



Retele de calculatoare

Programarea in retea II

Sabin-Corneliu Buraga

<http://www.infoiasi.ro/~busaco>



“Writing software is more fun than working.”
/usr/games/fortune



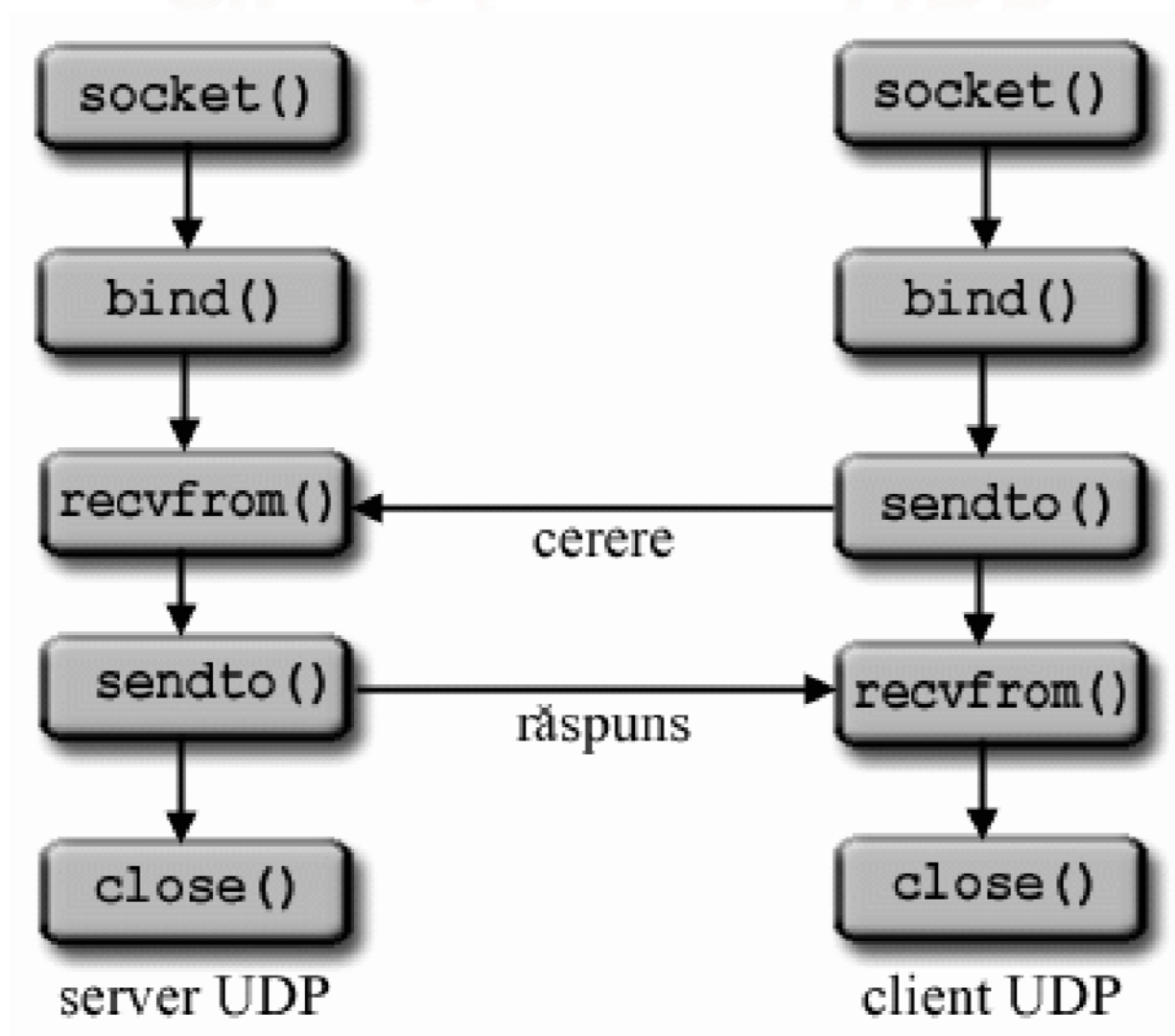
Cuprins

- Modelul client/server UDP
- Primitive I/O folositoare
- Aspecte mai avansate de programare Internet
- Critici aduse API-ului *socket*

Client/server UDP

- Pentru **socket ()** se va folosi **SOCK_DGRAM** in loc de **SOCK_STREAM**
- Apelurile **listen ()**, **accept ()**, **connect ()** nu vor mai fi utilizate in mod uzual
- Pentru citire/scriere de datagrame, se pot folosi **sendto ()** si **recvfrom ()**, respectiv
- Se pot utiliza, mai general, **send ()** si **recv ()**
- Nimeni nu garanteaza ca datele expediate au ajuns la destinatar sau ca nu sint duplicate!

Client/server UDP



Client/server UDP

- Socket-urile UDP pot fi conectate: clientul poate folosi **connect ()** pentru a specifica adresa (IP, port) a punctului terminal (serverul) – **pseudo-conexiuni**
 - Convenabil pentru transmiterea mai multor datagrame la acelasi server, fara a mai specifica adresa serverului pentru fiecare datagrama in parte
 - Pentru UDP, **connect ()** va retine doar informatiile despre punctul terminal, fara a se initia nici un schimb de date
 - Desi **connect ()** raporteaza succes, nu inseamna ca adresa punctului terminal e valida sau serverul este disponibil

Client/server UDP

- **Pseudo-conexiuni UDP**
 - Se poate utiliza **shutdown()** pentru a opri directionat transmiterea de date, dar nu se va trimite nici un mesaj partenerului de conversatie
 - Primitiva **connect()** poate fi apelata si pentru a elimina o pseudo-conexiune

Alte primitive I/O

`readv()`/`writev()`

- Mai generale ca `read()`/`write()`, ofera posibilitatea de a transmite date aflate in zone necontigue de memorie

`recv()`/`send()`

- Oferă modalități de controlare a transmisiei (*e.g.*, receptare/trimitere de pachete “*out-of-band*”, “*urgent data*”, fara rutare locala,...)

`recvmsg()`/`sendmsg()`

- Receptioneaza/transmite mesaje, extragindu-le din structura `msghdr`

Alte primitive | informatii

getsockname ()

- Returneaza numele curent al unui *socket* (local)
 - adresa la care este atasat

```
int getsockname( int sockfd,  
                 struct sockaddr *name,  
                 socklen_t *namelen );
```

getpeername ()

- Returneaza informatii despre celalalt capat al conexiunii

```
int getpeername( int sockfd,  
                 struct sockaddr *name,  
                 socklen_t *namelen );
```

Programare retea avansata

- Optiuni atasate *socket*-urilor
 - `getsockopt()` si `setsockopt()`
- I/O bazate pe semnale
- Multiplexare I/O
- Alternative la modelul client/server clasic
- Date trimise in regim *out-of-band*

Primitive | optiuni

- Optiuni atasate *socket*-urilor
 - Atribute utilizate pentru consultarea sau modificarea unui comportament, general ori specific unui protocol, pentru unele (tipuri de) *socket*-uri
 - Tipuri de valori:
 - booleene (*flag*-uri)
 - mai “complexe”:
int, *timeval*, *in_addr*, *sock_addr* etc.

Primitive | optiuni

- Consultarea optiunilor

```
int getsockopt ( int sockfd,
                int level,
                int optname,
                void *opval,
                socklen_t *optlen );
```

Numele,
val. si lung.
optiunii

level – indica daca optiunea este una generala ori
specifica unui protocol

Primitive | optiuni

- Setarea optiunilor

```
int setsockopt ( int sockfd,  
                int level,  
                int optname,  
                void *opval,  
                socklen_t optlen );
```

- Returneaza 0 = succes, -1 = eroare:
EBADF, ENOTSOCK, ENOPROTOOPT, EFAULT

Primitive | optiuni

- Optiuni generale
 - Independente de protocol
 - Unele suportate doar de anumite tipuri de *socket*-uri (**SOCK_DGRAM**, **SOCK_STREAM**)
 - SO_BROADCAST**
 - SO_DONTROUTE**
 - SO_ERROR**
 - SO_KEEPALIVE**
 - SO_LINGER**
 - SO_RCVBUF**, **SO_SNDBUF**
 - SO_REUSEADDR**

Primitive | optiuni

SO_BROADCAST (boolean)

- Activeaza/dezactiveaza trimiterea de date in regim *broadcast*
- Nivelul legatura de data folosit trebuie sa suporte *broadcasting*-ul
- Utilizata doar pentru **SOCK_DGRAM**
- Previne anumite aplicatii sa nu trimita in mod neadecvat *broadcast*-uri



Primitive | optiuni

SO_DONTROUTE (boolean)

- Utilizat de *daemon*-ii de rutare
- Dezactiveaza/activeaza rutarea pachetelor de date



Primitive | optiuni

SO_ERROR (int)

- Indica eroarea survenita (similara lui **errno**)
- Poate fi doar consultata, nu setata

Primitive | optiuni

SO_KEEPALIVE (boolean)

- Folosita pentru **SOCK_STREAM**
- Se va trimite o informatie de proba celuilalt punct terminal daca nu s-a realizat schimb de date timp indelungat
- Utilizata de TCP (*e.g.*, telnet): permite proceselor sa determine daca procesul/gazda cealalta a picat

Primitive | optiuni

SO_LINGER (struct linger)

- Controleaza daca si dupa cit timp un apel de inchidere a conexiunii va astepta confirmari (ACK-uri) de la punctul terminal
- Folosita doar pentru *socket*-uri orientate-conexiune pentru a ne asigura ca un apel `close()` nu va returna imediat
- Valorile vor fi de tipul:

```
struct linger {
    int l_onoff;          /* 0 = off */
    int l_linger;        /* timpul in secunde */
};
```

Primitive | optiuni

SO_RCVBUF / SO_SNDBUF (int)

- Modifica dimensiunile *buffer*-elor de receptionare sau de trimitere a datelor
- Utilizate pentru **SOCK_DGRAM** si **SOCK_STREAM**
- Pentru TCP, se controleaza dimensiunea ferestrei glisante
 - trebuie stabilita inainte de stabilirea conexiunii

Primitive | optiuni

SO_REUSEADDR (boolean)

- Permite atasarea la o adresa (port) deja in uz
- Folosita pentru servere tranziente pentru ca un *socket* pasiv sa poata folosi un port deja utilizat de alte procese
- Poate fi utila cind dorim sa avem servere separate pentru acelasi serviciu (folosind eventual alte interfete/adrese IP)
- Exemplu: servere Web virtuale



Primitive | optiuni

- Optiuni specifice protocolului IP
 - IP_TOS** permite setarea cimpului *Type Of Service* (e.g., ICMP) din antetul IP
 - IP_TTL** permite setarea cimpului *Time To Live* din antetul IP

Exista si multe alte optiuni pentru IPv6

Primitive | optiuni

- Optiuni specifice protocolului TCP
 - **TCP_KEEPALIVE** seteaza timpul de asteptare daca **SO_KEEPALIVE** este activat
 - **TCP_MAXSEG** stabileste lungimea maxima a unui segment (nu toate implementarile permit modificarea acestei valori de catre aplicatie)
 - **TCP_NODELAY** seteaza dezactivarea **algoritmului Nagle** (reducerea numarului de pachete de dimensiuni mici intr-o retea WAN; TCP va trimite intotdeauna pachete de marime maxima, daca este posibil) – utilizata pentru generatori de pachete mici (*e.g.*, clienti interactivi precum telnet)

Primitive | optiuni

Exemplu de utilizare:

```
int valoare = 1;
if (setsockopt (sd, /* desc. de socket */
              SOL_SOCKET,
              SO_REUSEADDR,
              (void *)&valoare,
              sizeof (valoare)) < 0) {
    perror ("setsockopt");
    exit (1);
}
```

I/O

- Problema:
 - Sa se trimita un semnal procesului atunci cind se intimpla “ceva” la un *socket*
- Solutie:
 - Indicam nucleului sa trimita semnalul **SIGIO** la fiecare eveniment I/O asupra unui descriptor de *socket*
 - Functia de tratare a semnalului trebuie sa determine cauzele aparitiei si sa realizeze actiunea dorita

I/O | UDP

- Semnalul **SIGIO** apare cind:
 - Se receptioneaza o datagrama
 - Are loc o eroare asincrona
 - Exemplu:
 - eroare ICMP (*net unreachable, invalid address*)
- Putem permite proceselor sa realizeze alte activitati si sa monitorizeze transferurile UDP
- Exemplu: “stampilarea” datagramelor

I/O | TCP

- Conditii de aparitie a **SIGIO**:
 - Conexiunea a fost complet stabilita
 - O cerere de deconectare a fost initiata
 - Cererea de deconectare a fost realizata completa
 - **shutdown()** pentru o directie a comunicatiei
 - Au aparut date de la celalalt punct terminal
 - Datele au fost trimise
 - Eroare asincrona

- Utilizat pentru **comunicatii asincrone**

Multiplexarea I/O

- Posibilitatea de a monitoriza mai multi descriptori I/O
 - Un client TCP generic (*e.g.*, telnet)
 - Un client interactiv (scp, client IRC, browser Web,...)
 - Un server care poate manipula mai multe protocoale (TCP si UDP) simultan
 - Rezolvarea unor situatii neasteptate (*i.e.* caderea unui server in mijlocul comunicarii)
- Exemplu: datele citite de la intrarea standard trebuie scrise la un *socket*, iar datele receptionate prin retea trebuie afisate la iesirea standard

Multiplexarea I/O | solutii

1. Utilizarea mecanismului neblokant folosind primitivele `fnctl()` / `ioctl()`
2. Folosirea `alarm()` pentru a intrerupe apelurile de sistem lente
3. Utilizarea unor procese/*thread*-uri multiple
4. Folosirea unor primitive care suporta verificarea existentei datelor de intrare de la descriptori de citire multipli: `select()` si `poll()`

Multiplexarea I/O | solutii

- Utilizarea apelului `fnctl()`

- Se seteaza apelurile I/O ca neblocante

```
int flags;
```

```
flags = fcntl ( sd, F_GETFL, 0 );
```

```
fcntl ( sd, F_SETFL,
        flags | O_NONBLOCK );
```

- Orice citire – cu `read()` ori alt apel – va returna eroare si `errno` va fi setat pe `EWOULDBLOCK`

- In loc de `fnctl()` se poate folosi si `ioctl()`

Multiplexarea I/O | solutii

- Utilizarea `fnctl()` – exemplu:

```

while (!nu_am_terminat) {
    if ((n = read (0,...) < 0))          /* citire de la STDIN */
        if (errno != EWOULDBLOCK) { /* eroare */ }
    else
        write (tcpsock,...);          /* scriere pe retea */

    if ( (n = read (tcpsock,...) < 0)) /* citire de pe retea */
        if (errno != EWOULDBLOCK) { /* eroare */ }
    else
        write (1,...);                /* scriere la STDOUT */
}

```

Multiplexarea I/O | solutii

- Utilizarea alarmelor

```
signal (SIGALRM, alarma);
```

```
alarm (MAX_TIME);
```

```
read (0, ...);
```

```
...
```

```
signal (SIGALRM, alarma);
```

```
alarm (MAX_TIME);
```

```
read (tcpsock, ...);
```

```
...
```

Functie scrisa
de programator

Multiplexarea I/O

- Probleme care apar:
 - Folosind apeluri nebloccante, se utilizeaza intens procesorul
 - Pentru `alarm()`, care este valoarea optima a constantei `MAX_TIME`?

Multiplexarea I/O | select()

- Permite utilizarea apelurilor blocante pentru un set de descriptori (fisiere, *pipe*-uri, *socket*-uri,...)

```
int select (int nfdsl,
           fd_set *readfds,
           fd_set *writefds,
           fd_set *exceptfds,
           struct timeval *timeout);
```

Valoarea maxima a
descript. plus 1

Multimea descriptorilor de citire,
scriere si exceptie

Timpul de
asteptare

Multiplexarea I/O | select()

- Pentru timpul de asteptare se folosește structura definită în `sys/time.h`:

```
struct timeval {
    long tv_sec;          /* secunde */
    long tv_usec;        /* microsecunde */
}
```

Multiplexarea I/O | select()

- Manipularea elementelor multimii de descriptori (tipul `fd_set`):

```
void FD_ZERO (fd_set *fdset);  
void FD_SET (int fd, fd_set *fdset);  
void FD_CLR (int fd, fd_set *fdset);  
int FD_ISSET (int fd, fd_set *fdset);
```



Multiplexarea I/O | select()

- Returneaza:
 - Numarul descriptorilor pregatiti pentru o operatiune de citire, scriere sau exceptie
 - Valoarea **0** – timpul s-a scurs, nici un descriptor nu este gata
 - Valoarea **-1** in caz de eroare

Multiplexarea I/O | select()

- “Reteta” de utilizare:
 - Declararea unei variabile de tip `fd_set`
 - Initializarea multimei cu `FD_ZERO()`
 - Adaugarea cu `FD_SET()` a fiecarui descriptor dorit a fi monitorizat
 - Apelarea primitivei `select()`
 - La intoarcerea cu succes, verificarea cu `FD_ISSET()` a descriptorilor pregatiti pentru I/O

Multiplexarea I/O | `select()`

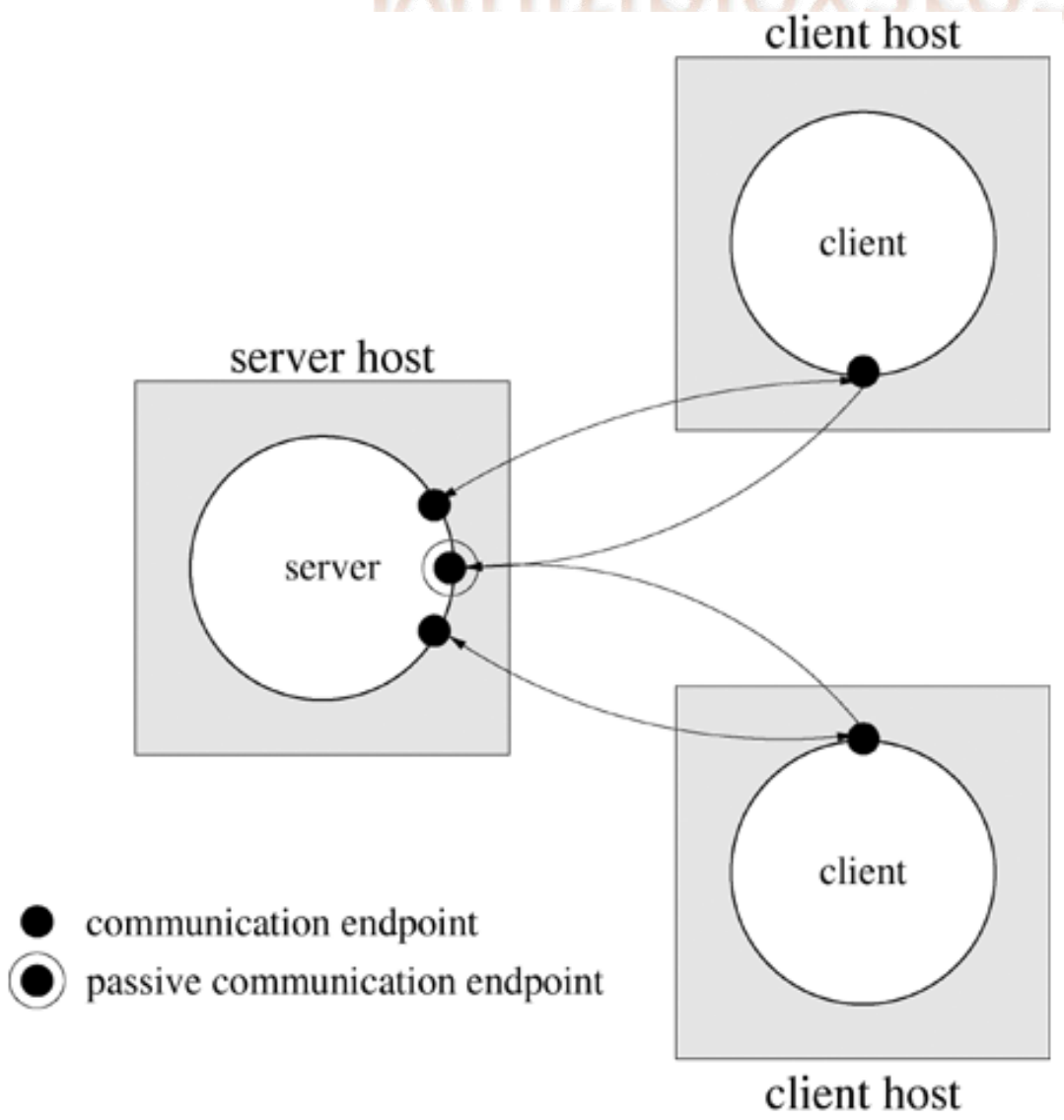
- Un descriptor de *socket* e gata de citire daca:
 - Exista octeti receptionati in *buffer*-ul de intrare (`read()` returneaza >0)
 - O conexiune TCP a receptionat **FIN** (`read()` returneaza 0)
 - *Socket*-ul e *socket* de ascultare si nr. de conexiuni complete este nenul – se poate utiliza `accept()`
 - A aparut o eroare la *socket* (`read()` returneaza -1 , cu `errno` setat) – erorile pot fi tratate via `getsockopt()` cu optiunea **SO_ERROR**

Multiplexarea I/O | select()

- Un descriptor de *socket* e gata de scriere daca:
 - Exista un numar de octeti disponibili in *buffer*-ul de scriere (`write()` returneaza >0)
 - Conexiunea in sensul scrierii este inchisa (`write()` va genera SIGPIPE)
 - A aparut o eroare de scriere (`write()` returneaza -1, cu `errno` setat) – erorile pot fi tratate via `getsockopt()` cu optiunea `SO_ERROR`
- Un descriptor de *socket* este gata de exceptie daca:
 - Exista date *out-of-band* sau *socket*-ul este marcat ca *out-of-band* – vezi urmatoarele *slide*-uri



Multiplexarea I/O | select()



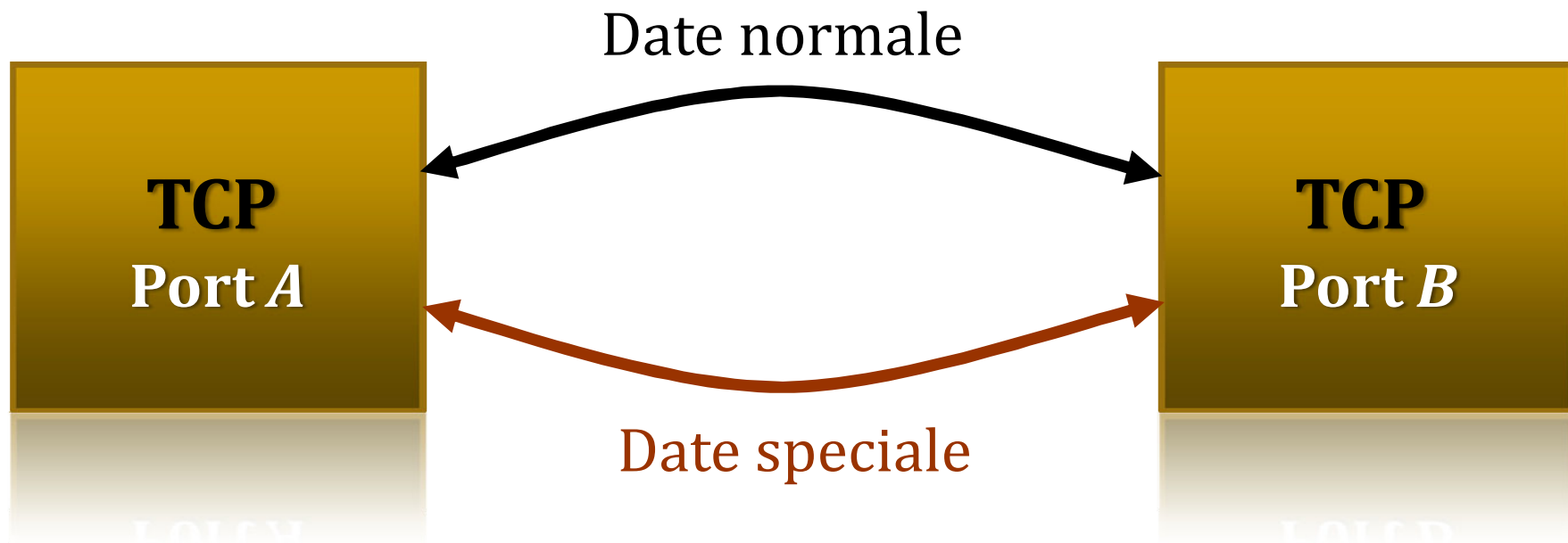
Mai multi clienti pot face cereri la acelasi punct final de comunicare

Alternative

- Servere concurente *pre-forked*
 - Se creeaza un numar de procese copil imediat la initializare, fiecare proces liber interactionind cu un anumit client
- Servere concurente *pre-threaded*
 - Ca mai sus, dar se folosesc *thread*-uri (fire de executie) in locul proceselor (vezi POSIX threads – [pthread.h](#))
 - Exemplu: serverul Apache (initial, se creeaza 6 instante [httpd](#))
- Probleme:
 - Numarul de clienti mai mare decit numarul de procese/*thread*-uri
 - Numarul de procese/*thread*-uri prea mare fata de numarul de clienti

Out of band

- TCP (si alte protocoale de transport) ofera un mecanism pentru transmiterea cu prioritate ridicata a anumitor date
- Astfel, putem avea doua fluxuri de date:

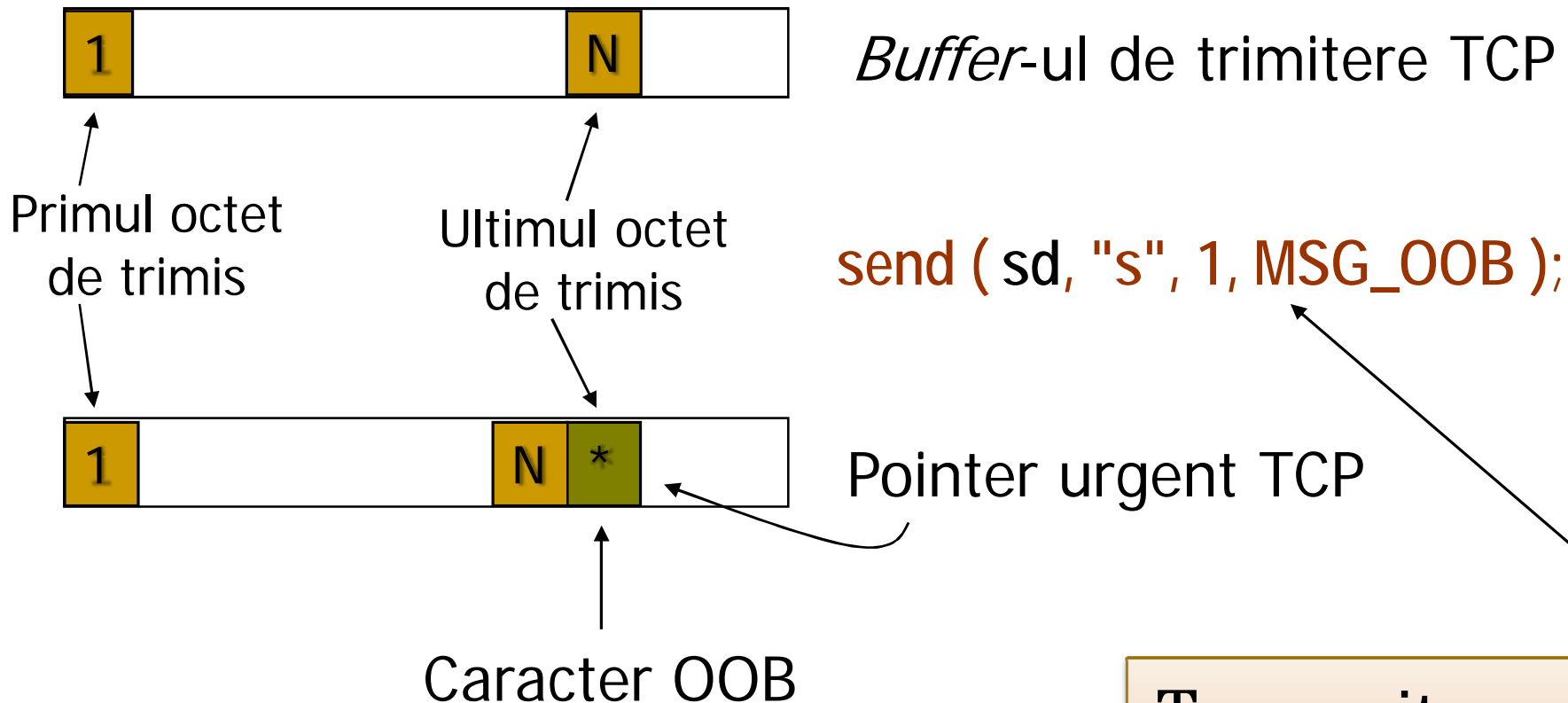


Out of band

- Se utilizeaza bitul **URG** setat in antetul TCP
- Antetul TCP contine un cimp indicind locatia datelor urgente ce trebuie trimise
- Trimiterea datelor OOB:
 - Pentru a expedia un octet urgent intr-un flux de date putem utiliza **send()**:
send (sd, buff, 1, MSG_OOB);
- Receptionarea datelor OOB:
 - Se genereaza semnalul **SIGURG**
 - Apelul **select()** va modifica lista descriptorilor de exceptie

Out of band

- Exemplu de trimitere a unui octet OOB



Transmiterea unui caracter OOB

Out of band

- Citirea datelor OOB:
 - Se utilizeaza **recv()**, setind **MSG_OOB**
 - Se foloseste monitorizarea *out-of-band-mark* pentru conexiune: **socketmark()**

Out of band

- Utilizari:
 - Modalitate de comunicare celuilalt punct terminal a unei conditii de exceptie chiar si in cazul cind controlul fluxului a oprit emitatorul
 - Pentru a detecta timpuriu erori de comunicare intre client si server (*heart-beat*)

Out of band

- Erori posibile
 - Se asteapta citirea de date OOB, dar ele nu au fost inca trimise – se returneaza **EINVAL**
 - Procesul a fost notificat ca va primi date OOB (via **select()** sau **SIGURG**), dar ele inca nu au ajuns – se returneaza **EWOULDBLOCK**
 - Se incearca sa se citeasca un acelasi octet OOB de mai multe ori – se returneaza **EINVAL**

- Tratare OOB prin SIGURG

...

```
sd = listen();
cl = accept (sd, NULL, NULL);
signal (SIGURG, fct_urg);
fcntl (cl, F_SETOWN, getpid());
while (1) {
    if ((n = read (cl, buf, sizeof (buf) - 1)) == 0)
        { /* EOF */ }
    buf[n] = '\0';
    printf ("am citit %d octeti normali: %s", n, buf);
}

void fct_urg (int sig) { /* trateaza SIGURG */
    n = recv (cl, buf, sizeof (buf) - 1, MSG_OOB);
    buf[n] = '\0';
    printf ("OOB %d: %s", n, buf);
}
```

Critici

- API-ul *socket* BSD are o serie de limitari:
 - Incurajeaza programarea cu erori
 - Are o complexitate ridicata, deoarece a fost proiectat sa suporte familii de protocoale multiple (dar rar folosite in practica)
 - Nu e asigurata portabilitatea (unele apeluri/tipuri au alte denumiri/reprezentari pe alte platforme; numele fisierelor-antet *.h* depind de sistem)
 - Exemplu: la WinSock descriptorii de *socket* sunt pointeri, in contrast cu implementarile Unix care folosesc intregi

```

1 #include <sys/types.h>
2 #include <sys/socket.h>
3 const int PORT_NUM = 10000;

```

```

4 int echo_server () {

```

```

5 struct sockaddr_in addr;

```

```

6 int addr_len;

```

```

7 char buf[BUFSIZ];

```

```

8 int cd;

```

```

9 int sd = socket (PF_UNIX, SOCK_DGRAM, 0);

```

```

10 if (sd == -1) return -1;

```

```

11 addr.sin_family = AF_INET;

```

```

12 addr.sin_port = PORT_NUM;

```

```

13 addr.sin_addr.addr = INADDR_ANY;

```

```

14 if (bind (sd, (struct sockaddr *) &addr, sizeof addr) == -1) return -1;

```

```

15 if (cd = accept (sd, (struct sockaddr *) &addr, &addr_len) != -1) {

```

```

16     int n;

```

```

17     while (n = read (sd, buf, sizeof (buf)) > 0)

```

```

18         write (cd, buf, n);

```

```

19     close (cd);

```

```

20 }

```

```

21 }

```

neinitializarea
tuturor membrilor

neinitializat

htons() lipseste

PF_UNIX incompatibil cu AF_INET!

listen() omis!

Critici

- Exemplul nu este portabil, deoarece la WinSock *socket*-urile sunt de tip **SOCKET** si nu **int** (mai mult, erorile sunt indicate via *macro*-ul nestandardizat **INVALID_SOCKET_HANDLE**)
- **accept()** e apelat pentru un *socket* de tip invalid, insa compilatorul nu semnaleaza eroarea
- Erori in liniile 15 si 17 privitoare la precedenta operatorilor (greseala comuna in C/C++)
- In linia 17, **read()** foloseste *socket*-ul pasiv
⇒ erori bizare in momentul rularii
- Pot aparea posibile erori de scriere: nu se verifica numarul octetilor trimisi in retea via **write()** – linia 18

Rezumat

- Modelul client/server UDP
- Primitive I/O folositoare
- Aspecte mai avansate de programare Internet
- Critici aduse API-ului *socket*



Intrebari?