PROTOCOALE DE COMUNICAȚIE : LABORATOR 7

Protocolul de transport TCP

Responsabil: Radu-Ioan CIOBANU

Cuprins

Obiective

În urma parcurgerii acestui laborator, studentul va fi capabil:

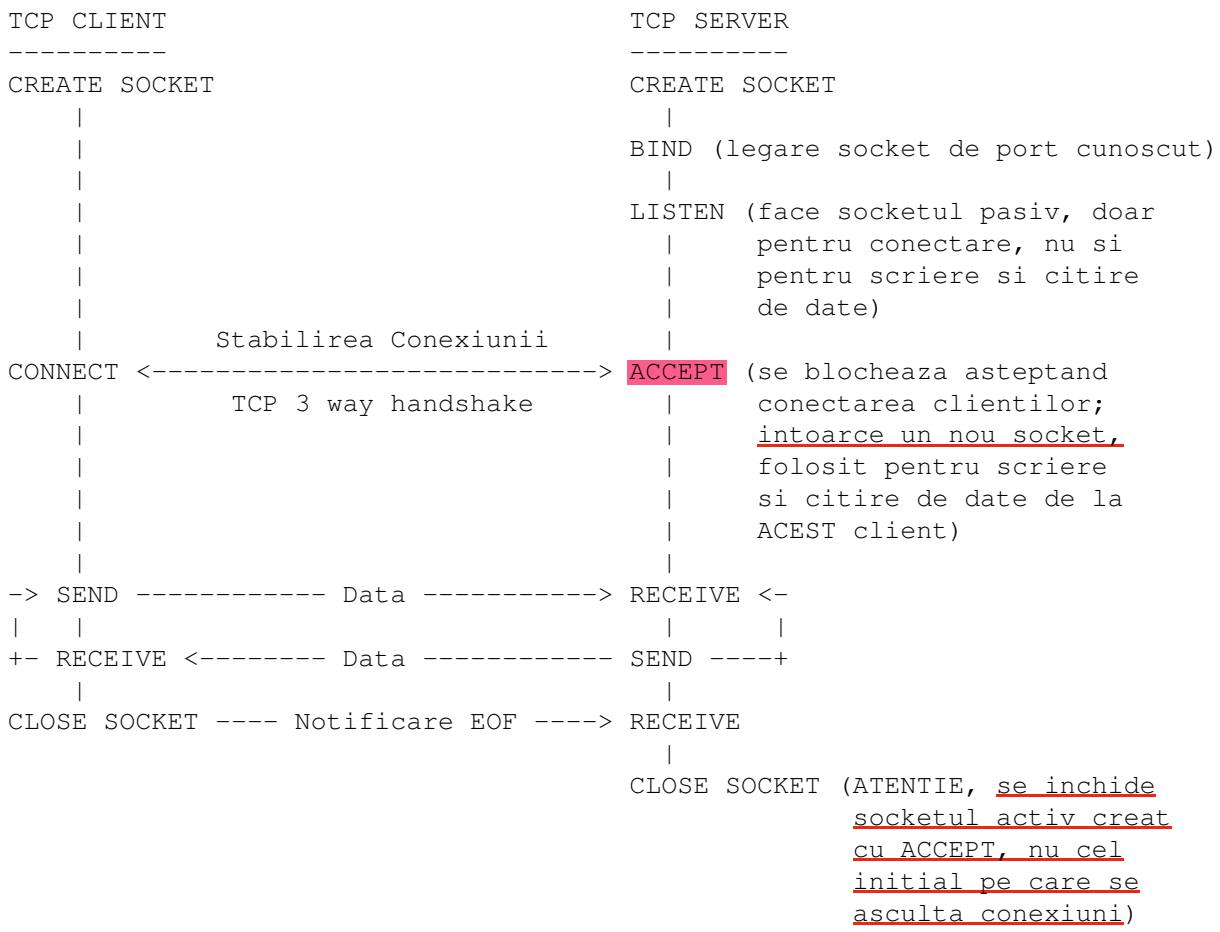
- să explice în ce constă protocolul de transport TCP
- să scrie un program care foloseşte socketi TCP.

Noțiuni teoretice

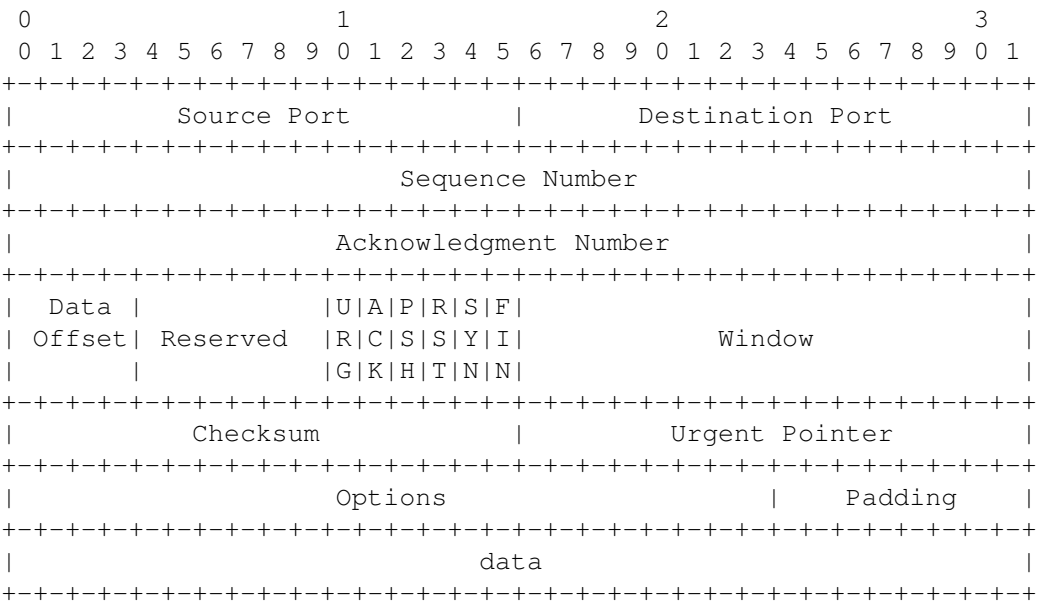
TCP (Transport Control Protocol) este un protocol ce furnizează un flux sigur de date între două calculatoare. Acest protocol asigură stabilirea unei conexiuni permanente între cele două calculatoare pe parcursul comunicaţiei, și este descris în RFC 793. Protocolul TCP are următoarele proprietăţi:

- stabilirea unei conexiuni permanente între client şi server; serverul va aştepta apeluri de conexiune din partea clienților
- garantarea ordinii primirii mesajelor şi prevenirea pierderii pachetelor
- controlul congestiei (fereastră glisantă)
- overhead mare în comparaţie cu UDP (are un header de 20 Bytes, față de UDP, care are doar 8 Bytes).

Schema tipică a interacțiunilor client/server în TCP



Header TCP



Explicaţii header:

- **portul sursă** este ales random de către maşina sursă a pachetului, dintre porturile libere existente pe acea maşină
- **portul destinaţie** este portul pe care maşina destinaţie poate recepţiona pachete
- **checksum** este valoarea sumei de control pentru un pachet TCP.

API

socket()

Pentru obţinerea descriptorului de fişier, se foloseşte, ca și în cazul UDP, funcția:

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

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

Câmpurile *domain* și *protocol* au aceleași valori ca și în cazul UDP (*AF_INET* / *PF_INET*, respectiv 0). În cazul TCP, pentru câmpul *type* se utilizează valoarea *SOCK_STREAM*. Diferența notabilă dintre TCP și UDP este în modul de utilizare de către server a descriptorului întors de *socket()*. În cazul UDP, acest socket este implicit activ, adică se fac trimiteri și recepționări de date pe el. În cazul TCP, acest socket va fi folosit numai în mod pasiv, pentru stabilirea conexiunii. Nu se vor trimite sau recepționa date ale aplicației prin intermediul său. Va exista un alt socket (diferit pentru fiecare client) care va fi folosit pentru transmisie.

bind()

Odata ce am obţinut un socket, trebuie să îi asociem un port pe maşina locală (acest lucru este uzual în cazul în care se doreşte aşteptarea de către server de conexiuni pe un anumit port). Utilizarea este exact ca în cazul UDP.

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

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

connect()

După ce a creat socketul, clientul trebuie să se conecteze la server.

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

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

Argumentul *sockfd* este un descriptor de fişier obţinut în urma apelului *socket()*, *addr* conţine portul şi adresa IP ale serverului, iar *addrlen* poate fi setat la *sizeof(struct sockaddr)*. Ca şi în cazul celorlate funcţii, rezultatul este -1 în caz de eroare, iar în caz de succes 0.

listen()

Comunicaţia prin conexiune stabilă este asimetrică. Mai precis, unul din cele două procese implicate joacă rol de server, iar celălalt joacă rol de client. Cu alte cuvinte, serverul trebuie să îi asocieze socketului propriu o adresă pe care oricare client trebuie să o cunoască, şi apoi să "asculte" pe acel socket cererile ce provin de la clienţi. Mai mult decât atât, în timp ce serverul este ocupat cu tratarea unei cereri, există posibilitatea de a întârzia cererile ce provin de la alţi clienţi, prin plasarea lor într-o coadă de aşteptare. Lungimea acestei cozi trebuie specificată prin apelul *listen()*.

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

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

Argumentul *sockfd* este un descriptor de fişier obţinut în urma apelului *socket()*, iar *backlog* reprezintă numărul de conexiuni acceptate în coada de aşteptare. Conexiunile care se fac de către clienți vor aştepta în aceasta coadă până când se face *accept()*, şi nu pot fi mai mult de *backlog* conexiuni în aşteptare. Apelul *listen()* întoarce 0 în caz de succes şi -1 în caz de eroare.

accept()

Ce se întâmplă în momentul în care un client încearcă să apeleze *connect()* pe o maşină şi un port pe care s-a facut în prealabil *listen()*? Conexiunea va fi pusă în coada de aşteptare până în momentul în care se face un apel de *accept()* de către server. Acest apel întoarce un nou socket care va fi folosit pentru această conexiune.

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

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

Argumentul *sockfd* reprezintă socketul pe care s-a făcut *listen()* (deci cel întors de apelul *socket()*). Funcția *accept()* întoarce un nou socket, care va fi folosit pentru operații *send() / recv()*. *addr* reprezintă un pointer spre o structură de tip *struct sockaddr* în care se va afla informaţia despre conexiunea făcuta (ce maşină de pe ce port a iniţiat conexiunea). Noul socket obţinut prin apelul *accept()* va fi folosit în continuare pentru operaţiile de transmisie și recepție de date.

send() / recv()

Aceste două funcţii se folosesc pentru a transmite date prin socketi de tip stream sau socketi datagramă conectaţi. Sintaxa pentru trimitere şi primire este asemănătoare.

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

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

Argumentul *sockfd* este socketul căruia se dorește să se trimită date (fie este returnat de apelul *socket()*, fie de apelul *accept()*). Argumentul *buf* este un pointer către adresa de memorie unde se găsesc datele ce

se doresc a fi trimise, iar argumentul *len* reprezintă numărul de octeți din memorie începand de la adresa respectivă ce se vor trimite. Functia *send()* întoarce numărul de octeți efectiv trimiși (acesta poate fi mai mic decât numărul care s-a precizat că se dorește a fi trimis, adică *len*). În caz de eroare, funcția returnează -1, setându-se corespunzător variabila globală *errno*.

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

ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

La funcția *recv()*, argumentul *sockfd* reprezintă socketul de unde se citesc datele, *buf* reprezintă un pointer către o adresă din memorie unde se vor scrie octeții citiți, iar *len* reprezintă numărul maxim de octeți ce se vor citi. Functia *recv()* întoarce numărul de octeți efectiv citiți în *buf* sau -1 în caz de eroare.

Observații:

- *recv()* poate întoarce și 0, acest lucru însemnând că entitatea cu care se comunică a închis conexiunea
- pentru scrierea/citirea în/din socketi TCP, se pot folosi cu succes și functiile *write()* și *read()* (foarte asemănătoare cu *send()* și *recv()*, mai puțin câmpul *flags*, pe care oricum îl setăm pe 0).

close()

Pentru a închide un socket TCP, se folosește funcția de închidere a unui descriptor de fișier din Unix:

```
#include <unistd.h>

int close(int fd);
```

Atenție, în cazul TCP avem de-a face cu mai mulți socketi: unul întors de *socket()* (folosit pentru stabilire de conexiuni cu clienții) și unul sau mai multi socketi întorși de *accept()* (câte unul pentru fiecare client conectat), folosiți pentru comunicația cu respectivul client. În cazul în care clientul s-a deconectat (*recv()* întoarce 0), vrem să închidem socketul corespunzător creat cu *accept()*, nu pe cel creat cu *socket()*.

Exerciții

1. Scrieți o aplicație client-server TCP în care serverul se va comporta ca ecoul clientului (echo server). Într-o buclă, clientul citește un string de la tastatură, îl trimite serverului, așteaptă răspuns de la server și îl afișează. Serverul trimite înapoi clientului același lucru pe care îl primește de la el. Atât serverul cât și clientul primesc ca argumente adresa și portul serverului.
2. Completați codul serverului de mai sus astfel încât să funcționeze cu 2 clienți (ambele apeluri de *accept()* trebuie făcute înainte de primul *send()* / *recv()*). Serverul va intermedia un fel de chat între cei doi clienți: va primi ceva de la un client și va trimite celuilalt, și reciproc. Trebuie avută atenție la ordinea operațiilor (scriere/citire de pe socket) atunci când rulați clienții (în laboratorul viitor, vom folosi în server un mecanism de multiplexare care va elimina acest inconvenient; clienții nu vor mai trebui să scrie/citească de pe socket într-o anumită ordine).