errors

Model client/server

- **Modelul unui server TCP iterativ:**

- Creare *socket* pentru tratarea conexiunilor cu clientii: `socket()`
- Pregatirea structurilor de date (`sockaddr_in`)
- Atasarea *socket*-ului la adresa locala (port): `bind()`
- Pregatirea *socket*-ului pentru ascultarea portului in vederea stabilirii conexiunii cu clientii: `listen()`
- Asteptarea realizarii unei conexiuni cu un anumit client (deschidere pasiva): `accept()`
- Procesarea cererilor clientului, folosindu-se *socket*-ul returnat de `accept()`: succesiune de `read()/write()`
- Inchiderea (directionata) a conexiunii cu clientul: `close()`, `shutdown()`

Model client/server

- **Modelul unui client TCP:**

- Creare *socket* pentru conectarea la server: `socket()`
- Pregatirea structurilor de date (`sockaddr_in`)
- Atasarea socket-ului: `bind()` – optional
- Conectarea la server (deschidere activa): `connect()`
- Solicitarea de servicii si receptionarea rezultatelor trimise de server: succesiune de `write()/read()`
- Inchiderea (directionata) a conexiunii cu serverul: `close()`, `shutdown()`

Model general - server/client TCP

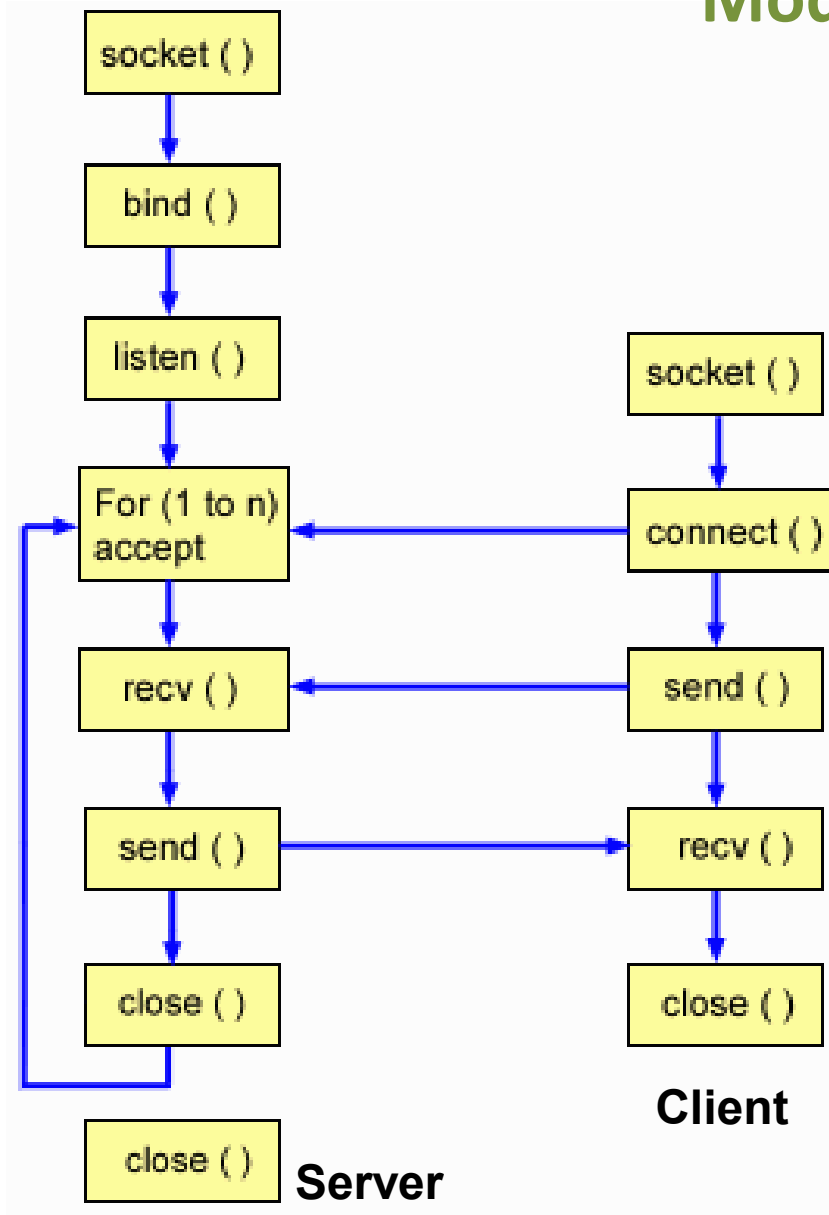
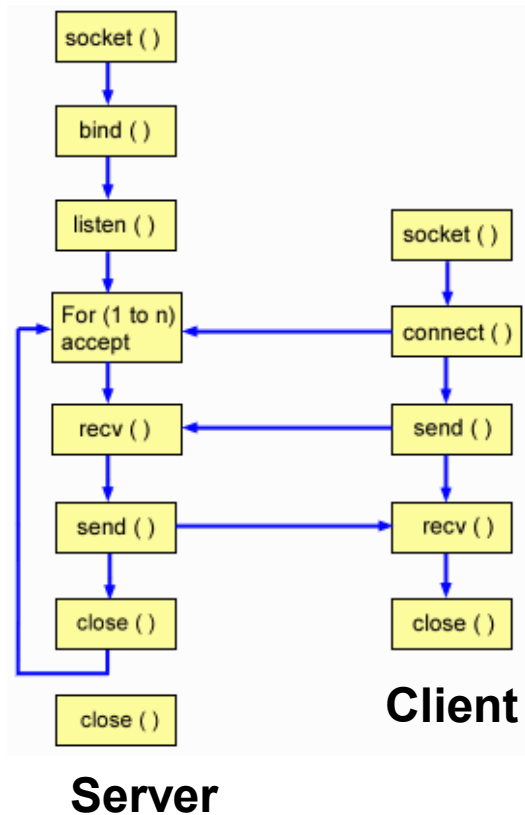


Figura: Server TCP Iterativ-sucesiunea de evenimente

DEMO ☺

Exemplu de server/client TCP iterativ



Cuprins

- Modelul client/server
- API pentru programarea in retea
- *Socket*-uri BSD
 - Caracterizare
 - Creare
 - Primitive
- Modelul client/server TCP



Intrebari?