Acesta este capitolul 8 — *Programarea în reţea — introducere* — al ediţiei electronică a cărţii *Reţele de calculatoare*, publicată la Casa Cărţii de Ştiinţă, în 2008, ISBN: 978-973-133-377-9.

Cartea, integrală, poate fi descărcată gratuit de la adresa
`http://www.cs.ubbcluj.ro/~rlupsa/works/retele.pdf`

Capitolul 8

Programarea în reţea — introducere

8.1. Interfaţa de programare *socket BSD*

Interfaţa *socket* este un ansamblu de funcţii sistem utilizate de programe (de fapt, de procese) pentru a comunica cu alte procese, aflate în execuţie pe alte calculatoare. Interfaţa *socket* a fost dezvoltată în cadrul sistemului de operare BSD (sistem de tip UNIX, dezvoltat la Universitatea Berkley) — de aici denumirea de *socket BSD*. Interfaţa socket este disponibilă în aproape toate sistemele de operare actuale.

Termenul *socket* se utilizează atât pentru a numi ansamblul funcţiilor sistem legate de comunicaţia prin reţea, cât şi pentru a desemna fiecare capăt al unei conexiuni deschise în cadrul reţelei.

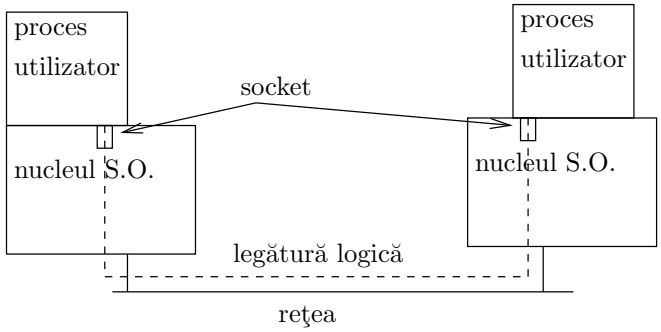

Figura 8.1: Comunicaţia între două procese prin reţea

Prezentăm în continuare principiile de bază ale interfeţei *socket* (vezi

şi figura 8.1):

- Pe fiecare calculator rulează mai multe procese şi fiecare proces poate avea mai multe căi de comunicaţie deschise. Prin urmare, pe un calculator trebuie să poată exista la un moment dat mai multe legături (conexiuni) active.
- Realizarea comunicării este intermediată de sistemele de operare de pe calculatoarele pe care rulează cele două procese. Deschiderea unei conexiuni, închiderea ei, transmiterea sau recepţionarea de date pe o conexiune şi configurarea parametrilor unei conexiuni se fac de către sistemul de operare, la cererea procesului. Cererile procesului se fac prin apelarea funcţiilor sistem din familia *socket*.
- În cadrul comunicaţiei dintre procesul utilizator şi sistemul de operare local (prin intermediul apelurilor din familia *socket*), capetele locale ale conexiunilor deschise sunt numite *socket*-uri şi sunt identificate prin numere întregi, unice în cadrul unui proces la fiecare moment de timp.
- Fiecare entitate care comunică în cadrul reţelei este identificat printr-o *adresă* unică. O adresa este asociată de fapt unui socket. Adresa este formată conform regulilor protocolului de reţea utilizat.
- Interfaţa socket conţine funcţii pentru comunicaţiei atât conform modelului conexiune cât şi conform modelului cu datagrame.
- Funcţiile sistem oferite permit stabilirea comunicaţiei prin diferite protocoale (de exemplu, IPv4, IPv6, IPX), dar au aceeaşi sintaxă de apel independent de protocolul dorit.

8.1.1. Comunicaţia prin conexiuni

În cele ce urmează, prin *client* desemnăm procesul care solicită în mod activ deschiderea conexiunii către un partener de comunicaţie specificat printr-o *adresă*, iar prin *server* înţelegem procesul care aşteaptă în mod pasiv conectarea unui *client*.

Vom da în cele ce urmează o scurtă descriere a operaţiilor pe care trebuie să le efectueze un proces pentru a deschide o conexiune şi a comunica prin ea. Descrierea este împărţită în patru părţi: deschiderea conexiunii de către client, deschiderea conexiunii de către server, comunicaţia propriu-zisă şi închiderea conexiunii.

O descriere mai amănunţită a funcţiilor sistem apelate şi a parametrilor mai des utilizaţi este făcută separat (§ 8.1.3), iar pentru detalii suplimentare se recomandă citirea paginilor corespunzătoare din documentaţia on-line.

8.1.1.1. Deschiderea conexiunii de către client

Procesul client trebuie să ceară mai întâi sistemului de operare local crearea unui socket. Trebuie specificat protocolul de reţea utilizat (TCP/IPv4, TCP/IPv6, etc), dar încă nu se specifică partenerul de comunicaţie. Socket-ul proaspăt creat este în starea *neconectat*.

După crearea socket-ului, clientul cere sistemului de operare conectarea socket-ului la un anumit server, specificat prin adresa *socket*-ului serverului. De exemplu, pentru protocolul TCP/IPv4, adresa partenerului se specifică prin adresa IP (vezi § 10.1) şi numărul portului (vezi § 10.3.1).

Funcţiile sistem apelate sunt: `socket()` pentru crearea socket-ului şi `connect()` pentru deschiderea efectivă a conexiunii.

8.1.1.2. Deschiderea conexiunii de către server

Procesul server începe tot prin a cere sistemului de operare crearea unui socket de tip conexiune pentru protocolul dorit. Acest socket nu va servi pentru conexiunea propriu-zisă cu un client, ci doar pentru aşteptarea conectării clienţilor; ca urmare este numit uneori *socket de aşteptare*. După crearea acestui socket, serverul trebuie să ceară sistemului de operare stabilirea adresei la care serverul aşteaptă cereri de conectare (desigur, acea parte din adresă care identifică maşina serverului nu este la alegerea procesului server) şi apoi cere efectiv începerea aşteptării clienţilor. Funcţiile apelate în această fază sunt, în ordinea în care trebuie apelate: `socket()` pentru crearea socket-ului, `bind()` pentru stabilirea adresei şi `listen()` pentru începerea aşteptării clienţilor.

Preluarea efectivă a unui client conectat se face prin apelarea unei funcţii sistem numită `accept()`. La apelul funcţiei `accept()`, sistemul de operare execută următoarele:

- aşteaptă cererea de conectare a unui client şi deschide conexiunea către acesta;
- crează un nou socket, numit *socket de conexiune*, care reprezintă capătul dinspre server al conexiunii proaspăt deschise;
- returnează apelantului (procesului server) identificatorul socket-ului de conexiune creat.

După un apel `accept()`, socket-ul de aşteptare poate fi utilizat pentru a aştepta noi clienţi, iar socket-ul de conexiune nou creat se utilizează pentru a comunica efectiv cu acel client.

8.1.1.3. Comunicaţia propriu-zisă

O dată deschisă conexiunea, clientul poate trimite şiruri de octeţi către server şi invers, serverul poate trimite şiruri de octeţi către client. Cele două sensuri de comunicaţie funcţionează identic (nu se mai distinge cine a fost client şi cine a fost server) şi complet independent (trimiterea datelor pe un sens nu este condiţionată de recepţionarea datelor pe celălalt sens).

Pe fiecare sens al conexiunii, se poate transmite un şir arbitrar de octeţi. Octeţii trimişi de către unul dintre procese spre celălalt sunt plasaţi într-o coadă, transferaţi prin reţea la celălalt capăt şi citiţi de către procesul de acolo. Comportamentul acesta este similar cu cel al unui *pipe* UNIX.

Trimiterea datelor se face prin apelul funcţiei `send()` (sau, cu funcţionalitate mai redusă, `write()`). Apelul acestor funcţii plasează datele în coadă spre a fi transmise, dar nu aşteaptă transmiterea lor efectivă (returnează, de principiu, imediat controlul către procesul apelant). Dacă dimensiunea datelor din coadă este mai mare decât o anumită valoare prag, (aleasă de sistemele de operare de pe cele două maşini), apelul `send()` se blochează, returnând controlul procesului apelant abia după ce partenerul de comunicaţie citeşte date din coadă, ducând la scăderea dimensiunii datelor din coadă sub valoarea prag.

Recepţionarea datelor trimise de către partenerul de comunicaţie se face prin intermediul apelului sistem `recv()` (cu functionalitate mai redusă se poate utiliza `read()`). Aceste funcţii returnează procesului apelant datele deja sosite pe calculatorul receptor şi le elimină din coadă. În cazul în care nu sunt încă date disponibile, ele aşteaptă sosirea a cel puţin un octet.

Sistemul garantează sosirea la destinaţie a tuturor octeţilor trimişi (sau înştiinţarea receptorului, printr-un cod de eroare, asupra căderii conexiunii), în ordinea în care au fost trimişi. Nu se păstrează însă demarcarea între secvenţele de octeţi trimise în apeluri `send()` distincte. De exemplu, este posibil ca emiţătorul să trimită, în două apeluri succesive, şirurile `abc` şi `def`, iar receptorul să primească, în apeluri `recv()` succesive, şirurile `ab`, `cde` şi `f`.

8.1.1.4. Închiderea conexiunii

Închiderea conexiunii se face separat pentru fiecare sens şi pentru fiecare capăt. Există două funcţii:

- `shutdown()` închide, la capătul local al conexiunii, sensul de comunicaţie cerut de procesul apelant;
- `close()` închide la capătul local ambele sensuri de comunicaţie şi în plus distruge socket-ul, eliberând resursele alocate (identificatorul de socket

şi memoria alocată în spaţiul nucleului).

Terminarea unui proces are efect identic cu un apel `close()` pentru toate socket-urile existente în acel moment în posesia acelui proces.

Dacă capătul de emisie al unui sens de comunicaţie a fost închis, receptorul poate citi în continuare datele existente în acel moment în coadă, după care un eventual apel `recv()` va semnaliza apelantului faptul că a fost închisă conexiunea.

Dacă capătul de recepţie al unui sens a fost închis, o scriere ulterioară de la celălalt capăt este posibil să returneze un cod de eroare (pe sistemele de tip UNIX, scrierea poate duce şi la primirea, de către procesul emiţător, a unui semnal SIGPIPE).

8.1.2. Comunicaţia prin datagrame

În comunicaţia prin datagrame, datagramele sunt transmise independent una de cealaltă şi fiecare datagramă are o adresă sursă, o adresă destinaţie şi nişte date utile. Un proces ce doreşte să trimită sau să primească datagrame trebuie mai întâi să creeze un *socket* de tip *dgram*; un astfel de socket conţine în principal adresa de reţea a procesului posesor al socket-ului. După crearea unui socket, se poate cere sistemului de operare să asocieze socket-ului o anumită adresă sau se poate lăsa ca sistemul de operare să-i atribuie o adresă liberă arbitrară. Crearea unui socket se face prin apelul funcţiei `socket()`, iar atribuirea unei adrese se face prin apelul `bind()`.

O dată creat un socket, procesul poate trimite datagrame de pe acel socket, prin apelul funcţiei `sendto()`. Datagramele trimise vor avea ca adresă sursă adresa socket-ului şi ca adresă destinaţia şi conţinut util valorile date ca parametri funcţiei `sendto()`. De pe un socket se pot trimite, succesiv, oricâte datagrame şi oricâtor destinatari.

Datagramele emise sunt transmise către sistemul de operare al destinatarului, unde sunt memorate în buffer-ele sistemului. Destinatarul poate citi o datagramă apelând funcţia `recvfrom()`. Această funcţie ia următoarea datagramă adresată socket-ului dat ca parametru la `recvfrom()` şi o transferă din buffer-ele sistemului local în memoria procesului apelant. Funcţia oferă apelantului conţinutul datagramei (datele utile) şi, separat, adresa expeditorului datagramei. În ciuda numelui, `recvfrom()` nu poate fi instruită să ia în considerare doar datagramele expediate de la o anumită adresă.

Sistemul nu garantează livrarea tuturor datagramelor (este posibilă pierderea unor datagrame) şi nici nu oferă vreun mecanism de informare a expeditorului în cazul unei pierderi. Mai mult, există posibilitatea (e drept, rară) ca o datagamă să fie duplicată (să ajungă două copii la destinatar) şi

este posibil ca două sau mai multe datagrame adresate aceluiaşi destinatar să ajungă la destinaţie în altă ordine decât cea în care au fost emise. Dacă astfel de situaţii sunt inadmisibile pentru aplicaţie, atunci protocolul de comunicaţie trebuie să prevadă confirmări de primire şi repetarea datagramelor pierdute, precum şi numere de secvenţă sau alte informaţii pentru identificarea ordinii corecte a datagramelor şi a duplicatelor. Implementarea acestor mecanisme cade în sarcina proceselor.

La terminarea utilizării unui socket, procesul posesor poate cere distrugerea socket-ului şi eliberarea resurselor asociate (identificatorul de socket, memoria ocupată în sistemul de operare pentru datele asociate socket-ului, portul asociat socket-ului). Distrugerea socket-ului se face prin apelul funcţiei `close()`.

În mod curent, într-o comunicaţie prin datagrame, unul dintre procese are rol de *client*, în sensul că trimite cereri, iar celălalt acţionează ca *server*, în sensul că prelucrează cererile clientului şi trimite înapoi clientului răspunsurile la cereri. Într-un astfel de scenariu, serverul crează un socket căruia îi asociază o adresă prestabilită, după care aşteaptă cereri, apelând în mod repetat `recvfrom()`. Clientul crează un socket, căruia nu-i asociază o adresă (nu execută `bind()`). Clientul trimite apoi cererea sub forma unei datagrame de pe socket-ul creat. La trimiterea primei datagrame, sistemul de operare dă o adresă socket-ului; datagrama emisă poartă ca adresă sursă acestă adresă. La primirea unei datagrame, serverul recuperează datele utile şi adresa sursă, procesează cererea şi trimite răspunsul către adresa sursă a cererii. În acest fel, răspunsul este adresat exact socket-ului de pe care clientul a trimis cererea. Clientul obţine răspunsul executând `recvfrom()` asupra socket-ului de pe care a expediat cererea.

Cu privire la tratarea datagramelor pierdute, un caz simplu este acela în care clientul pune doar întrebări (interogări) serverului, iar procesarea interogării nu modifică în nici un fel starea serverului. Un exemplu tipic în acest sens este protocolul DNS (§ 10.4). În acest caz, datagrama cerere conţine interogarea şi daatgrama răspuns conţine atât cererea la care se răspunde cât şi răspunsul la interogare. Serverul ia (în mod repetat) câte o cerere, calculează răspunsul şi trimite o înapoi o datagramă cu cererea primită şi răspunsul la cerere. Clientul trimite cererile sale şi aşteaptă răspunsurile. Deoarece fiecare răspuns conţine în el şi cererea, clientul poate identifica fiecare răspuns la ce cerere îi corespunde, chiar şi în cazul inversării ordinii datagramelor. Dacă la o cerere nu primeşte răspuns într-un anumit interval de timp, clientul repetă cererea; deoarece procesarea unei cereri nu modifică starea serverului, duplicarea cererii de către reţea sau repetarea cererii de către client ca urmare a pierderii

răspunsului nu au efecte nocive. Clientul trebuie să ignore răspunsurile duplicate la o aceeaşi interogare.

8.1.3. Principalele apeluri sistem

8.1.3.1. Funcţia `socket()`

Funcţia are sintaxa:

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

Funcţia crează un socket şi returnează identificatorul său. Parametrii sunt:

- `type`: desemnează tipul de servicii dorite:
  - `SOCK_STREAM`:conexiune punct la punct, flux de date bidirecţional la nivel de octet, asigurând livrare sigura, cu pastrarea ordinii octeţilor şi transmisie fara erori.
  - `SOCK_DGRAM`:datagrame, atât punct la punct cât şi difuziune; transmisia este garantată a fi fără erori, dar livrarea nu este sigura şi nici ordinea datagramelor garantata.
  - `SOCK_RAW`:acces la protocoale de fivel coborât; este de exemplu utilizat de către comanda `ping` pentru comunicaţie prin protocolul ICMP.
- `proto_family` identifică tipul de reţea cu care se lucrează (IP, IPX, etc). Valori posibile:
  - `PF_INET`:protocol Internet, versiunea 4 (IPv4)
  - `PF_INET6`:protocol Internet, versiunea 6 (IPv6)
  - `PF_UNIX`:comunicaţie locală pe o maşină UNIX.
- `protocol` selectează protocolul particular de utilizat. Acest parametru este util dacă pentru un tip de reţea dat şi pentru un tip de serviciu fixat există mai multe protocoale utilizabile. Valoarea 0 desemnează protocolul implicit pentru tipul de reţea şi tipul de serviciu alese.

8.1.3.2. Funcţia `connect()`

Funcţia are sintaxa:

```
int connect(int sock_id, struct sockaddr* addr, int addr_len)
```

Funcţia are ca efect conectarea socketului identificat de primul parametru — care trebuie să fie un socket de tip conexiune proaspăt creat (încă neconectat)

— la serverul identificat prin adresă dată prin parametrii `addr` şi `addr_len`. La adresa respectivă trebuie să existe deja un server care să aştepte conexiuni (să fi fost deja executat apelul `listen()` asupra socket-ului serverului).

Adresa trebuie plasată, înainte de apelul `connect()`, într-o structură având un anumit format; conţinutul acestei structuri va fi descris în § 8.1.3.6. Adresa în memorie a acestei structuri trebuie dată ca parametrul `addr`, iar lungimea structurii de adresă trebuie dată ca parametrul `addr_len`. Motivul acestei complicaţii este legat de faptul că funcţia `connect()` trebuie să poată lucra cu formate diferite de adresă, pentru diferite protocoale, iar unele protocoale au adrese de lungime variabilă.

Funcţia `connect()` returnează 0 în caz de succes şi -1 în caz de eroare. Eroarea survenită poate fi constatată fie verificând valoarea variabilei globale `errno`, fie apelând funcţia `perror()` imediat după funcţia sistem ce a întâmpinat probleme. Eroarea cea mai frecventă este lipsa unui server care să asculte la adresa specificată.

8.1.3.3. Funcţia `bind()`

```
int bind(int sd, struct sockaddr* addr, socklen_t len)
```

Funcţia are ca efect atribuirea adresei specificate în parametrul `addr` socket-ului identificat prin identificatorul `sd`. Această funcţie se apelează în mod normal dintr-un proces server, pentru a pregăti un socket *stream* de aşteptare sau un socket *dgram* pe care se aşteaptă cereri de la clienţi.

Partea, din adresa de atribuit socket-ului, ce conţine adresa maşinii poate fi fie una dintre adresele maşinii locale, fie valoarea specială `INADDR_ANY` (pentru IPv4) sau `IN6ADDR_ANY_INIT` (pentru IPv6). În primul caz, socket-ul va primi doar cereri de conexiune (sau, respectiv, pachete) adresate adresei IP date socket-ului, şi nu şi cele adresate altora dintre adresele maşinii server.

Exemplul 8.1: Să presupunem că maşina server are adresele 193.226.40.130 şi 127.0.0.1. Dacă la apelul funcţiei `bind()` se dă adresa IP 127.0.0.1, atunci socket-ul respectiv va primi doar cereri de conectare destinate adresei IP 127.0.0.1, nu şi adresei 193.226.40.130. Dimpotrivă, dacă adresa acordată prin `bind()` este `INADDR_ANY`, atunci socket-ul respectiv va accepta cereri de conectare adresate oricăreia dintre adresele maşinii locale, adică atât adresei 193.226.40.130 cât şi adresei 127.0.0.1.

Adresa atribuită prin funcţia `bind()` trebuie să fie liberă în acel moment. Dacă în momentul apelului `bind()` există un alt socket de acelaşi tip având aceeaşi adresă, apelul `bind()` eşueaza.

Pe sistemele de tip UNIX, pentru atribuirea unui număr de port mai mic decât 1024 este necesar ca procesul apelant să ruleze din cont de administrator.

Funcţia `bind()` poate fi apelată doar pentru un socket proaspăt creat, căruia nu i s-a atribuit încă o adresă. Aceasta înseamnă că funcţia `bind()` nu poate fi apelată de două ori pentru acelaşi socket. De asemenea, funcţia `bind()` nu poate fi apelată pentru un socket de conexiune creat prin funcţia `accept()` şi nici pentru un socket asupra căruia s-a apelat în prealabil vreuna dintre funcţiile `connect()`, `listen()` sau `sendto()` — aceste funcţii având ca efect atribuirea unei adrese libere aleatoare.

Funcţia returnează 0 în caz de succes şi -1 în caz de eroare. Eroarea cea mai frecventă este că adresa dorită este deja ocupată.

8.1.3.4. Funcţia `listen()`

```
int listen(int sd, int backlog)
```

Funcţia cere sistemului de operare să accepte, din acel moment, cererile de conexiune pe adresa socket-ului `sd`. Dacă socketului respectiv nu i s-a atribuit încă o adresă (printr-un apel `bind()` anterior), funcţia `listen()` îi atribuie o adresă aleasă aleator.

Parametrul `backlog` fixează dimensiunea cozii de aşteptare în acceptarea conexiunilor. Anume, vor putea exista `backlog` clienţi care au executat `connect()` fără ca serverul să fi creat încă pentru ei socket-uri de conexiune prin apeluri `accept()`. De notat că nu există nici o limitare a numărului de clienţi conectaţi, preluaţi deja prin apelul `accept()`.

8.1.3.5. Funcţia `accept()`

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

Apelul funcţiei `accept()` are ca efect crearea unui socket de conexiune, asociat unui client conectat (prin apelul `connect()`) la socket-ul de aşteptare `sd`. Dacă nu există încă nici un client conectat şi pentru care să nu se fi creat socket de conexiune, funcţia `accept()` aşteaptă până la conectarea următorului client.

Funcţia returnează identificatorul socket-ului de conexiune creat.

Dacă procesul server nu doreşte să afle adresa clientului, va da valori `NULL` parametrilor `addr` şi `addrlen`. Dacă procesul server doreşte să afle adresa clientului, atunci va trebui să aloce spaţiu pentru o structură pentru memorarea adresei clientului, să pună adresa structurii respective în parametrul

`addr`, să plaseze într-o variabilă de tip întreg dimensiunea memoriei alocate pentru adresa clientului şi să pună adresa acestui întreg în parametrul `adrlen`. În acest caz, la revenirea din apelul `accept()`, procesul server va găsi în structura de adresă adresa socket-ului client şi în variabila întreagă a cărui adresă a fost dată în parametrul `adrlen` va găsi dimensiunea efectiv utilizată de sistemul de operare pentru a scrie adresa clientului.

8.1.3.6. Formatul adreselor

Pentru funcţiile socket ce primesc de la apelant (ca parametru) o adresă din reţea (`bind()`, `connect()` şi `sendto()`), precum şi pentru cele ce returnează apelantului adrese de reţea (`accept()`, `recvfrom()`, `getsockname()` şi `getpeername()`), sunt definite structuri de date (`struct`) în care se plasează adresele socket-urilor.

Pentru ca funcţiile de mai sus să poată avea aceeaşi sintaxă de apel independent de tipul de reţea (şi, în consecinţă, de structura adresei), funcţiile primesc adresa printr-un pointer la zona de memorie ce conţine adresa de reţea. Structura zonei de memorie respective depinde de tipul reţelei utilizate. În toate cazurile, aceasta începe cu un întreg pe 16 biţi reprezentând tipul de reţea.

Dimensiunea structurii de date ce conţine adresa de reţea depinde de tipul de reţea şi, în plus, pentru anumite tipuri de reţea, dimensiunea este variabilă. Din acest motiv:

- funcţiile care primesc de la apelant o adresă (`connect()`, `bind()` şi `sendto()`) au doi parametri: un pointer către structura de adresă şi un întreg reprezentând dimensiunea acestei structuri;
- funcţiile care furnizează apelantului o adresă (`accept()`, `recvfrom()`, `getsockname()` şi `getpeername()`) primesc doi parametri: un pointer către structura de adresă şi un pointer către o variabilă de tip întreg pe care apelantul trebuie s-o iniţializeze, înaintea apelului, cu dimensiunea pe care a alocat-o pentru structura de adresă şi în care funcţia pune, în timpul apelului, dimensiunea utilizată efectiv de astuctura de adresă.

În ambele cazuri, parametrul pointer către structura de adresă este declarat ca fiind de tip `struct sockaddr*`. La apelul acestor funcţii este necesară conversia a pointer-ului către structura de adresă de la pointer-ul specific tipului de reţea la `struct sockaddr*`.

O adresă a unui capăt al unei conexiuni TCP sau a unei legături prin datagrame UDP este formată din adresa IP a maşinii şi numărul de port (vezi § 10.2.3.1, § 10.3.1.6 şi § 10.3.2). Pentru nevoile funcţiilor de mai sus, adresele socket-urilor TCP şi UDP se pun, în funcţie de protocolul de nivel

reţea (IPv4 sau IPv6), într-o structură de tip `sockaddr_in` pentru IPv4 sau `sockaddr_in6` pentru IPv6.

Pentru adrese IPv4 este definită structura `sockaddr_in` având următorii membrii:

`sin_family`:trebuie să conţină constanta `AF_INET`;

`sin_port`:de tip întreg de 16 biţi (2 octeţi), fără semn, în ordine reţea (cel mai semnificativ octet este primul), reprezentând numărul portului;

`sin_addr`:conţine adresa IP. Are tipul `struct in_addr`, având un singur câmp, `s_addr`, de tip întreg pe 4 octeţi în ordine reţea.

Adresa IPv4 poate fi convertită de la notaţia obişnuită (notaţia zecimală cu puncte) la `struct in_addr` cu ajutorul funcţiei

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

Conversia inversă, de la structura `in_addr` la string în notaţie zecimală cu punct se face cu ajutorul funcţiei

```
char *inet_ntoa(struct in_addr in);
```

care returnează rezultatul într-un buffer static, apelantul trebuind să copieze rezultatul înainte de un nou apel al functiei.

Pentru adrese IPv6 este definită structura `sockaddr_in6` având următorii membrii:

`sin6_family`:trebuie să conţină constanta `AF_INET6`;

`sin6_port`:de tip întreg de 16 biţi (2 octeţi), fără semn, în ordine reţea (cel mai semnificativ octet este primul), reprezentând numărul portului;

`sin6_flow`:eticheta de flux.

`sin6_addr`:conţine adresa IP. Are tipul `struct in6_addr`, având un singur câmp, `s6_addr`, de tip tablou de 16 octeţi.

Obţinerea unei adrese IPv4 sau IPv6 cunoscând numele de domeniu (vezi § 10.4) se face cu ajutorul funcţiei

```
struct hostent *gethostbyname(const char *name);
```

care returnează un pointer la o structură ce conţine mai multe câmpuri dintre care cele mai importante sunt:

`int h_addrtype`:tipul adresei, `AF_INET` sau `AF_INET6`;

`char **h_addr_list`:pointer la un şir de pointeri către adresele IPv4 sau IPv6 ale maşinii cu numele `name`, în formatul `in_addr` sau respectiv `in6_addr`;

`int h_length`:lungimea şirului `h_addr_list`.

8.1.3.7. Interacţiunea dintre `connect()`, `listen()` şi `accept()`

La apelul `connect()`, sistemul de operare de pe maşina client trimite maşinii server o cerere de conectare. La primirea cererii de conectare, sistemul de operare de pe maşina server acţionează astfel:

- dacă adresa din cerere nu corespunde unui socket pentru care s-a efectuat deja apelul `listen()`, refuză conectarea;
- dacă adresa corespunde unui socket pentru care s-a efectuat `listen()`, încearcă plasarea clientului într-o coadă de clienţi conectaţi şi nepreluaţi încă prin `accept()`. Dacă plasarea reuşeşte (coada fiind mai mică decât valoarea parametrului `backlog` din apelul `listen()`), sistemul de operare trimite sistemului de operare de pe maşina client un mesaj de acceptare; în caz contrar trimite un mesaj de refuz.

Apelul `connect()` revine în procesul client în momentul sosirii acceptului sau refuzului de la sistemul de operare de pe maşina server. Revenirea din apelul `connect()` nu este deci condiţionată de apelul `accept()` al procesului server.

Apelul `accept()` preia un client din coada descrisă mai sus. Dacă coada este vidă în momentul apelului, funcţia aşteaptă sosirea unui client. Dacă coada nu este vidă, apelul `accept()` returnează imediat.

Parametrul `backlog` al apelului `listen()` se referă la dimensiunea cozii de clienţi conectaţi (prin `connect()`) şi încă nepreluaţi prin `accept()`, şi nu la clienţii deja preluaţi prin `accept()`.

8.1.3.8. Funcţiile `getsockname()` şi `getpeername()`

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

Funcţia `getsockname()` furnizează apelantului adresa socket-ului `sd`. Funcţia `getpeername()`, apelată pentru un socket de tip conexiune deja conectat, furnizează adresa partenerului de comunicaţie.

Funcţia `getsockname()` este utilă dacă un proces acţionează ca server, creînd în acest scop un socket de aşteptare, dar numărul portul pe care aşteaptă conexiunile nu este prestabilit ci este transmis, pe altă cale, viitorilor client. În acest caz, procesul server crează un socket (apelând `socket()`),

cere primirea cererilor de conexiune (apelând `listen()`, dar fără a fi apelat `bind()`) după care determină, prin apelul `getsockname()`, adresa atribuită la `listen()` socket-ului respectiv şi transmite această adresă viitorilor clienţi.

8.1.3.9. Funcţiile `send()` şi `recv()`

Apelurile sistem `send()` şi `recv()` sunt utilizate în faza de comunicaţie pentru socket-uri de tip conexiune. Descriem în continuare utilizarea acestor funcţii considerând un singur sens de comunicaţie şi ca urmare ne vom referi la un *proces emiţător* şi un *proces receptor* în raport cu sensul considerat. De notat însă că o conexiune *socket stream* este bidirecţională şi comunicarea în cele două sensuri se desfăşoară independent şi prin aceleaşi mecanisme.

Sintaxa funcţiilor este:

```
ssize_t send(int sd, const void *buf, size_t len, int flags);
ssize_t recv(int sd, void *buf, size_t len, int flags);
```

Funcţia `send()` trimite pe conexiunea identificată prin socket-ul `sd` un număr de `len` octeţi din variabila a cărui adresă este indicată de pointer-ul `buf`. Funcţia returnează controlul după plasarea datelor de transmis în buffer-ele sistemului de operare al maşinii locale. Valoarea returnată de funcţia `send()` este numărul de octeţi scrişi efectiv, sau -1 în caz de eroare. Datele plasate în buffer-e prin apelul `send()` urmează a fi trimise spre receptor fără alte acţiuni din partea emiţătorului.

În modul normal de lucru, dacă nu există spaţiu suficient în buffer-ele sistemului de operare, funcţia `send()` aşteaptă ca aceste buffer-e să se elibereze (prin transmiterea efectivă a datelor către sistemul de operare al receptorului şi citirea lor de către procesul receptor). Această aşteptare are ca rol frânarea procesului emiţător dacă acesta produce date la un debit mai mare decât cel cu care este capabilă reţeaua să le transmită sau procesul receptor să le preia. Prin plasarea valorii `MSG_DONTWAIT` în parametrul `flags`, acest comportament este modificat. Astfel, în acest caz, dacă nu există suficient spaţiu în buffer-ele sistemului de operare, funcţia `send()` scrie doar o parte din datele furnizate şi returnează imediat controlul procesului apelant. În cazul în care funcţia `send()` nu scrie nimic, ea returnează valoarea -1 şi setează variabila globală `errno` la valoarea `EAGAIN`. În cazul în care funcţia `send()` scrie cel puţin un octet, ea returnează numărul de octeţi scrişi efectiv. În ambele cazuri, este sarcina procesului emiţător să apeleze din nou, la un moment ulterior, funcţia `send()` în vederea scrierii octeţilor rămaşi.

Deoarece funcţia `send()` returnează înainte de transmiterea efectivă a datelor, eventualele erori legate de transmiterea datelor nu pot fi raportate

apelantului prin valoarea returnată de `send()`. Pot să apară două tipuri de erori: căderea reţelei şi închiderea conexiunii de către receptor. Aceste erori vor fi raportate de către sistemul de operare al emiţătorului procesului emiţător prin aceea că apeluri `send()` ulterioare pentru acelaşi socket vor returna -1. În plus, pe sistemele de tip UNIX, apelul `send()` pentru o conexiune al cărui capăt destinaţie este închis duce la primirea de către procesul emiţător a unui semnal SIGPIPE, care are ca efect implicit terminarea imediată a procesului emiţător.

Funcţia `recv()` extrage date sosite pe conexiune şi aflate în buffer-ul sistemului de operare local. Funcţia primeşte ca argumente identificatorul socket-ului corespunzător conexiunii, adresa unei zone de memorie unde să plaseze datele citite şi numărul de octeţi de citit.

Numărul de octeţi de citit reprezintă numărul maxim de octeţi pe care funcţia îi va transfera din buffer-ul sistemului de operare în zona procesului apelant. Dacă numărul de octeţi disponibili în buffer-ele sistemului de operare este mai mic, doar octeţii disponibili în acel moment vor fi transferaţi. Dacă în momentul apelului nu există nici un octet disponibil în buffer-ele sistemului de operare local, funcţia `recv()` aşteaptă sosirea a cel puţin un octet. Funcţia returnează numărul de octeţi transferaţi (citiţi de pe conexiune).

Comportamentul descris mai sus poate fi modificat prin plasarea uneia din următoarele valori în parametrul `flags`:

`MSG_DONTWAIT`:în cazul în care nu este nici un octet disponibil, funcţia `recv()` returnează valoarea -1 şi setează variabila globală `errno` la valoarea `EAGAIN`;

`MSG_WAITALL`:funcţia `recv()` aşteaptă să fie disponibili cel puţin `len` octeţi şi citeşte exact `len` octeţi.

Este important de notat că datele sunt transmise de la sistemul de operare emiţător spre cel receptor în fragmente (pachete), că împărţirea datelor în fragmente este independentă de modul în care au fost furnizate prin apeluri `send()` succesive şi că, în final, fragmentele ce vor fi disponibile succesiv pentru receptor sunt independente de fragmentele furnizate în apelurile `send()`. Ca urmare, este posibil ca emiţătorul să trimită, prin două apeluri `send()` consecutive, şirurile de octeţi `abc` şi `def`, iar receptorul, apelând repetat `recv()` cu `len`=3 şi `flags=0`, să primească `ab`, `cd` şi `ef`. Singurul lucru garantat este că prin concatenarea tuturor fragmentelor trimise de emiţător se obţine acelaşi şir de octeţi ca şi prin concatenarea tuturor fragmentelor primite de receptor.

În cazul închiderii conexiunii de către emiţător, apelurile `recv()` efectuate de procesul receptor vor citi mai întâi datele rămase în buffer-e, iar

după epuizarea acestora vor returna valoarea 0. Prin urmare, funcţia `recv()` returnează valoarea 0 dacă şi numai dacă emiţătorul a închis conexiunea şi toate datele trimise înainte de închiderea conexiunii au fost deja citite. Dealtfel, valoarea 0 returnată de `recv()` sau `read()` este semnalizarea uzuală a terminării datelor de citit şi se utilizează şi pentru a semnaliza sfârşitul unui fişier sau închiderii capătului de scriere într-un *pipe* UNIX.

8.1.3.10. Funcţiile `shutdown()` şi `close()`

```
int shutdown(int sd, int how);
int close(int sd);
```

Funcţia `shutdown()` închide sensul de emisie, sensul de recepţie sau ambele sensuri de comunicaţie ale conexiunii identificate de indetificatorul de socket `sd`, conform valorii parametrului `how`: `SHUT_WR`, `SHUT_RD` sau respectiv `SHUT_RDWR`. Utilitatea principală a funcţiei este închiderea sensului de emisie pentru a semnaliza celuilalt capăt terminarea datelor transmise (apelurile `recv()` din procesul de la celălalt capăt al conexiunii vor returna 0). Funcţia `shutdown()` poate fi apelată doar pe un socket conectat şi nu distruge socket-ul.

Funcţia `close()` distruge socket-ul `sd`. Dacă socket-ul era un socket conectat în acel moment, închide ambele sensuri de comunicaţie. După apelul `close()`, identificatorul de socket este eliberat şi poate fi utilizat ulterior de către sistemul de operare pentru a identifica socket-uri sau alte obiecte create ulterior. Apelul `close()` este necesar pentru a elibera resursele ocupate de socket. Poate fi efectuat oricând asupra oricărui tip de socket.

Terminarea unui proces, indiferent de modul de terminare, are ca efect şi distrugerea tuturor socket-urilor existente în acel moment, printr-un mecanism identic cu câte un apel `close()` pentru fiecare socket.

8.1.3.11. Funcţiile `sendto()` şi `recvfrom()`

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

Funcţia `sendto()` trimite o datagramă de pe un socket *dgram*. Parametrii reprezintă :

- `sd`: socket-ul de pe care se transmite datagrama, adică a cărui adresă va fi utilizată ca adresă sursă a datagramei;

- `to`: pointer spre structura ce conţine adresa de reţea a destinatarului; `tolen` reprezintă lungimea structurii pointate de `to`;
- `buf`: pointer spre o zonă de memorie ce conţine datele utile; `len` reprezintă lungimea datelor utile. Datele utile sunt un şir arbitrar de octeţi.

Funcţia returnează numărul de octeţi ai datagramei trimise (adică valoarea lui `len`) în caz de succes şi -1 în caz de eroare. Funcţia returnează controlul apelantului înainte ca pachetul să fie livrat destinatarului şi ca urmare eventuala pierdere a pachetului nu poate fi raportată apelantului.

Funcţia `recvfrom()` citeşte din bufferele sistemului de operare local următoarea datagramă adresată socket-ului dat ca parametru. Dacă nu există nici o datagramă, funcţia aşteaptă sosirea următoarei datagrame, cu excepţia cazului în care `flags` conţine valoarea `MSG_DONTWAIT`, caz în care `recvfrom()` returnează imediat valoarea -1 şi setează `errno` la valoarea `EAGAIN`.

Datagrama este citită în zona de memorie pointată de parametrul `buf` şi a cărei dimensiune este dată în variabila `len`. Funcţia `recvfrom()` returnează dimensiunea datagramei. Dacă datagrama este mai mare decât valoara parametrului `len`, finalul datagramei este pierdut; funcţia `recvfrom()` nu scrie niciodată dincolo de `len` octeţi în memoria procesului.

Adresa emiţătorului datagramei este plasată de funcţia `recvfrom()` în variabila pointată de `from`. Parametrul `fromlen` trebuie să pointeze la o variabilă de tip întreg a cărui valoare, înainte de apelul `recvfrom()`, trebuie să fie egală cu dimensiunea, în octeţi, a zonei de memorie alocate pentru adresa emiţătorului. Funcţia `recvfrom()` modifică această variabilă, punând în ea dimensiunea utilizată efectiv pentru scrierea adresei emiţătorului.

8.1.4. Exemple

8.1.4.1. Comunicare prin conexiune

Dăm mai jos textul sursă (în C pentru Linux) pentru un client care se conectează la un server TCP/IPv4 specificat prin numele maşinii şi numărul portului TCP (date ca argumente în linia de comandă), îi trimite un şir de caractere fixat (`abcd`), după care citeşte şi afişează tot ce trimite server-ul.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
int main(int argc, char* argv[])
{
```

```
  int port, sd, r;
  struct hostent* hh;
  struct sockaddr_in adr;
  char buf[100];
  if(argc!=3){
    fprintf(stderr, "Utilizare: cli adresa port\n");
    return 1;
  }
  memset(&adr, 0, sizeof(adr));
  adr.sin_family = AF_INET;
  if(1!=sscanf(argv[2], "%d", &port)){
    fprintf(stderr, "numarul de port trebuie sa fie un numar\n");
    return 1;
  }
  adr.sin_port = htons(port);
  hh=gethostbyname(argv[1]);
  if(hh==0 || hh->h_addrtype!=AF_INET || hh->h_length<=0){
    fprintf(stderr, "Nu se poate determina adresa serverului\n");
    return 1;
  }
  memcpy(&adr.sin_addr, hh->h_addr_list[0], 4);
  sd=socket(PF_INET, SOCK_STREAM, 0);
  if(-1==connect(sd, (struct sockaddr*)&adr, sizeof(adr)) )
  {
    perror("connect()");
    return 1;
  }
  send(sd, "abcd", 4, 0);
  shutdown(sd, SHUT_WR);
  while((r=recv(sd, buf, 100, 0))>0){
    write(1,buf,r);
  }
  if(r==-1){
    perror("recv()");
    return 1;
  }
  close(sd);
  return 0;
}
```

Dăm în continuare textul sursă pentru un server care aşteaptă conectarea unui client pe portul specificat în linia de comandă, afişează adresa de la care s-a conectat clientul (adresa IP şi numărul de port), citeşte de pe conexiune

şi afişează pe ecran tot ce transmite clientul (până la închiderea sensului de conexiune de la client la server) şi apoi trimite înapoi textul `xyz`.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
int main(int argc, char* argv[])
{
  int sd, sd_c, port, r;
  struct sockaddr_in my_addr, cli_addr;
  socklen_t cli_addr_size;
  char buf[100];
  if(argc!=2){
    fprintf(stderr, "Utilizare: srv port\n");
    return 1;
  }
  memset(&my_addr, 0, sizeof(my_addr));
  my_addr.sin_family = AF_INET;
  if(1!=sscanf(argv[1], "%d", &port)){
    fprintf(stderr, "numarul de port trebuie sa fie un numar\n");
    return 1;
  }
  my_addr.sin_port=htons(port);
  my_addr.sin_addr.s_addr=htonl(INADDR_ANY);
  sd=socket(PF_INET, SOCK_STREAM, 0);
  if(-1==bind(sd, (struct sockaddr*)&my_addr,
        sizeof(my_addr)) )
  {
        perror("bind()");
        return 1;
  }
  listen(sd, 1);
  cli_addr_size=sizeof(cli_addr);
  sd_c = accept(sd, (struct sockaddr*)&cli_addr,
      &cli_addr_size);
  printf("client conectat de la %s:%d\n",
      inet_ntoa(cli_addr.sin_addr),
      ntohs(cli_addr.sin_port)
  );
  close(sd);
  while((r=recv(sd_c, buf, 100, 0))>0){
```

```
    write(1,buf,r);
  }
  if(r==-1){
    perror("recv()");
    return 1;
  }
  send(sd_c, "xyz", 3, 0);
  close(sd_c);
  return 0;
}
```

8.1.4.2. Comunicare prin datagrame

Mai jos este descris un client UDP/IPv4 care se conectează la un server specificat prin numele maşinii sau adresa IP şi numărul de port. Clientul trimite serverului o datagramă de 4 octeţi conţinând textul `abcd` şi aşteaptă o datagramă ca răspuns, a cărei conţinut îl afişează.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
int main(int argc, char* argv[])
{
  int port, sd, r;
  struct hostent* hh;
  struct sockaddr_in adr;
  socklen_t adr_size;
  char buf[100];
  if(argc!=3){
    fprintf(stderr, "Utilizare: cli adresa port\n");
    return 1;
  }
  memset(&adr, 0, sizeof(adr));
  adr.sin_family = AF_INET;
  if(1!=sscanf(argv[2], "%d", &port)){
    fprintf(stderr, "numarul de port trebuie sa fie un numar\n");
    return 1;
  }
  adr.sin_port = htons(port);
  hh=gethostbyname(argv[1]);
  if(hh==0 || hh->h_addrtype!=AF_INET || hh->h_length<=0){
```

```
    fprintf(stderr, "Nu se poate determina adresa serverului\n");
    return 1;
  }
  memcpy(&adr.sin_addr, hh->h_addr_list[0], 4);
  sd=socket(PF_INET, SOCK_DGRAM, 0);
  if(sd==-1){
    perror("socket()");
    return 1;
  }
  if(-1==sendto(sd, "abcd", 4, 0,
      (struct sockaddr*)&adr, sizeof(adr)) )
  {
    perror("sendto()");
    return 1;
  }
  adr_size=sizeof(adr);
  r=recvfrom(sd, buf, 100, 0,
      (struct sockaddr*)&adr, &adr_size);
  if(r==-1){
    perror("recvfrom()");
    return 1;
  }
  printf("datagrama primita de la de la %s:%d\n",
      inet_ntoa(adr.sin_addr),
      ntohs(adr.sin_port)
  );
  buf[r]=0;
  printf("continut: \"%s\"\n", buf);
  close(sd);
  return 0;
}
```

În continuare descriem un server UDP/IPv4. Acesta aşteaptă o datagramă de la un client, afişează adresa de la care a fost trimisă datagrama precum şi conţinutul datagramei primite. Apoi trimite înapoi, la adresa de la care a sosit datagrama de la client, o datagramă conţinând şirul de 3 octeţi xyz.

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

```
int main(int argc, char* argv[])
{
  int sd, port, r;
  struct sockaddr_in my_addr, cli_addr;
  socklen_t cli_addr_size;
  char buf[101];
  if(argc!=2){
    fprintf(stderr, "Utilizare: srv port\n");
    return 1;
  }
  memset(&my_addr, 0, sizeof(my_addr));
  my_addr.sin_family = AF_INET;
  if(1!=sscanf(argv[1], "%d", &port)){
    fprintf(stderr, "numarul de port trebuie sa fie un numar\n");
    return 1;
  }
  my_addr.sin_port=htons(port);
  my_addr.sin_addr.s_addr=htonl(INADDR_ANY);
  sd=socket(PF_INET, SOCK_DGRAM, 0);
  if(-1==bind(sd, (struct sockaddr*)&my_addr,
        sizeof(my_addr)) )
  {
    perror("bind()");
    return 1;
  }
  cli_addr_size=sizeof(cli_addr);
  r=recvfrom(sd, buf, 100, 0,
      (struct sockaddr*)&cli_addr, &cli_addr_size);
  if(r==-1){
    perror("recvfrom()");
    return 1;
  }
  printf("datagrama primita de la de la %s:%d\n",
      inet_ntoa(cli_addr.sin_addr),
      ntohs(cli_addr.sin_port)
  );
  buf[r]=0;
  printf("continut: \"%s\"\n", buf);
  sendto(sd, "xyz", 3, 0,
      (struct sockaddr*)&cli_addr, cli_addr_size);
  close(sd);
  return 0;
}
```

8.2. Formatarea datelor

Diferite formate de reprezentare a datelor pe conexiune au fost descrise în capitolul 7. În acest paragraf ne vom ocupa de problemele privind transmiterea şi recepţia datelor în astfel de formate.

8.2.1. Formate binare

Formatele binare sunt asemănătoare cu formatele utilizate de programele compilate pentru stocarea datelor în variabilele locale. Până la un punct, este rezonabilă transmiterea unei informaţii prin instrucţiuni de forma

```
Tip msg;
...
send(sd, &msg, sizeof(msg), 0);
```

şi recepţia prin

```
Tip msg;
...
recv(sd, &msg, sizeof(msg), MSG_WAITALL);
```

unde `Tip` este un tip de date oarecare declarat identic în ambele programe (emiţător şi receptor).

Există însă câteva motive pentru care o astfel de abordare nu este, în general, acceptabilă. Vom descrie în continuare problemele legate de fiecare tip de date în parte, precum şi câteva idei privind rezolvarea lor.

8.2.1.1. Tipuri întregi

La transmiterea variabilelor întregi apar două probleme de portabilitate:

- dimensiunea unui întreg nu este, în general, standardizată exact (în C/C++ un `int` poate avea 16, 32 sau 64 de biţi);
- ordinea octeţilor în memorie (*big endian* sau *little endian*) depinde de arhitectura calculatorului.

Dacă scriem un program pentru un anumit tip de calculatoare şi pentru un anumit compilator, pentru care ştim exact dimensiunea unui `int` şi ordinea octeţilor, putem transmite şi recepţiona date prin secvenţe de tipul:

```
int a;
...
send(sd, &a, sizeof(a), 0);
```

pentru emiţător şi

```
int a;
...
recv(sd, &a, sizeof(a), MSG_WAITALL);
```

pentru receptor. Dacă însă emiţătorul este compilat pe o platformă pe care `int` are 16 biţi şi este reprezentat *big endian*, iar receptorul este compilat pe o platformă pe care `int` are 32 de biţi şi este *little endian*, cele două programe nu vor comunica corect.

Pentru a putea scrie programe portabile, biblioteca C standard pe sisteme de tip UNIX conţine, în header-ul `arpa/inet.h`:

- `typedef`-uri pentru tipuri întregi de lungime standardizată (independentă de compilator): `uint16_t` de 16 biţi şi `uint32_t` de 32 de biţi;
- funcţii de conversie între formatul locat (*little endian* sau *big endian*, după caz) şi formatul *big endian*, utilizat cel mai adesea pentru datele transmise în Internet. Aceste funcţii sunt: `htons()` şi `htonl()` (de la *host to network, short*, respectiv *host to network, long*), pentru conversia de la format local la format *big endian* (numit şi *format reţea*), şi `ntohs()` şi `ntohl()` pentru conversia în sens invers. Variantele cu `s` (`htons()` şi `ntohs()`) convertesc întregi de 16 biţi (de tip `uint16_t`, iar cele cu `l` convertesc întregi de 32 de biţi (`uint32_t`).

Implementarea acestor `typedef`-uri şi functii depinde de platformă (de arhitectură şi de compilator). Utilizarea lor permite ca restul sursei programului să nu depindă de platformă.

Transmiterea unui întreg, într-un mod portabil, se face astfel:

```
uint32_t a;
...
a=htonl(a);
send(sd, &a, sizeof(a), 0);
```

```
uint32_t a;
...
recv(sd, &a, sizeof(a), MSG_WAITALL);
a=ntohl(a);
```

Indiferent pe ce platformă sunt compilate, fragmentele de mai sus emit, respectiv recepţionează, un întreg reprezentat pe 32 de biţi în format *big endian*.

8.2.1.2. Şiruri de caractere şi tablouri

Transmiterea sau memorarea unui tablou necesită transmiterea (respectiv memorarea), într-un fel sau altul, a numărului de elemente din tablou. Două metode sunt frecvent utilizate în acest scop: transmiterea în prealabil a numărului de elemente şi transmiterea unui element cu valoare speciale (terminator) după ultimul element.

Pe lângă numărul de elemente efective ale tabloului este necesară cunoaşterea numărului de elemente alocate. La reprezentarea în memorie sau în fişiere pe disc, sunt utilizate frecvent tablouri de dimensiune fixată la compilare. Avantajul dimensiunii fixe este că variabilele situate după tabloul respectiv se pot plasa la adrese fixe şi pot fi accesate direct; dezavantajul este un consum sporit de memorie şi o limită mai mică a numărului de obiecte ce pot fi puse în tablou.

La transmiterea tablourilor prin conexiuni în reţea, de regulă numărul de elemente transmise este egal cu numărul de elemente existente în mod real, plus elementul terminator (dacă este adoptată varianta cu terminator). Nu sunt utilizate tablouri de lungime fixă deoarece datele situate după tablou oricum nu pot fi accesate direct.

În cazul reprezentării cu număr de elemente, receptorul citeşte întâi numărul de elemente, după care alocă spaţiu (sau verifică dacă spaţiul alocat este suficient) şi citeşte elementele. În cazul reprezentării cu terminator, receptorul citeşte pe rând fiecare elemen şi-i verifică valoarea; la întâlnirea terminatorului se opreşte. Înainte de-a citi fiecare element, receptorul trebuie să verifice dacă mai are spaţiu alocat pentru acesta, iar în caz contrar fie să realoce spaţiu pentru tablou şi să copieze în spaţiul nou alocat elementele citite, fie să renunţe şi să semnaleze eroare.

Exemplul 8.2: Se cere transmiterea unui şir de caractere. Reprezentarea şirului pe conexiune este: un întreg pe 16 biţi *big endian* reprezentând lungimea şirului, urmat de şirul propriu-zis (reprezentare diferită deci de reprezentarea uzuală în memorie, unde şirul se termină cu un caracter nul). Descriem în continuare emiţătorul:

```
char* s;
uint16_t l;
...
l=htons(strlen(s));
send(sd, &l, 2, 0);
send(sd, s, strlen(s), 0);
```

şi receptorul:

```
char* s;
uint16_t l;
if(2==recv(sd, &l, 2, MSG_WAITALL) &&
  0!=(s=new char[l=ntohs(l)+1]) &&
  l==recv(sd, s, l, MSG_WAITALL)){
    s[l]=0;
    // sir citit cu succes
} else {
    // tratare eroare
}
```

De remarcat la receptor necesitatea de-a reface terminatorul nul, netransmis prin reţea.

Exemplul 8.3: Se cere transmiterea unui şir de caractere. Reprezentarea pe conexiune va fi ca un şir de caractere urmat de un caracter nul (adică reprezentare identică celei din memorie, dar pe lungime variabilă, egală cu minimul necesar). Emiţătorul este:

```
char* s;
...
send(sd, s, strlen(s)+1, 0);
```

Receptorul:

```
char s[500];
int dim_alloc=500, pos, ret;
while(pos<dim_alloc-1 &&
  1==(ret=recv(sd, s+pos, 1, 0)) &&
  s[pos++]!=0) {}
if(ret==1 && s[pos-1]==0){
    // sir citit cu succes
} else {
    // tratare eroare
}
```

8.2.1.3. Variabile compuse (`struct`-uri)

La prima vedere, variabilele compuse (`struct`-urile) sunt reprezentate la fel şi în memorie şi pe conexiune — se reprezintă câmpurile unul după altul — şi ca urmare transmiterea lor nu ridică probleme deosebite.

Din păcate însă, reprezentarea unei structuri în memorie depinde de platformă (arhitectura calculatorului şi compilator), datorită problemelor

privind alinierea întregilor. Din considerente legate de arhitectura magistralei de date a calculatorului (detalii ce ies din cadrul cursului de faţă), accesarea de către procesor a unei variabile de tip întreg sau real a cărui adresă în memorie nu este multiplu de un anumit număr de octeţi este pentru unele procesoare imposibilă iar pentru celelalte ineficientă. Numărul ce trebuie să dividă adresa se numeşte *aliniere* şi este de obicei minimul dintre dimensiunea variabilei şi lăţimea magistralei. Astfel, dacă magistrala de date este de 4 octeţi, întregii de 2 octaţi trebuie să fie plasaţi la adrese pare, iar întregii de 4 sau 8 octeţi trebuie să fie la adrese multiplu de 4; nu există restricţii cu privire la caractere (întregi pe 1 octet). Compilatorul, împreună cu funcţiile de alocare dinamică a memoriei, asigură alinierea recurgând la următoarele metode:

- adaugă octeţi nefolosiţi între variabile,
- adaugă octeţi nefolosiţi între câmpurile unei structuri,
- adaugă octeţi nefolosiţi la finalul unei structuri ce face parte dintr-un tablou,
- alocă variabilele de tip structură la adrese multiplu de o lăţimea magistralei.

Ca urmare, reprezentarea în memorie a unei strcturi depinde de platformă şi, în consecinţă, un fragment de cod de forma:

```
struct Msg {
  char c;
  uint32_t i;
};
Msg m;
...
m.i=htonl(m.i);
send(sd, &m, sizeof(m), 0);
```

este neportabil. În funcţie de lăţimea magistralei, de faptul că alinierea incorectă duce la imposibilitatea accesării variabilei sau doar la ineficienţă şi, în acest din urmă caz, de opţiunile de compilare, dimensiunea structurii `Msg` de mai sus poate fi 5, 6 sau 8 octeţi, între cele două câmpuri fiind respectiv 0, 1 sau 3 octeţi neutilizaţi.

Rezolvarea problemei de portabilitate se face transmiţând separat fiecare câmp:

```
struct Msg {
  char c;
  uint32_t i;
```

```
};
Msg m;
...
m.i=htonl(m.i);
send(sd, &m.c, 1, 0);
send(sd, &m.i, 4, 0);
```

8.2.1.4. Pointeri

Deoarece un pointer este o adresă în cadrul unui proces, transmiterea unui pointer către un alt proces este complet inutilă.

8.2.2. Formate text

Într-un format text, fiecare câmp este în esenţă un şir de caractere terminat cu spaţiu, tab, *newline* sau un alt caracter specificat prin standard. Metodele descrise pentru transmiterea şi recepţionarea unui şir de caractere se aplică şi la obiectele transmise în formate de tip text.

8.2.3. Probleme de robusteţe şi securitate

Orice apel de funcţie sistem poate eşua din multe motive; ca urmare, la fiecare apel `send()` sau `recv()` programul trebuie să verifice valoarea returnată.

Un receptor robust trebuie să se comporte rezonabil la orice fel de date trimise de partenerul de comunicaţie, inclusiv în cazul încălcării de către acesta a standardului de reprezentare a datelor şi inclusiv în cazul în care emiţătorul închide conexiunea în mijlocul transmiterii unei variabile.

Validitatea datelor trebuie verificată întotdeauna după citire. Dacă receptorul aşteaptă un întreg pozitiv, este necesar să se verifice prin program că numărul primit este într-adevăr pozitiv. Este de asemenea necesar să se stabilească şi să se impună explicit nişte limite maxime. Astfel, să presupunem că programul receptor primeşte un şir de întregi reprezentat prin lungimea, ca număr de elemente, pe 32 de biţi, urmată de elementele propriu-zise. Chiar dacă de principiu numărul de elemente ne aşteptăm să fie între 1 şi câteva sute, trebuie să ne asigurăm că programul se comportă rezonabil dacă numărul de elemente anunţat de emiţător este 0, 2147483647 (adică $2^{31}-1$) sau alte asemenea valori. Comportament rezonabil înseamnă fie să fie capabil să proceseze corect datele, fie să declare eroare şi să încheie curat operaţiile începute.

Orice program care nu este robust este un risc de securitate.

8.2.4. Probleme privind costul apelurilor sistem

Apelul funcţiilor `send()` şi `recv()` este scump, în termeni de timp de procesor, deoarece, fiind funcţii sistem, necesită o comutare de drepturi în procesor, salvarea şi restaurarea contextului apelului şi o serie de verificări din partea nucleului sistemului de operare; în total, echivalentul câtorva sute de instrucţiuni. Acest cost este independent de numărul de octeţi transferaţi.

Este, prin urmare, eficient ca fiecare apel `send()` sau `recv()` să transfere cât de mulţi octeţi se poate. Un program care trimite date este bine să pregătească datele într-o zonă tampon locală şi să trimită totul printr-un singur apel `send()`. Un program care primeşte date este bine să ceară (prin `recv()`) câte un bloc mai mare de date şi apoi să analizeze datele primite. Acest mod de lucru se realizează cel mai bine prin intermediul unor funcţii de bibliotecă adecvate.

Descriem în continuare funcţiile din biblioteca standard C utilizabile în acest scop. Biblioteca poate fi utilizată atât pentru emisie şi recepţie printr-o conexiune *socket stream* cât şi pentru citire sau scriere într-un fişier sau pentru comunicare prin *pipe* sau *fifo*.

Elementul principal al bibliotecii este structura `FILE`, ce conţine:

- un identificator de fişier deschis, conexiune *socket stream*, *pipe* sau *fifo*;
- o zonă de memorie tampon, împreună cu variabilele necesare gestionării ei.

Funcţiile de citire ale bibliotecii sunt `fread()`, `fscanf()`, `fgets()` şi `fgetc()`. Fiecare dintre aceste funcţii extrage datele din zona tampon a structurii `FILE` dată ca parametru. Dacă în zona tampon nu sunt suficienţi octeţi pentru a satisface cererea, aceste funcţii apelează funcţia sistem `read()` asupra identificatorului de fişier din structura `FILE` pentru a obţine octeţii următori. Fiecare astfel de apel `read()` încearcă să citească câţiva kiloocteţi. Pentru fiecare din funcţiile de mai sus, dacă datele de returnat aplicaţiei se găsesc deja în zona tampon, costul execuţiei este de câteva instrucţiuni pentru fiecare octet transferat. Ca urmare, la citirea a câte un caracter o dată, utilizarea funcţiilor de mai sus poate reduce timpul de execuţie de câteva zeci de ori.

Exemplul 8.4: Fie următoarele fragmente de cod:

```
int sd;
char s[512];
int i,r;
...
while( ((r=recv(sd, s+i, 1, 0))==1 && s[i++]!=0 ) {}
```

şi

```
FILE* f;
char s[512];
int i,r;
...
while( (r=fgetc(f))!=EOF && (s[i++]=r)!=0) {}
```

Ambele fragmente de cod citesc de pe un *socket stream* un şir de octeţi terminat cu un caracter nul. Primul fragment de cod apelează `recv()` o dată pentru fiecare caracter al şirului. Al doilea fragment apelează `fgetc()` o dată pentru fiecare caracter al şirului. La o mică parte dintre aceste apeluri, functia `fgetc()` va apela în spate funcţia sistem `read()` pentru a citi efectiv datele de pe conexiune; la restul apelurilor, `fgetc()` returnează apelantului câte un caracter aflat deja în zona tampon. Ca rezultat global, al doilea fragment de cod se va executa de câteva zeci de ori mai repede decât primul.

Testarea închiderii conexiunii şi terminării datelor se poate face apelând funcţia `foef()`. De notat că această funcţie poate să returneze `false` chiar dacă s-a ajuns la finalul datelor; rezultatul `true` este garantat doar după o tentativă nereuşită de-a citi dincolo de finalul datelor transmise.

Pentru scriere, funcţiile de bibliotecă corespunzătoare sunt `fwrite()`, `fprintf()`, `fputs()` şi `fputc()`. Aceste funcţii scriu datele în zona tampon din structura `FILE`. Transmiterea efectivă pe conexiune (sau scrierea în fişier) se face automat la umplerea zonei tampon. Dacă este necesar să ne asigurăm că datele au fost transmise efectiv (sau scrise în fişier), funcţia `fflush()` efectuează acest lucru. Funcţia `fclose()` de asemenea trimite sau scrie ultimele date rămase în zona tampon.

Asocierea unei zone tampon unei conexiuni deja deschise se face apelând funcţia `fdopen()`. Funcţia `fdopen()` primeşte doi parametri. Primul parametru este identificatorul de socket căruia trebuie să-i asocieze zona tampon (identificatorul returnat de funcţia `socket()` sau `accept()`). Al doilea parametru specifică funcţiei `fdopen()` dacă trebuie să asocieze zona tampon pentru citire sau pentru scriere; este de tip şir de caractere şi poate avea valoarea `"r"` pentru citire sau `"w"` pentru scriere. Pentru un *socket stream*, cele două sensuri functionând complet independent, aceluiaşi socket i se pot asocia două zone tampon (două structuri `FILE`), câte una pentru fiecare sens.

Funcţia `fclose()` scrie informaţiile rămase în zona tampon (dacă zona tampon a fost creată pentru sensul de scriere), eliberează memoria alocată zonei tampon şi închide conexiunea.

8.3. Probleme de concurenţă în comunicaţie

O particularitate a majorităţii programelor ce comunică în reţea este aceea că trebuie să răspundă prompt la mesaje provenind din surse diferite şi într-o ordine necunoscută dinainte.

Să luăm de exemplu un server *ssh* (§ 11.2.1). Serverul poate avea mai mulţi clienţi conectaţi simultan. La fiecare moment, este imposibil de prezis care dintre clienţi va trimite primul o comandă.

Dacă serverul execută un apel `recv()` blocant de pe socket-ul unui client, serverul va fi pus în aşteptare până ce acel client va trimite date. Este posibil ca utilizatorul ce comandă acel client să stea 10 minute să se gândească. Dacă în acest timp un alt client trimite o comandă, serverul nu o va putea „vedea“ cât timp este blocat în aşteptarea datelor de la primul client. Ca urmare, datele de la al doilea client vor aştepta cel puţin 10 minute pentru a fi procesate.

Există mai multe soluţii la problema de mai sus:

- Serverul citeşte de la clienţi, pe rând, prin apeluri `recv()` neblocante (cu flagul `MSG_DONTWAIT`):

  ```
  for(i=0 ; true ; i=(i+1)%nr_clienti){
    r=recv(sd[i], buf, dim, MSG_DONTWAIT);
    if(r>=0 || errno!=EAGAIN){
      /* prelucreaza mesajul primit
      sau eroarea aparuta */
    }
  }
  ```

  În acest fel, serverul nu este pus în aşteptare dacă un client nu i-a trimis nimic. Dezavantajul soluţiei este acela că, dacă o perioadă de timp nici un client nu trimite nimic, atunci bucla se execută în mod repetat, consumând inutil timp de procesor.

- Pentru evitarea inconvenientului soluţiei anterioare, sistemele de operare de tip UNIX oferă o funcţie sistem, numită `select()`, care primeşte o listă de identificatori de *socket* şi, opţional, o durată, şi pune procesul în aşteptare până când fie există date disponibile pe vreunul din *socket*-ii daţi, fie expiră durata de timp specificată.

- O abordare complet diferită este aceea de-a crea mai multe procese — sau, în sistemele de operare moderne, mai multe fire de execuţie (thread-uri) în cardul procesului server — fiecare proces sau fir de execuţie urmărind un singur client. În acest caz, procesul sau firul de execuţie poate executa `recv()` blocant asupra socket-ului corespunzător clientului său. În lipsa

activităţii clienţilor, fiecare proces sau fir de execuţie al serverului este blocat în apelul `recv()` asupra socket-ului corespunzător. În momentul în care un client trimite date, nucleul sistemului de operare trezeşte procesul sau firul ce executa `recv()` pe socket-ul corespunzător; procesul sau firul execută prelucrările necesare după care probabil execută un nou `recv()` blocant.

Cazul unui server cu mai mulţi clienţi nu este singura situaţie în care este nevoie de a urmări simultan evenimente pe mai multe canale. Alte situaţii sunt:

- un client care trebuie să urmărească simultan acţiunile utilizatorului şi mesajele sosite de la server;
- un server care poate trimite date cu debit mai mare decât capacitatea reţelei sau capacitatea de prelucrare a clientului; în acest caz serverul are de urmărit simultan, pe de o parte noi cereri ale clienţilor, iar pe de altă parte posibilitatea de-a trimite noi date spre clienţi în urma faptului că vechile date au fost prelucrate de aceştia.
- un server care trebuie să preia mesaje de la clienţii conectaţi şi, în acelaşi timp, să poată accepta clienţi noi.

Un aspect important ce trebuie urmărit în proiectarea unui server concurent este servirea echitabilă a clienţilor, independent de comportamentul acestora. Dacă un client trimite cereri mai repede decât este capabil serverul să-l servească, serverul trebuie să execute o parte din cereri, apoi să servească cereri ale celorlalţi clienţi, apoi să revină la primul şi să mai proceseze o parte din cereri şi aşa mai departe. Nu este permis ca un client care inundă serverul cu cereri să acapareze întreaga putere de calcul a serverului şi ceilalţi clienţi să aştepte la infinit.