Cuprins

I Socluri **7**

1. Socluri **9**
1.1. Introducere 9
1.2. Concepte 10
1.3. Apeluri sistem 13
1.4. Exemple simple 17
1.4.1. Exemplul 1: transfer de fişiere 20
1.4.2. Exemplul 2: utilizarea datagramelor 24
1.4.3. Exemplul 3: server concurent 26
1.4.4. Exerciţii 37
1.5. Aplicaţii 38
1.5.1. Fire de execuţie 38
1.5.2. Distribuitor de mesaje 45

Partea I

Socluri

Capitolul 1

Programarea rețelelor utilizând socluri

1.1. Introducere

Am văzut în capitolul anterior că nivelul transport din modelul OSI pune la dispoziţia nivelurilor superioare servicii capăt la capăt de comunicaţie. Nivelurile superioare (peste nivelul aplicaţie) sunt îndeobşte implementate de către aplicaţii ce rulează deasupra sistemului de operare. Sistemul de operare este cel care asigură, de obicei, funcţionalitatea la nivelurile inferioare. Prin urmare, este necesar să existe o metodă de a folosi serviciile nivelului transport de către aplicaţiile utilizatorilor. Există câteva astfel de interfeţe (API), dintre care probabil cea mai comună este Berkeley sockets, apărută mai întâi în sistemele BSD iar mai apoi răspândită pe alte variante de UNIX şi pe alte sisteme de operare.

Există o sumedenie de cărţi, ghiduri de programare şi alte resurse referitoare la *socket programming* (programarea cu socluri), de aceea ne propunem să dăm mai degrabă o serie de exemple comentate, din care să reiasă modul de lucru cu socluri în diverse situaţii, decât un material de referinţă. Pentru mai multe detalii despre Berkeley sockets, vezi bibliografia.

Înţelegerea exemplelor date în acest capitol necesită cunoaşterea limbajului C. De asemenea, este bine ca cititorul să fie familiarizat cu programarea în Linux/UNIX. În particular, programele următoare folosesc apeluri sistem

şi funcţii de bibliotecă pentru a manipula fişiere, pentru a crea procese şi pentru a trata semnale. În ceea ce priveşte cunoştinţele despre reţelele de calculatoare, sunt suficiente noţiunile date în capitolul precedent. Însă cu cât mai bine înţelegem comportamentul reţelelor de calculatoare cu atât mai uşor ne este să scriem programe distribuite ce se vor comporta corect în toate situaţiile.

Ca şi în restul cărţii, ne vom axa pe suita de protocoale TCP/IP, deşi soclurile pot fi folosite cu diverse alte arhitecturi de reţea. În terminologia Berkeley sockets, aceste arhitecturi se numesc domenii (*domains*) sau familii de protocoale (*protocol families*). Astfel, TCP/IP se numeşte *domeniul Internet* (*Internet domain*). Un exemplu de alt domeniu este *domeniul UNIX*, care permite comunicarea între procese ce rulează pe acelaşi sistem.

Există diferenţe între diversele implementări pe diferite platforme, sau de la o versiune la alta. De obicei aceste diferenţe nu sunt mari în ceea ce priveşte funcţionalitatea de bază a soclurilor, dar recomandăm cititorului să consulte paginile de manual sau documentaţia aferentă sistemului de operare folosit. Exemplele date în acest capitol au fost scrise şi testate sub Linux, cu un nucleu versiunea 2.4.20. Cu eventuale mici modificări, ele ar trebui să funcţioneze corect şi pe alte platforme.

Vom utiliza termenii „domeniu" , „familie de protocoale" şi „arhitectură de reţea" interschimbabil. De asemenea, vom utiliza atât termenul de „soclu" cât şi varianta engleză, „socket". „Sockets", la plural, va fi folosit adesea pentru a desemna întregul mecanism al soclurilor.

Următorul subcapitol prezintă familiile de protocoale suportate de socluri şi felul în care se reprezintă adresele de reţea. Secţiunea 1.3 cuprinde o scurtă prezentare a celor mai importante apeluri de sistem legate de programarea cu socluri. Cele câteva exemple simple din secţiunea 1.4 ilustrează modul de folosire al acestor apeluri de sistem cât şi al câtorva funcţii de bibliotecă utile. Subcapitolul 1.5 prezintă câteva aplicaţii mai avansate şi conceptele de programare corespunzătoare.

1.2. Concepte de programare a sockets

Modul de lucru cu *sockets* este foarte asemănător cu lucrul cu fişiere. Un soclu reprezintă un capăt de comunicaţie. La fel ca şi în cazul fişierelor, putem scrie şi citi un soclu prin intermediul unui descriptor iar aceste operaţii vor fi traduse de sockets în operaţii la nivelul protocoalelor

de reţea.

Mecanismul soclurilor suportă, în principiu, mai multe familii de protocoale. Diversele versiuni de sockets implementează seturi distincte de domenii. Implementarea pe care o utilizăm în acest capitol (vezi secţiunea 1.1) foloseşte, printre altele: TCP/IP, IPv6, Novel IPX, Appletalk, cât şi familia UNIX utilizată exclusiv pentru comunicarea între procesele de pe aceeaşi maşină. Acestor familii de protocoale le corespunde câte o constantă. De exemplu, pentru TCP/IP avem PF_INET iar pentru domeniul UNIX avem PF_UNIX .

Diversele arhitecturi de reţea folosesc şi diverse scheme de adresare. Pentru a păstra o interfaţă de programare unică, proiectanţii mecanismului de socluri au definit o structură generică[1]:

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

unde *sa_family* este identic cu tipul soclului. Există constantele cu prefix AF (de la address family), cum ar fi AF_INET sau AF_UNIX . Ele sunt identice ca valoare cu echivalentele lor cu prefix PF .

Câmpul *sa_data* conţine datele de adresare, în formatul specific diverselor familii de protocoale. Deşi lungimea este fixată aici la 14 octeţi, pentru fiecare arhitectură de reţea sunt definite structuri separate ce pot depăşi, unde este cazul, dimensiunea **struct** sockaddr . Din acest motiv, toate apelurile sistem şi funcţiile de bibliotecă ce au ca parametru o adresă vor mai avea un parametru, ce reprezintă lungimea structurii ce se transmite efectiv.

Toate aceste structuri au primul câmp, *sa_family*, identic. Pentru TCP/IP (v4) este definită următoarea structură:

[1] Această structură poate varia uşor, la rândul ei, de la un sistem la altul.

```
struct sockaddr_in {
  /* address family: AF_INET */
  sa_family_t     sin_family;
  /* port in network byte order */
  u_int16_t       sin_port;
  /* internet address */
  struct in_addr sin_addr;
};
/* Internet address. */
struct in_addr {
  /* address in network byte order */
  u_int32_t       s_addr;
};
```

Câmpul *sin_port* reprezintă portul TCP sau UDP, ce identifică procesul. Câmpul *sin_addr* este o structură de tip in_addr ce are un singur câmp, *s_addr*, care are 4 octeţi şi care reprezintă o adresă IP. Octeţii constituenţi atât ai porturilor, cât şi ai adreselor trebuie să fie ordonaţi în ordinea folosită de reţea („network byte order"). După cum se ştie, un număr întreg ce se reprezintă pe mai mulţi octeţi se poate păstra în memoria calculatorului în cel puţin două feluri: cu octeţii mai semnificativi la început (la adresele mai mici) sau la sfârşit (la adresele mai mari). Spre exemplu, numărul 515 (515 = 2 * 256 + 3), dacă îl reprezentăm pe doi octeţi, se poate păstra ca (2, 3) sau ca (3, 2). Numărul 2 este cel mai semnificativ octet, iar numărul 3 cel mai puțin semnificativ. Ambele reprezentări sunt folosite. De exemplu, platformele Intel folosesc ordinea cu cel mai puţin semnificativ octet înainte. Prin convenţie, în reţea numerele sunt reprezentate astfel încât cel mai semnificativ octet este primul. Prin urmare, este necesar să transformăm aceste numere din reprezentarea internă a maşinii în cea externă, a reţelei, atunci când transmitem numere şi invers atunci când le recepţionăm.

Există câteva funcţii de bibliotecă care ne ajută să facem acest lucru:

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Funcţiile htonl şi htons convertesc din formatul intern în cel extern (host to network long/short) numere reprezentate pe 4, respectiv pe 2 octeţi. Celelalte două funcţii realizează transformarea inversă.

1.3. Apeluri sistem utilizate cu *sockets*

Putem împărţi protocoalele de comunicaţie în două mari categorii: orientate pe conexiune sau fără conexiune. Această distincţie este importantă din mai multe puncte de vedere. Mai întâi, depinde de natura aplicaţiei dacă este bine să utilizăm un tip de protocol sau altul. Apoi, funcţiile şi apelurile de sistem diferă într-un caz şi în altul (deşi vom vedea că nu aşa de mult).

Apelul sistem socket

Acest apel sistem foloseşte la crearea unui soclu. Prototipul său este:

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

Parametrul *domain* reprezintă familia de protocoale şi pentru TCP/IP folosim constanta PF_INET . Al doilea parametru, *type*, desemnează tipul soclului. Pentru TCP/IP ne interesează valorile SOCK_STREAM care crează un socket pentru comunicaţii orientate pe conexiune şi SOCK_DGRAM pentru cele fără conexiune (datagrame).

Ultimul parametru selectează protocolul folosit. Există arhitecturi de reţea care au mai multe tipuri de protocoale orientate pe conexiune sau cu datagrame, oferind mai multe categorii de servicii. În cazul nostru, pentru TCP/IP avem doar câte un singur protocol pentru cele două tipuri de sockets: TCP pentru SOCK_STREAM şi UDP pentru SOCK_DGRAM . De aceea, putem folosi valoarea 0 pentru *protocol*, ceea ce înseamnă protocolul implicit pentru un tip anume de socket.

Apelul sistem socket întoarce un „socket descriptor", descriptor de care avem nevoie pentru a identifica soclul creat atunci când folosim alte apeluri sistem. În caz de eroare, socket întoarce valoarea -1 şi poziţionează corespunzător variabila globală errno .

Apelul sistem bind

După crearea unui soclu, acestuia nu îi corespunde nici o adresă. Se spune că el nu este legat. Cu ajutorul apelului sistem bind asociem unui soclu o adresă de maşină şi o adresă de proces (vezi **??**), sau în termeni TCP/IP o adresă IP şi un port.

```
int bind(int sockfd, struct sockaddr *my_addr, socklen_t
    addrlen);
```

Primul parametru este descriptorul de soclu. Al doilea, *my_addr*, este o structură ce reprezintă adresa pe care dorim să o asociem soclului, aşa cum am descris în secţiunea 1.2. Parametrul *addrlen* este lungimea în octeți a structurii **my_addr*.

În mod normal, adresa folosită în *my_addr* este fie adresa uneia din interfeţele de reţea ale staţiei pe care rulează procesul, fie INADDR_ANY . În acest ultim caz, adresa locală asociată soclului va fi „0.0.0.0", ceea ce înseamnă că el va fi legat la *toate* interfeţele. Un socket cu adresa locală INADDR_ANY poate recepţiona pachete şi conexiuni de reţea sosite către oricare din adresele staţiei, iar în cazul în care se iniţiază o conexiune sau se trimit datagrame cu adresa locală legată la „0.0.0.0", se va folosi adresa interfeţei de reţea prin care pachetele IP vor circula.

Parametrul *my_addr* conţine de asemenea şi portul la care dorim să legăm soclul. Valoarea 0 lasă sistemul de operare să aleagă un port pentru noi. Acest mod de lucru este folosit de mulţi clienţi, dar în general nu şi de servere. Există o discuţie puţin mai detaliată despre porturi în primul exemplu din secţiunea 1.4.

Apelul sistem connect

Realizarea unei legături cu un alt proces se face cu ajutorul apelului sistem cu prototipul de mai jos:

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

Semantica acestui apel sistem este diferită în funcţie de tipul soclului folosit. Pentru protocoalele orientate pe conexiune, rezultatul lui connect este stabilirea unei conexiuni între procesul apelant şi procesul aflat la

distanţă, proces care este determinat de parametrul *serv_addr*. Spre exemplu, pentru TCP are loc procesul descris în secţiunea **??**. connect se termină doar atunci când legătura a fost stabilită, sau în cazul în care apare o eroare.

Pentru protocoalele fără conexiune, cum ar fi UDP, nu se transmit nici un fel de mesaje procesului aflat la distanţă, ci doar se înregistrează valoarea *serv_addr* pentru a fi folosită atunci când procesul care a apelat connect scrie sau citeşte date prin apeluri sistem cum ar fi read sau write . Apelul sistem connect revine imediat în acest caz. Fără connect , aceste apeluri nu ar putea fi folosite întrucât sistemul nu ar şti cui trebuie să trimită datele.

Ultimul parametru specifică lungimea lui *serv_addr*.

Apelul sistem listen

Apelul sistem listen este folosit doar pentru protocoale orientate pe conexiune. El are prototipul următor:

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

În urma apelării lui listen , sistemul de operare ştie că procesul care l-a apelat doreşte să accepte conexiuni. Din acest moment, atunci când sosesc mesaje care stabilesc conexiuni către soclul *s*, acestea sunt memorate într-o coadă de aşteptare până când sunt acceptate (vezi apelul sistem accept , mai jos). Lungimea acestei cozi este dată de parametrul *backlog*. Dacă mai multe cereri de stabilire a unei conexiuni sosesc fără a fi acceptate şi numărul lor depăşeşte valoarea dată de *backlog*, cererile excedentare sunt ignorate sau respinse (funcţie de protocolul utilizat).

Apelul sistem accept

Acest apel sistem este folosit de către un proces atunci când doreşte să preia o cerere de la un proces client din coada de conexiuni (vezi listen mai sus). Fireşte, accept este folosit doar pentru protocoale orientate pe conexiune.

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

accept creează un nou socket şi întoarce descriptorul corespunzător. Acest nou socket este „conectat" la procesul client a cărui cerere de conexiune a fost preluată din coadă. În cazul în care o astfel de cerere nu există, accept se blochează în aşteptarea unei conexiuni.

Acest apel sistem va completa spaţiul dat prin parametrul *addr* cu adresa (şi portul) procesului client. Numărul maxim de octeţi pe care accept îl poate scrie la adresa *addr* este transmis prin parametrul *addrlen*. Acest parametru este la rândul lui un pointer (şi are ca valoare adresa unei variabile de tip socklen_t), deoarece accept va completa la adresa respectivă numărul de octeţi efectiv scrişi la adresa *addr*. Pentru ca lucrurile să fie mai clare, urmăriţi folosirea acestui apel sistem în exemplul 1 pe care îl dăm mai jos.

Apelurile sistem read şi write

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Aceste apeluri sistem sunt folosite în general pentru operaţii de intrare/ieşire. Ele pot fi folosite şi cu sockets, cu următoarele observaţii:

- nu pot fi folosite pentru sockets de tip SOCK_DGRAM (protocoale fără conexiune) decât în cazul în care s-a folosit în prealabil apelul sistem connect ;
- spre deosebire de operaţiile cu fişiere, atunci când sunt folosite cu sockets, read şi write se pot termina înainte ca toate datele solicitate să fie citite sau scrise (vezi subcapitolul 1.4).

Apelul sistem close

Apelul sistem close închide un descriptor.

```
int close(int fd);
```

Dacă tipul soclului care se închide este SOCK_STREAM , sistemul de operare va încerca să trimită datele din tampoane ce nu au fost încă trimise.

Apelurile sistem recvfrom şi sendto

recvfrom şi sendto sunt folosite pentru socluri de tip SOCK_DGRAM (protocoale fără conexiune). Ele sunt similare cu apelurile sistem read şi respectiv write , dar permit mai mulţi parametri.

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

Primii trei parametri – *s*, *buf* şi *len* – sunt echivalenți cu cei de la read şi write , cu excepţia faptului că *s* trebuie să fie un descriptor de soclu. Parametrul *from* este folosit pentru a indica adresa variabilei în care recvfrom va depozita structura **struct** sockaddr corespunzătoare procesului care a emis datagrama recepţionată şi care va conţine adresa şi portul sursă, pentru datagrame de tip UDP. Parametrul *to* al lui sendto este folosit pentru a transmite funcţiei datele de identificare ale procesului destinaţie. Pentru UDP, structura de tip **struct** sockaddr a cărei adresă este transmisă prin *to* va conţine adresa şi portul destinaţie a datagramei.

Pentru ambele apeluri sistem, parametrul *flags* permite setarea unor opţiuni (cum ar fi trimiterea sau recepţionarea de date *out-of-band*). Pentru mai multe detalii, studiaţi paginile de manual corespunzătoare.

Parametrii *fromlen* şi *tolen* dau lungimea structurilor adresate de *from* şi respectiv *to*. Observaţi că *fromlen* este un pointer, fiind un parametru folosit atât pentru transmiterea unei valori către recvfrom (lungimea maximă pentru structura **to*), cât şi pentru returnarea unei valori (lungimea efectivă a structurii **to*).

1.4. Exemple simple

Pentru a ilustra modul de utilizare a funcţiilor descrise în secţiunea 1.3, vom da două exemple simple, unul folosind un protocol orientat pe conexiune (TCP) iar celalalt folosind serviciile fără conexiune (UDP). Al treilea exemplu introduce modul de programare cu server concurent. Totodată, vom prezenta câteva funcţii ajutătoare, grupate într-o bibliotecă pe care am numit-o *netio*. Trei astfel de funcții vor fi descrise aici: set_addr , stream_read şi stream_write .

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <netdb.h>

#include "netio.h"

int set_addr(struct sockaddr_in *addr, char *name,
             u_int32_t inaddr, short sin_port) {
    struct hostent *h;

    memset((void *)addr, 0, sizeof(*addr));
    addr->sin_family = AF_INET;
    if (name != NULL) {
        h = gethostbyname(name);
        if (h == NULL)
            return -1;
        addr->sin_addr.s_addr =
          *(u_int32_t *) h->h_addr_list[0];
    } else
        addr->sin_addr.s_addr = htonl(inaddr);
    addr->sin_port = htons(sin_port);
    return 0;
}

int stream_read(int sockfd, char *buf, int len) {
    int nread;
    int remaining = len;

    while (remaining > 0) {
        if (-1 ==
            (nread = read(sockfd, buf, remaining)))
            return -1;
        if (nread == 0)
            break;
        remaining -= nread;
        buf += nread;
    }
    return len - remaining;
```

```
}

int stream_write(int sockfd, char *buf, int len) {
    int nwr;
    int remaining = len;

    while (remaining > 0) {
        if (-1 == (nwr = write(sockfd, buf, remaining)))
            return -1;
        remaining -= nwr;
        buf += nwr;
    }
    return len - remaining;
}
```

Prima dintre aceste trei funcţii, set_addr , ne ajută să completăm o structură de tip **struct** sockaddr_in . Primul argument este un pointer la structura sockaddr_in ce va fi completată de funcţie cu adresa şi portul specificate de restul argumentelor. cp este un pointer la un şir de caractere ce conţine adresa în format zecimal cu punct (*dotted-decimal*, de exemplu „192.168.5.23") sau un nume de domeniu DNS (de exemplu „foo.test.com"). Dacă cp *nu* este NULL, parametrul următor (s_addr) este ignorat, iar dacă este NULL, adresa va fi luată din s_addr , care trebuie să aibă octeţii în ordinea maşinii (*host byte order*). Acest ultim caz este efectiv folosit în exemplele date doar atunci când dorim să utilizăm adresa INADDR_ANY .

Ultimul argumet, port , reprezintă portul şi trebuie să fie reprezentat în ordinea maşinii.

Funcţiile stream_read şi stream_write sunt similare apelurilor de sistem read şi write . Ele sunt necesare în cazul folosirii protocolului TCP (prin SOCK_STREAM), deoarece este posibil ca read şi write să citească sau să scrie un număr mai mic de octeţi decât cel specificat ca argument, fără ca acest fapt să constituie o eroare. Acest lucru se întâmplă, de exemplu, atunci când tampoanele din nucleul sistemului de operare sunt depăşite. Funcţiile stream_read şi stream_write verifică numărul de octeţi scrişi sau citiţi şi în cazul în care acesta este mai mic decât cel cerut prin argumentul len , apelează din nou read respectiv write pentru datele rămase (de mai multe ori, dacă este cazul, până când toate datele au fost scrise sau citite).

Trebuie avut în vedere faptul că exemplele de mai jos sunt doar nişte exemple şi nu aplicaţii reale, deoarece ele nu tratează o mulţime de situaţii. Multe teste de eroare au fost omise, mai ales în primele exemple, pentru claritatea codului. De asemenea, tratarea semnalelor lipseşte în cele mai multe cazuri cu desăvârşire. Protocoalele descrise şi implementate nu sunt suficient de riguroase. Acele programe care creează procese noi nu apelează wait pentru a prelua starea fiilor. Atunci când se scriu programe reale, trebuie să se ţină cont de nenumărate aspecte legate de tratarea erorilor, interacţiunea cu utilizatorii și cu sistemul de operare. Nu ne-am propus însă ca în spaţiul acestei cărţi să abordăm astfel de aspecte.

1.4.1. Exemplul 1: transfer de fişiere

Vom începe cu o aplicaţie ce transferă fişiere de la un sistem la altul. Deoarece este important ca fişierul transferat să coincidă exact cu fişierul original, este oportun să folosim protocolul TCP (un *stream socket*).

Exemplul acesta, cât şi celelalte exemple, cuprinde două programe: un client şi un server. Acest lucru nu este întâmplător, deoarece majoritatea aplicaţiilor pentru reţele de calculatoare sunt construite în acest mod.

După cum am precizat anterior, programele prezentate nu conţin toate testele necesare. În acest prim exemplu nu avem teste aproape deloc, pentru ca cititorul să poată urmări cât mai uşor folosirea apelurilor sistem şi funcţiilor de bibliotecă.

Programul „server" este următorul:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <fcntl.h>
#include "netio.h"

#define SERVER_PORT     5678

int main(void) {
    int fd, sockfd, connfd;
    char buf[1024];
```

```
struct sockaddr_in local_addr, rmt_addr;
int nread, nfis;
socklen_t rlen;
char file_name[10];

sockfd = socket(PF_INET, SOCK_STREAM, 0);
set_addr(&local_addr, NULL, INADDR_ANY,
         SERVER_PORT);
bind(sockfd, (struct sockaddr *)&local_addr,
     sizeof(local_addr));
listen(sockfd, 5);
nfis = 1;
while (nfis < 100) {
    rlen = sizeof(rmt_addr);
    connfd =
      accept(sockfd, (struct sockaddr *)&rmt_addr,
             &rlen);
    /* scrie un fisier nou */
    snprintf(file_name, 10, "fisier%.3d", nfis);
    fd =
      open(file_name, O_WRONLY | O_CREAT | O_TRUNC,
           00644);
    if (fd == -1) {
        printf("Nu pot scrie fisierul %s\n",
               file_name);
        exit(1);
    }
    while (0 <
           (nread =
            stream_read(connfd, (void *)buf,
                        1024))) {
        write(fd, (void *)buf, nread);
    }
    nfis++;
    if (nread < 0)
        printf("Eroare la citirea de la retea\n");
    else
        printf("%s receptionat\n", file_name);
    close(connfd);
    close(fd);
}
```

```
    close(sockfd);
    exit(0);
}
```

Serverul se leagă la adresa 0.0.0.0 (INADDR_ANY), ceea ce înseamnă că acceptă conexiuni pe oricare din adresele maşinii locale şi foloseşte portul 5678. Această valoare nu este total arbitrară: ea trebuie să fie mai mare de 1023 şi este bine să fie mai mare de 5000. Porturile sub 1024 se numesc privilegiate şi nu pot fi folosite decât de superuser (root) şi sunt folosite în general de serviciile standard, cum ar fi e-mail-ul sau FTP. Programele care lasă sistemul să le aloce un port (specificând valoarea 0) primesc în mod convenţional[2] porturi cu valori între 1024 şi 4999. De aici nevoia de a aloca pentru server un port cu valoare peste 5000, pentru ca să nu nimerim întâmplător peste un port alocat deja.

În continuare, programul aşteaptă conexiuni de la clienţi care urmează să transfere câte un fişier, până la limita de 100 de conexiuni. Atunci când se stabileşte o nouă conexiune, serverul crează un fişier nou, în directorul curent, cu numele „fisierXXX" (unde XXX este 001, 002 etc.), în care copiază tot ceea ce primeşte prin conexiunea cu clientul. Observaţi că apelul accept creează un soclu nou pentru fiecare conexiune acceptată. Este important ca, în momentul în care nu mai dorim să folosim o conexiune, să o închidem cu apelul close (sau cu shutdown), pentru a elibera porturile şi alte resurse folosite în sistem, cât şi pentru a semnala capătului celuilalt al conexiunii faptul că nu mai dorim să folosim legatura în continuare. Atunci când terminăm un proces prin apelul de sistem exit , se închid toţi descriptorii deschişi, aşa că nu mai este necesar să folosim explicit close .

Programul „client" se leagă la o adresă locală arbitrară şi se conectează la SERVER_ADDRESS . Am utilizat valoarea „127.0.0.1", ceea ce semnifică adresa de loopback, pentru a putea experimenta pe un acelaşi calculator. Această adresă va trebui înlocuită cu adresa maşinii pe care rulează serverul.

```
#include <stdio.h>
#include <unistd.h>
```

[2]În unele distribuţii Linux (RedHat), porturile alocate aplicaţiilor sunt implicit între 32768 şi 61000. Alte distribuţii, cum ar fi Debian, se comportă „normal". Diverse sisteme de operare pot să adere sau nu la convenţia descrisă. Acest comportament se poate modifica, în Linux, scriind fişierul */proc/sys/net/ipv4/ip_local_port_range* sau prin sysctl modificând variabila *net.ipv4.ip_local_port_range*.

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include "netio.h"

#define SERVER_ADDRESS "127.0.0.1"
#define SERVER_PORT    5678

int main(int argc, char *argv[]) {
    int fd, sockfd;
    char buf[1024];
    struct sockaddr_in local_addr, remote_addr;
    int nread;

    if (argc != 2) {
        printf("Folosire: %s <fisier>\n", argv[0]);
        exit(1);
    }
    fd = open(argv[1], O_RDONLY);
    sockfd = socket(PF_INET, SOCK_STREAM, 0);
    set_addr(&local_addr, NULL, INADDR_ANY, 0);
    bind(sockfd, (struct sockaddr *)&local_addr,
         sizeof(local_addr));
    set_addr(&remote_addr, SERVER_ADDRESS, 0,
             SERVER_PORT);
    connect(sockfd, (struct sockaddr *)&remote_addr,
            sizeof(remote_addr));
    /* trimite fisierul */
    while (0 < (nread = read(fd, (void *)buf, 1024))) {
        stream_write(sockfd, (void *)buf, nread);
    }
    if (nread < 0) {
        printf("Eroare la citirea din fisier\n");
        exit(1);
    }
    close(sockfd);
    close(fd);
```

```
    printf("Fisierul a fost trimis cu succes\n");
    exit(0);
}
```

După conectare, clientul trimite serverului fişierul al cărui nume îi este dat ca parametru.

1.4.2. Exemplul 2: utilizarea datagramelor

Utilizarea datagramelor, prin protocolul UDP, este asemănătoare, din punct de vedere al funcţiilor şi apelurilor sistem folosite, cu *stream sockets* (protocolul TCP). Nu mai este obligatoriu să folosim connect , dar este util, pentru a nu fi nevoiţi să precizăm de fiecare dată destinaţia datagramelor. Pentru acest exemplu, programul server așteaptă date, până la o lungime maximă stabilită prin MAXBUF , pe care le afişează la terminal.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include "netio.h"

#define SERVER_PORT     5678
#define MAXBUF          2048

int main(void) {
    int sockfd;
    char buf[MAXBUF];
    struct sockaddr_in local_addr;
    int nread;

    if (-1 == (sockfd = socket(PF_INET, SOCK_DGRAM, 0))) {
        printf("Eroare la socket()\n");
        exit(1);
    }
    set_addr(&local_addr, NULL, INADDR_ANY,
             SERVER_PORT);
```

```
    if (-1 ==
        bind(sockfd, (struct sockaddr *)&local_addr,
             sizeof(local_addr))) {
        printf("Eroare la bind()\n");
        exit(1);
    }
    while (0 < (nread = read(sockfd, &buf, MAXBUF))) {
        printf("%.*s\n", nread, buf);
    }
    exit(0);
}
```

Programul client trimite şirul de caractere „abcd" mai întâi printr-o singură operaţie write iar ulterior prin două apeluri sistem. Observaţi că serverul va afişa:

```
abcd
ab
cd
```

Acest lucru se datorează faptului că apelul sistem read , atunci când este folosit cu socluri de tip SOCK_DGRAM , revine în momentul în care se recepţionează o datagramă. La rândul lui, apelul sistem write , folosit cu socluri de tip SOCK_DGRAM , va determina sistemul de operare să trimită o datagramă pentru fiecare apel al său.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "netio.h"

#define SERVER_ADDRESS "127.0.0.1"
#define SERVER_PORT    5678

int main(void) {
```

```
    int sockfd;
    struct sockaddr_in local_addr, remote_addr;

    if (-1 == (sockfd = socket(PF_INET, SOCK_DGRAM, 0))) {
        printf("Eroare la socket()\n");
        exit(1);
    }
    set_addr(&local_addr, NULL, INADDR_ANY, 0);
    if (-1 ==
        bind(sockfd, (struct sockaddr *)&local_addr,
             sizeof(local_addr))) {
        printf("Eroare la bind()\n");
        exit(1);
    }
    set_addr(&remote_addr, SERVER_ADDRESS, 0,
             SERVER_PORT);
    if (-1 ==
        connect(sockfd, (struct sockaddr *)&remote_addr,
                sizeof(remote_addr))) {
        printf("Eroare la connect()\n");
        exit(1);
    }
    write(sockfd, "abcd", strlen("abcd"));
    write(sockfd, "ab", strlen("ab"));
    write(sockfd, "cd", strlen("cd"));
    exit(0);
}
```

1.4.3. Exemplul 3: server concurent

Am văzut în primul exemplu un server care preia fişiere de la clienţi aflaţi la distanţă, fişiere pe care le scrie pe disc. Dacă aţi avut curiozitatea să rulaţi mai mulţi clienţi deodată, aţi constatat fie că transferurile nu se efectuează decât pe rând, fie chiar că sunt refuzate unele conexiuni, dacă numărul de clienţi este mai mare. Acest lucru se datorează modului în care este scris programul server. El deserveşte o singură conexiune la un moment dat. Acest mod de lucru se numeşte *server iterativ*. Dacă dorim ca serverul să deservească mai mulţi clienţi simultan, el trebuie scris într-o manieră ce se numeşte *server concurent*. De fapt, acesta este motivul pentru care apelul sistem `accept` creează un nou soclu. În momentul în care se acceptă

o nouă conexiune, serverul concurent creează un nou proces care va deservi clientul, iar procesul original (părinte) continuă să „asculte” soclul original şi să preia noi cereri.

Mai întâi avem un fişier antet, comun pentru toate programele din acest exemplu:

```
#ifndef _EX3_H
/* *INDENT-OFF* */
#define _EX3_H

#define EX3_SUCCESS      0
#define EX3_GOAHEAD      1
#define EX3_BYE          2
#define EX3_READERR      3
#define EX3_LONGLINE     4
#define EX3_FILECREA     5
#define EX3_FILEWRERR    6
#define EX3_EARLYEOF     7
#define EX3_FNAMNOTSET   8
#define EX3_INVCMD       9

int readline(int, char *, int);
/* *INDENT-ON* */
#endif
```

Programul server este următorul:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <string.h>
#include "netio.h"
#include "ex3.h"

#define SERVER_PORT    5678
#define BUFSIZE        1024
```

```
/* MAXCMD trebuie sa fie mai mic sau egal cu BUFSIZE */
#define MAXCMD          300

#if MAXCMD > BUFSIZE
#    error "MAXCMD prea mare"
#endif

char *errorcodes[] = {
    "00 OK\r\n",
    "01 Da-i drumul\r\n",
    "02 La revedere\r\n",
    "03 Eroare de citire din retea\r\n",
    "04 Linia este prea lunga\r\n",
    "05 Nu am putut crea fisierul\r\n",
    "06 Nu am putut scrie fisierul\r\n",
    "07 Conexiunea s-a terminat prematur\r\n",
    "08 Numele fisierului nu este dat\r\n",
    "09 Comanda necunoscuta\r\n"
};

static inline void reply(int connfd, int code) {
    (void)write(connfd, errorcodes[code],
                strlen(errorcodes[code]));
}

void getfile(int connfd, char *file_name, char *buf) {
    int fd, nread;

    fd =
      open(file_name, O_WRONLY | O_CREAT | O_TRUNC,
           00644);
    if (fd == -1) {
        reply(connfd, EX3_FILECREA);
        return;
    }
    reply(connfd, EX3_GOAHEAD);
    /* citeste continutul fisierului */
    while (0 <
           (nread =
            stream_read(connfd, (void *)buf, 1024))) {
        if (-1 == write(fd, (void *)buf, nread)) {
```

```
            reply(connfd, EX3_FILEWRERR);
            return;
        }
    }
    close(fd);
    if (nread < 0)
        reply(connfd, EX3_READERR);
    else
        reply(connfd, EX3_SUCCESS);
    return;
}

void ex3_proto(int connfd) {
    int ret, n;
    char buf[BUFSIZE];
    char *file_name = NULL;
    char *cmd;

    do {
        ret = readline(connfd, buf, MAXCMD);
        if (ret != EX3_SUCCESS) {
            reply(connfd, ret);
            return;
        }
        cmd = buf;
        /* treci peste spatiile albe initiale */
        n = strspn(cmd, " \t\r\n");
        if (strlen(cmd) == n) {
            /* rand gol */
            continue;
        }
        cmd += n;
        n = strcspn(cmd, " \t\r\n");
        if (0 == strncmp(cmd, "quit", n)) {
            reply(connfd, EX3_BYE);
            return;
        }
        if (0 == strncmp(cmd, "filename", n)) {
            cmd += n + 1;
            file_name = (char *)
              malloc(strlen(cmd) + 1);
```

```
            strcpy(file_name, cmd);
            reply(connfd, EX3_SUCCESS);
            continue;
        }
        if (0 == strncmp(cmd, "data", n)) {
            if (!file_name) {
                reply(connfd, EX3_FNAMNOTSET);
                continue;
            }
            getfile(connfd, file_name, buf);
            return;
        }
        /* comanda necunoscuta */
        reply(connfd, EX3_INVCMD);
    } while (1);
}

int main(void) {
    int sockfd, connfd;
    struct sockaddr_in local_addr, remote_addr;
    socklen_t rlen;
    pid_t pid;

    if (-1 ==
        (sockfd = socket(PF_INET, SOCK_STREAM, 0))) {
        printf("Nu am putut crea soclul\n");
        exit(1);
    }
    set_addr(&local_addr, NULL, INADDR_ANY,
             SERVER_PORT);
    if (-1 ==
        bind(sockfd, (struct sockaddr *)&local_addr,
             sizeof(local_addr))) {
        printf("Eroare la bind()\n");
        exit(1);
    }
    if (-1 == listen(sockfd, 5)) {
        printf("Eroare la listen()\n");
        exit(1);
    }
    rlen = sizeof(remote_addr);
```

```
    while (1) {
        connfd =
          accept(sockfd,
                 (struct sockaddr *)&remote_addr,
                 &rlen);
        if (connfd < 0) {
            printf("Eroare la accept()\n");
            exit(1);
        }
        pid = fork();
        switch (pid) {
        case -1:
            printf("Eroare la fork()\n");
            exit(1);
        case 0:                 /* proces fiu */
            close(sockfd);
            ex3_proto(connfd);
            exit(0);
        default:                /* proces parinte */
            close(connfd);
        }
    }
    /* aici nu ar trebui sa ajung */
    exit(0);
}
```

Funcţia readline este definită într-un program separat, deoarece este utilizată şi în client:

```
#include <unistd.h>
#include "ex3.h"

#define DELIM1 '\r'
#define DELIM2 '\n'

int readline(int connfd, char *buf, int maxlen) {
    char *pos, *last;
#ifdef DELIM2
    int flag = 0;
#endif

    for (pos = buf, last = buf + maxlen; pos < last;
```

```
          pos++)
        switch (read(connfd, (void *)pos, 1)) {
        case -1:                /* eroare */
            return EX3_READERR;
        case 0:                 /* conexiune incheiata */
            return EX3_EARLYEOF;
        default:
            if (*pos == DELIM1) {
#ifdef DELIM2
                flag = 1;
                break;
            }
            if (*pos == DELIM2) {
                if (!flag)
                    break;
                *(pos - 1) = '\0';
#else
                *pos = '\0';
#endif
                return EX3_SUCCESS;
            }
#ifdef DELIM2
            if (flag)
                flag = 0;
#endif
            break;
        }
    /* numele fisierului e prea lung */
    return EX3_LONGLINE;
}
```

Observaţi modul în care se preia o comandă de la server. Aceasta se citeşte caracter cu caracter (octet cu octet), până când se întâlneşte *newline* sau se depăşeşte un număr dat de octeţi. De ce nu cerem dintr-o dată mai mulţi octeţi? De exemplu:

```
nread = read (connfd, buf, MAXBUF);
```

O problemă ar fi că s-ar putea întâmpla să citim mai mult decât o singură linie deodată. De fapt, aceasta nu este adevărata problemă. Am

prelucra şirul de caractere până la *newline* după care am păstra restul pentru linia următoare. Într-adevăr deranjant ar fi ca serverul să se blocheze aşteptând mai multe caractere decât are linia trimisă. Fiind o aplicaţie interactivă, acest fapt ar constitui o problemă, deoarece nu am primi răspunsul la o comandă imediat.

Felul în care este rezolvată problema în codul nostru nu este însă ideal. Pentru fiecare caracter efectuăm un apel sistem `read`, care necesită o schimbare de context între modul utilizator şi modul nucleu, fapt care ia destul de mult timp. În mod obişnuit acest lucru nu este ceva deosebit de grav, dar pot exista situaţii unde să dorim să reducem cât mai mult încărcarea sistemului. Astfel de situaţii sunt, spre exemplu, acelea în care sistemul folosit este lent sau încărcat, sau atunci când dorim să obţinem performanţe ridicate. În secţiunea 1.5.1 vom da o altă soluţie.

Un alt element nou în acest exemplu este existenţa unui protocol de comunicaţie între client şi server. Desigur, acest protocol, de nivel aplicaţie, este foarte simplu. De asemenea, am putea spune că şi în cazul primului exemplu există un protocol aplicaţie, dar ar fi exagerat.

Protocolul din acest exemplu ar putea fi descris în felul următor:

- (Clientul deschide conexiunea, iar serverul o acceptă.)
- Serverul intră în modul comandă. În acest mod, serverul așteaptă linii text (caractere ASCII ce se termină cu *newline*) ce reprezintă comenzi.
- Clientul trimite astfel de comenzi şi preia un cod de stare de la server.
- Pentru fiecare comandă, serverul răspunde printr-o linie de stare.

O comandă este formată dintr-un şir de litere urmat eventual de argumente. Argumentele sunt despărţite între ele şi de comandă prin exact un caracter spaţiu[3]. Comanda se termină prin caracterul *newline*. O linie de stare este formată din două caractere, pe primele poziţii din linie, care formează un cod de stare, un caracter spaţiu şi un text explicativ aferent. Codul de stare este numeric, cu valori între 0 şi 99 şi se reprezintă prin două caractere ASCII între „0” şi „9”. Scopul urmărit este acela de a putea tipări linia de stare direct pe ecran, motiv pentru care s-a inclus și textul explicativ. Astfel, operatorul uman poate urmări mai uşor ce se întâmplă.

[3]De fapt, codul dat acceptă mai multe spaţii sau caractere de tabulare orizontală

Acest lucru este inspirat din protocolul SMTP folosit la transferul mesajelor electronice.

Un alt beneficiu al alegerii acestui protocol este acela că putem folosi comanda `telnet` *`adresă port`* pentru a verifica serverul sau chiar pentru a-l utiliza la crearea fişierelor text.

Comenzile acceptate sunt:

filename {*nume-fişier*} Setează numele fişierului, aşa cum va fi creat pe maşina serverului. Serverul răspunde prin codul „00” (OK) sau „04” (numele fişierului este prea lung).

quit Termină conexiunea.

data Trece serverul în modul recepţie date până când conexiunea este închisă. Clientul transferă conținutul fişierului, după care închide conexiunea.

Codurile de eroare pe care le-am utilizat sunt redate în tabelul următor.

00	OK (confirmarea succesului unei operaţii)
01	Da-i drumul (clientul poate începe să transfere datele fişierului)
03	Eroare de citire din reţea
04	Linia este prea lungă
05	Nu am putut crea fişierul
06	Nu am putut scrie fişierul
07	Conexiunea s-a terminat prematur
08	Numele fişierului nu este dat
09	Comandă necunoscută

Tabela 1.1: Coduri de eroare pentru exemplul 3

Observaţi din nou că o serie de teste ce în mod normal ar trebui să apară într-un program real au fost omise. Astfel, ar trebui verificate caracterele ce constituie numele fişierului şi mai ales dacă acesta conţine `/` sau `..`, deoarece acestea reprezintă căi de fișiere şi pot crea probleme.

Programul client trebuie apelat cu două sau trei argumente. Primul argument reprezintă adresa sau numele staţiei pe care rulează serverul. Al doilea este numele fişierului ce se doreşte a fi transferat. Al treilea argument este opţional şi reprezintă numele cu care să salveze serverul fişierul transmis de client. În cazul în care nu se precizează un al treilea argument, clientul va transmite serverului numele fişierului local, dat în al doilea argument.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <string.h>
#include "netio.h"
#include "ex3.h"

#define SERVER_PORT     5678
#define MAXBUF          1024

int main(int argc, char *argv[]) {
    int fd, sockfd;
    char buf[MAXBUF];
    struct sockaddr_in local_addr, remote_addr;
    int nread, ret;
    char *delimitator = "\r\n";
    int ack;

    if (argc < 3 || argc > 4) {
        printf("%s adresa fisier [nume]\n", argv[0]);
        exit(1);
    }
    if (-1 == (fd = open(argv[2], O_RDONLY))) {
        printf("Nu am putut deschide fisierul %s\n",
                argv[1]);
        exit(1);
    }
    if (-1 ==
```

```
    (sockfd = socket(PF_INET, SOCK_STREAM, 0))) {
    printf("Eroare la socket()\n");
    exit(1);
}
set_addr(&local_addr, NULL, INADDR_ANY, 0);
if (-1 ==
    bind(sockfd, (struct sockaddr *)&local_addr,
         sizeof(local_addr))) {
    printf("Eroare la bind()\n");
    exit(1);
}
set_addr(&remote_addr, argv[1], 0, SERVER_PORT);
if (-1 ==
    connect(sockfd, (struct sockaddr *)&remote_addr,
            sizeof(remote_addr))) {
    printf("Conectarea la server a esuat\n");
    exit(1);
}
/* trimite numele fisierului */
snprintf(buf, MAXBUF, "filename %s%s",
         argv[argc - 1], delimitator);
stream_write(sockfd, (void *)buf, strlen(buf));
/* asteapta confirmarea */
ret = readline(sockfd, buf, MAXBUF);
if (ret != EX3_SUCCESS) {
    printf("Client: Eroare raspuns\n");
    exit(1);
}
ack = atoi(buf);
switch (ack) {
case EX3_SUCCESS:
    break;
default:
    printf("%s\n", buf);
    exit(1);
}
snprintf(buf, MAXBUF, "data%s", delimitator);
stream_write(sockfd, (void *)buf, strlen(buf));
ret = readline(sockfd, buf, MAXBUF);
if (ret != EX3_SUCCESS) {
    printf("Client: Eroare raspuns\n");
```

```
        exit(1);
    }
    ack = atoi(buf);
    switch (ack) {
    case EX3_GOAHEAD:
        break;
    default:
        printf("%s\n", buf);
        exit(1);
    }
    /* trimite fisierul */
    while (0 < (nread = read(fd, (void *)buf, MAXBUF))) {
        stream_write(sockfd, (void *)buf, nread);
    }
    if (nread < 0) {
        printf("Client:Eroare citire din fisier\n");
        exit(1);
    }
    shutdown(sockfd, SHUT_WR);
    close(fd);
    readline(sockfd, buf, MAXBUF);
    printf("%s\n", buf);
    exit(0);
}
```

1.4.4. Exerciţii

1. Exemplul 2 transferă şiruri de caractere predefinite. El nu este deosebit de util în această formă. Rescrieţi exemplul astfel încât serverul şi clientul să poată fi folosiţi de către doi utilizatori pentru a schimba mesaje. Pentru aceasta, atât serverul cât şi clientul vor citi intrarea standard şi vor trimite liniile citite prin reţea. Programul aflat la distanţă va prelua din reţea aceste linii de text şi le va afişa la ieşirea standard. Faceţi în aşa fel încât mesajele afişate să nu perturbeze utilizatorii care scriu la rândul lor mesaje!

2. Modificaţi exemplul 3 astfel încât serverul să accepte comanda *size*. Această comandă va specifica lungimea fişierului ce se doreşte transferat. Serverul va citi exact atâţia octeţi după comada *data*, după care trece din nou în modul comandă, de unde va putea trimite un nou

fişier. Rezultatul trebuie să fie posibilitatea de a trimite mai multe fişiere pe aceeaşi conexiune.

3. Scrieţi o aplicaţie distribuită care monitorizează cantitatea de date transferate în reţea pe un număr de staţii de lucru. Serverul va prelua datele de la mai mulţi clienţi şi va prezenta situaţia la cerere sau periodic prin afişarea numărului total de octeţi transferaţi de către toţi clienţii monitorizaţi şi a listei acelor 5 staţii de lucru care au utilizat rețeaua mai intens.

   Datele despre utilizarea reţelei le puteţi obţine, pentru sistemele Linux, din fişierul `/proc/net/dev`.

4. Creaţi un dicţionar accesibil prin reţea. Serverul va servi definiţiile pentru cuvintele cerute de către client. Definiţi un protocol în stilul folosit în exemplul 3 astfel încât serverul să poată fi folosit şi cu telnet pe post de client. Implementaţi comenzi pentru a primi definiţia unui cuvânt, pentru a adăuga un nou cuvânt la dicţionar, pentru a căuta un subşir de caractere şi pentru ștergerea unui cuvânt din lista de cuvinte.

5. Scrieţi un program server şi un client corespunză tor care să descarce prin reţea o ierarhie de directoare, cu fişierele din ea.

1.5. Aplicaţii

1.5.1. O aplicaţie cu fire de execuţie

Să presupunem că dorim să strângem într-un singur loc, pe un server, informaţiile de jurnalizare de la mai multe programe aflate pe calculatoare diferite. Să scriem deci o aplicaţie ce constă dintr-un server ce va culege linii de text de la mai mulţi clienţi şi le va scrie într-un fişier (jurnal) şi dintr-un client care trimite prin reţea serverului liniile ce îi parvin de la alte programe de pe același calculator cu el. Un program ce va dori să scrie în jurnal se va lega la intrarea standard a clientului.

Avem un număr destul de mare de clienţi ce se vor conecta la server. Cu un server concurent ce utilizează câte un proces pentru fiecare conexiune, s-ar putea întâmpla să încărcăm destul de mult sistemul pe care rulează. O variantă prin care să reducem această încărcare este să folosim, în locul proceselor, fire de execuţie.

Firele de execuţie au însă şi dezavantaje. Complexitatea programului server poate fi ceva mai ridicată decât atunci când folosim procese distincte. Mai grav, firele de execuţie nu sunt deosebit de portabile.

Atenţie! Atunci când un program lucrează cu fire de execuţie, trebuie să includeţi în faza de editare de legături şi biblioteca *pthreads*:

```
gcc -o loom -lpthread loom.c
```

Serverul aplicaţiei noastre este următorul:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <string.h>
#include <pthread.h>
#include "netio.h"

#define BUFSZ           1024
#define SERVER_PORT     (short)5678

pthread_mutex_t mutex;
int threadNumber = 0;
int fd;

int readline_init(int sfd) {
    int val;
    socklen_t len = sizeof(int);

    getsockopt(sfd, SOL_SOCKET, SO_RCVLOWAT, &val,
               &len);
    if (val != 1) {              /* incearca sa setezi */
        len = sizeof(int);
        setsockopt(sfd, SOL_SOCKET, SO_RCVLOWAT,
                   (void *)&val, len);
        getsockopt(sfd, SOL_SOCKET, SO_RCVLOWAT, &val,
                   &len);
```

```
        if (val != 1)            /* nu pot seta valoarea
                                  */
            return -1;
    }
    return 0;
}

int readline(int connfd, char *line, int *idx,
             char *buf, int maxlen) {
    int ret, n;

    for (; *idx < maxlen; *idx += ret) {
        ret = read(connfd, buf + *idx, maxlen - *idx);
        if (ret <= 0)
            return ret;
        for (n = *idx; n < *idx + ret; n++)
            if (buf[n] == '\n') {
                /* copiaza linia */
                memcpy(line, buf, n + 1);
                /* muta ce a mai ramas */
                memmove(buf, buf + n + 1, ret - n - 1);
                *idx = ret - n - 1;
                return n + 1;
            }
    }
    /* s-a depasit tamponul */
    return -1;
}

/* functia urmatoare este corpul fiecarui fir de
 * executie */
void *ex4_proto(void *arg) {
    int ret;
    char line[BUFSZ];
    char buf[BUFSZ];
    int idx = 0;
    int *connfd = (int *)arg;

    while (0 <
           (ret =
            readline(*connfd, line, &idx, buf,
```

```
                    BUFSZ))) {
        write(fd, buf, ret);     /* scrie linia in
                                  * jurnal */
    }
    close(*connfd);
    free(connfd);
    fsync(fd);                   /* sincronizeaza cu
                                  * discul */
    pthread_mutex_lock(&mutex);
    threadNumber--;
    printf("%d fire active\n", threadNumber);
    pthread_mutex_unlock(&mutex);
    return NULL;
}

int main(int argc, char *argv[]) {
    int sockfd, *connfd;
    struct sockaddr_in local_addr, remote_addr;
    socklen_t rlen;
    pthread_t thread;
    pthread_attr_t attr;
    char *file_name = "journal";

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, 1);
    pthread_mutex_init(&mutex, NULL);

    if (argc > 2) {
        printf("Utilizare: %s fisier\n", argv[0]);
        exit(1);
    }
    if (argc == 2)
        file_name = argv[2];
    fd =
      open(file_name, O_CREAT | O_APPEND | O_WRONLY,
           00644);
    if (fd == -1) {
        printf("Nu am putut deschide jurnalul\n");
        exit(1);
    }
    if (-1 ==
```

```
    (sockfd = socket(PF_INET, SOCK_STREAM, 0))) {
    printf("Nu am putut crea soclul\n");
    exit(1);
}
if (-1 == readline_init(sockfd)) {
    printf("Initializarea readline a esuat\n");
    exit(1);
}
set_addr(&local_addr, NULL, INADDR_ANY,
        SERVER_PORT);
if (-1 ==
    bind(sockfd, (struct sockaddr *)&local_addr,
         sizeof(local_addr))) {
    printf("Eroare la bind()\n");
    exit(1);
}
if (-1 == listen(sockfd, 5)) {
    printf("Eroare la listen()\n");
    exit(1);
}
rlen = sizeof(remote_addr);
while (1) {
    connfd = (int *)malloc(sizeof(int));
    if (connfd == NULL) {
        printf("Eroare la malloc()\n");
        exit(1);
    }
    *connfd =
      accept(sockfd,
             (struct sockaddr *)&remote_addr,
             &rlen);
    if (*connfd < 0) {
        printf("Eroare la accept()\n");
        exit(1);
    }
    if (0 !=
        pthread_create(&thread, &attr, ex4_proto,
                       (void *)connfd)) {
        printf("Nu am putut crea fir nou\n");
        exit(1);
    }
```

```
        pthread_mutex_lock(&mutex);
        threadNumber++;
        printf("%d fire active\n", threadNumber);
        pthread_mutex_unlock(&mutex);
    }
    exit(0);
}
```

Fiecare client este servit de un fir de execuţie distinct. Firele de execuție sunt create astfel încât să fie detaşate, ceea ce înseamnă că la terminarea lor resursele folosite se distrug imediat. Observaţi că am transmis descriptorul de socket într-un mod „ciudat". Funcţia pe care o va executa un *thread* trebuie să aibă un parametru de tip **void** $*$. Întâmplător, un pointer se reprezintă tot pe 4 octeţi ca şi un **int** . Evident *arg* nu va fi folosit ca şi adresă ci valoarea lui va fi tratată ca întreg. Acest artificiu simplifică transmiterea parametrului în cazul de faţă, dar nu poate fi folosit oricând.

Am modificat şi modul în care citim o linie de text din reţea. Felul în care este scrisă funcţia readline se bazează pe o caracteristică anume a apelurilor sistem de citire, atunci când sunt invocate pentru socluri. Astfel, o operaţie read (valabil pentru toate apelurile sistem de citire) nu se va bloca dacă există date disponibile. Numărul minim de octeţi care vor fi aşteptaţi de sistem poate fi controlat pe unele sisteme prin opţiunea SO_RCVLOWAT [4]. Această opţiune se poate citi şi scrie cu getsockopt respectiv cu setsockopt . Nu toate implementările posedă o astfel de opţiune şi de asemenea nu toate implementările permit şi setarea unei valori pe lângă citirea ei. Implicit, valoarea lui SO_RCVLOWAT este 1 ceea ce înseamnă ca read nu se va bloca dacă există cel puţin un octet disponibil.

Un client ce citeşte linii de la intrarea standard şi le transmite serverului de jurnalizare este prezentat mai jos:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
```

[4] Numele vine de la Receive Low Watermark

```
#include <arpa/inet.h>
#include "netio.h"

#define SERVER_PORT     5678
#define MAXBUF          1024

int main(int argc, char *argv[]) {
    int sockfd;
    char buf[MAXBUF];
    struct sockaddr_in local_addr, remote_addr;

    if (argc != 2) {
        printf("%s adresa \n", argv[0]);
        exit(1);
    }
    if (-1 ==
        (sockfd = socket(PF_INET, SOCK_STREAM, 0))) {
        printf("Eroare la socket()\n");
        exit(1);
    }
    set_addr(&local_addr, NULL, INADDR_ANY, 0);
    if (-1 ==
        bind(sockfd, (struct sockaddr *)&local_addr,
             sizeof(local_addr))) {
        printf("Eroare la bind()\n");
        exit(1);
    }
    if (-1 ==
        set_addr(&remote_addr, argv[1], 0,
                 SERVER_PORT)) {
        printf("Eroare de adresa\n");
        exit(1);
    }
    if (-1 ==
        connect(sockfd, (struct sockaddr *)&remote_addr,
                sizeof(remote_addr))) {
        printf("Conectarea la server a esuat\n");
        exit(1);
    }
    while (fgets(buf, MAXBUF, stdin)) {
        if (-1 ==
```

```
            stream_write(sockfd, (void *)buf,
                         strlen(buf))) {
            printf("Eroare la scriere\n");
            exit(1);
        }
    }
    exit(0);
}
```

1.5.2. Distribuitor de mesaje. Programare orientată pe evenimente

Cel mai eficient, dar şi cel mai complex mod de a scrie un program ce lucrează simultan cu mai multe fluxuri de date este acela de a utiliza apelul sistem `select`. Acesta verifică pentru un set de descriptori dacă operaţiile de scriere sau citire[5] pot fi efectuate imediat, fără blocare. Cu alte cuvinte, `select` aşteaptă evenimente ce constă în schimbarea stării acelor descriptori. După fiecare apel sistem `select`, trebuie să verificăm toţi descriptorii interesanţi (care fac parte din set), iar dacă starea lor s-a modificat, să efectuăm operaţiile necesare.

Programul de mai jos transmite datagramele UDP sosite, pe un port local, unui client a cărui adresă şi port sunt date în linia de comandă. De asemenea, programul se detaşază de terminal devenind *daemon*. Pentru aceasta execută o serie de operaţii, sugerate în [Ste90].

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <syslog.h>
```

[5] Se definesc seturi distincte pentru operaţia de citire şi pentru cea de scriere. Se pot verifica de asemenea şi situaţii excepţionale, cum ar fi *out-of-band data* la sockets.

```
#include <fcntl.h>
#include <errno.h>
#include "netio.h"

#define BUFFSZ  1500

/* seturile de descriptori urmariti */
fd_set rd_set, wr_set;
int sfd;

void sigterm_handler(int s) {
    syslog(LOG_INFO, "ex5_server stopped");
    exit(0);
}

void daemonize(void) {
    int i, maxfd;
    int fd;

    maxfd = getdtablesize();

    for (i = 0; i < maxfd; i++)
        close(i);

    chdir("/");

    switch (fork()) {
    case -1:
        syslog(LOG_ERR, "Eroare la fork()\n");
    case 0:                     /* fiu */
        break;
    default:                    /* parinte */
        exit(0);
    }

    setpgid(0, 0);

    fd = open("/dev/tty", O_RDWR);

    if (fd >= 0) {
        ioctl(fd, TIOCNOTTY);
```

```
        close(fd);
        syslog(LOG_INFO, "ex5_server started");
    }
}

void main_loop(struct sockaddr_in remote) {
    int ret;
    char buf[BUFFSZ];
    char *rpos = buf;
    char *wpos = buf;
    char *last = buf + BUFFSZ;
    int free = BUFFSZ;
    int avail = 0;

    for (;;) {
        FD_ZERO(&rd_set);
        FD_ZERO(&wr_set);
        if (free)
            FD_SET(sfd, &rd_set);
        if (avail)
            FD_SET(sfd, &wr_set);
        ret = select(sfd + 1, &rd_set, &wr_set,
                     NULL, NULL);
        if (ret == -1) {
            if (errno == EINTR)
                continue;
            syslog(LOG_ERR, "error at select()");
            exit(1);
        }

        if (FD_ISSET(sfd, &wr_set)) {
            ret =
              sendto(sfd, wpos, avail, 0,
                     (void *)&remote, sizeof(remote));
            if (-1 == ret) {
                syslog(LOG_ERR, "error write()ing");
            }

            avail -= ret;
            wpos += ret;
```

```
            if (avail == 0 && wpos == last) {
                wpos = rpos = buf;
                free = BUFFSZ;
            }
        }
        if (FD_ISSET(sfd, &rd_set)) {
            ret = read(sfd, rpos, free);

            if (-1 == ret) {
                syslog(LOG_ERR, "error read()ing");
                exit(1);
            }

            free -= ret;
            avail += ret;
            rpos += ret;
        }
    }
}

int main(int argc, char *argv[]) {
    int port_l, port_r;
    struct sockaddr_in local;
    struct sockaddr_in remote;

    if (argc != 4) {
        printf("Utilizare: %s port_l adresa port_r\n",
               argv[0]);
        exit(1);
    }

    port_l = atoi(argv[1]);
    port_r = atoi(argv[3]);

    if (port_l < 0 || port_r < 0) {
        printf("Port incorect");
        exit(1);
    }

    daemonize();
```

```
    signal(SIGTERM, sigterm_handler);
    signal(SIGPIPE, SIG_IGN);

    openlog("ex5", 0, LOG_WARNING);

    sfd = socket(PF_INET, SOCK_DGRAM, 0);

    if (sfd == -1) {
        syslog(LOG_ERR, "Could not create socket");
        exit(1);
    }

    set_addr(&local, NULL, INADDR_ANY, port_l);

    if (-1 ==
        bind(sfd, (struct sockaddr *)&local,
             sizeof(local))) {
        syslog(LOG_ERR, "Could not bind to local");
        exit(1);
    }

    if (-1 == set_addr(&remote, argv[2], 0, port_r)) {
        syslog(LOG_ERR, "Wrong address");
        exit(1);
    }

    main_loop(remote);

    exit(0);
}
```

Pentru a utiliza programul de mai sus, puteţi fie să scrieţi un client corespunzător, fie să folosiţi programul *netcat*[6] pentru a trimite şi recepționa datagrame:

```
./server 5000 localhost 6000
netcat -l -u -p 6000
```

Apoi, în alt terminal:

```
netcat -u <adresa server> 5000
```

[6]În distribuţia RedHat el se numeşte *nc*.

Bibliografie

[Ste90] W. Richard Stevens. *UNIX Network Programming.* Prentice Hall, Inc., 1990.