6

NIVELUL TRANSPORT

Nivelul transport nu este doar un alt nivel, el este miezul întregii ierarhii de protocoale. Sarcina sa este de a transporta date de la mașina sursă la mașina destinație într-o manieră sigură și eficace din punctul de vedere al costurilor, independent de rețeaua sau rețelele fizice utilizate. Fără nivelul transport și-ar pierde sensul întregul concept de ierarhie de protocoale. În acest capitol vom studia în detaliu nivelul transport, incluzând serviciile, arhitectura, protocoalele și performanțele sale.

6.1 SERVICIILE OFERITE DE NIVELUL TRANSPORT

În secțiunile următoare vom face o prezentare a serviciilor oferite de nivelul transport. Vom studia serviciile oferite nivelului aplicație. Pentru a face problema serviciului de transport mai concretă, vom examina două seturi de primitive ale nivelului transport. La început ne vom ocupa de unul simplu (dar ipotetic), pentru a arăta ideile de bază. Apoi va fi studiată interfața folosită în mod obișnuit în Internet.

6.1.1 Servicii furnizate nivelurilor superioare

Scopul principal al nivelului transport este de a oferi servicii eficiente, sigure și ieftine utilizatorilor, în mod normal procese aparținând nivelului aplicație. Pentru a atinge acest scop, nivelul transport utilizează serviciile oferite de nivelul rețea. Hardware-ul și/sau software-ul care se ocupă de toa-

te acestea în cadrul nivelului transport poartă numele de **entitate de transport**. Entitatea de transport poate aparține nucleului sistemului de operare, unui proces distinct, unei biblioteci legate de aplicațiile de rețea sau poate fi găsită în cadrul plăcii de rețea. Relația (logică) între nivelurile rețea, transport și aplicație este prezentată în fig. 6-1.

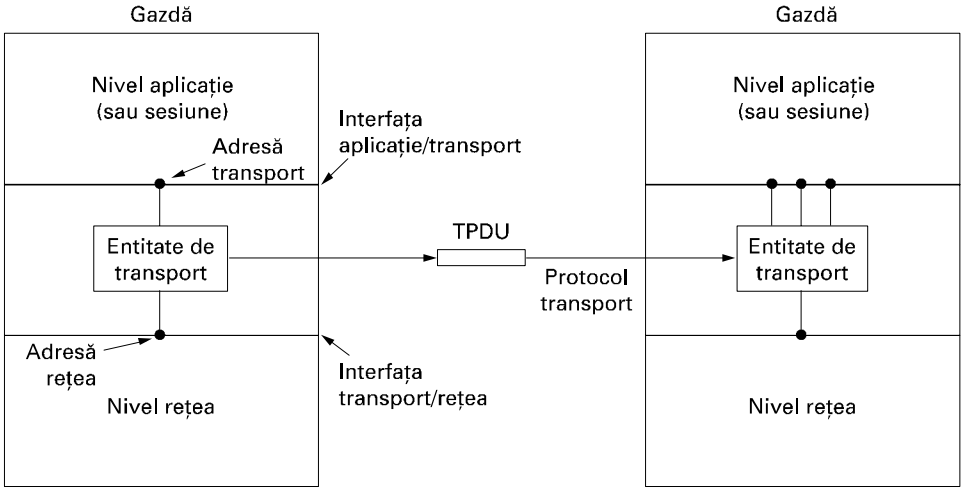


Fig. 6-1. Nivelurile rețea, transport și aplicație.

Cele două tipuri de servicii: orientate pe conexiune sau datagramă, existente în cadrul nivelului rețea, se regăsesc și la acest nivel. Serviciul orientat pe conexiune de la nivelul transport are multe asemănări cu cel de la nivel rețea. În ambele cazuri, conexiunile au trei faze: stabilirea conexiunii, transferul de date și eliberarea conexiunii. Adresarea și controlul fluxului sunt și ele similare pentru ambele niveluri. Mai mult, chiar și serviciul fără conexiune al nivelului transport este foarte asemănător cu cel al nivelului rețea.

O întrebare evidentă este atunci: dacă serviciile la nivel transport sunt atât de asemănătoare cu cele de la nivel rețea, de ce este nevoie de două niveluri distincte? De ce nu este suficient un singur nivel? Răspunsul este unul subtil, dar extrem de important, și ne cere să ne întoarcem la fig. 1-9. Codul pentru nivelul transport este executat în întregime pe mașinile utilizatorilor, dar nivelul rețea este executat în cea mai mare parte de mediul de transport (cel puțin pentru rețelele larg răspândite geografic - WAN). Ce s-ar întâmpla dacă nivelul rețea ar oferi servicii neadecvate? Dar dacă acesta ar pierde frecvent pachete? Ce se întâmplă dacă din când în când ruterul cade?

Ei bine, în toate aceste cazuri apar probleme. Deoarece utilizatorii nu pot controla nivelul rețea, ei nu pot rezolva problema unor servicii de proastă calitate folosind rutere mai bune sau adăugând o tratare a erorilor mai sofisticată la nivelul legătură de date. Singura posibilitate este de a pune deasupra nivelului rețea un alt nivel care să amelioreze calitatea serviciilor. Dacă pe o subrețea orientată pe conexiune, o entitate de transport este informată la jumătatea transmisiei că a fost închisă abrupt conexiunea sa la nivel rețea, fără nici o indicație despre ceea ce s-a întâmplat cu datele aflate în acel moment în tranzit, ea poate iniția o altă conexiune la nivel rețea cu entitatea de transport aflată la distanță. Folosind această nouă conexiune, ea își poate întreba corespondenta care date au ajuns la destinație și care nu, și poate continua comunicarea din locul de unde a fost întreruptă.

În esență, existența nivelului transport face posibil ca serviciile de transport să fie mai sigure decât cele echivalente de la nivelul rețea. Pachetele pierdute sau incorecte pot fi detectate și corectate de către nivelul transport. Mai mult, primitivele serviciului de transport pot fi implementate ca apeluri către procedurile de bibliotecă, astfel încât să fie independente de primitivele de la nivelul rețea. Apelurile nivelului rețea pot să varieze considerabil de la o rețea la alta (de exemplu, serviciile fără conexiune într-o rețea locală pot fi foarte diferite de serviciile orientate pe conexiune dintr-o rețea larg răspândită geografic). Ascunzând serviciul rețea în spatele unui set de primitive ale serviciului transport, schimbarea serviciului rețea necesită numai înlocuirea unui set de proceduri de bibliotecă cu un altul care face același lucru, cu un serviciu inferior diferit.

Mulțumită nivelului transport, programatorii de aplicații pot scrie cod conform unui set standard de primitive, pentru a rula pe o mare varietate de rețele, fără să își pună problema interfețelor de subrețea diferite sau transmisiilor nesigure. Dacă toate rețelele reale ar fi perfecte și toate ar avea același set de primitive și ar fi garantate să nu se schimbe niciodată, atunci probabil că nivelul transport nu ar mai fi fost necesar. Totuși, în lumea reală el îndeplinește importanta funcție de a izola nivelurile superioare de tehnologia, arhitectura și imperfecțiunile subrețelei.

Din această cauză, în general se poate face o distincție între nivelurile de la 1 la 4, pe de o parte, și cel (cele) de deasupra, pe de altă parte. Primele pot fi văzute ca **furnizoare de servicii de transport,** iar ultimele ca **utilizatoare de servicii de transport**. Această distincție între utilizatori și furnizori are un impact considerabil în ceea ce privește proiectarea arhitecturii de niveluri și conferă nivelului transport o poziție cheie, acesta fiind limita între furnizorul și utilizatorul serviciilor sigure de transmisie de date.

6.1.2 Primitivele serviciilor de transport

Pentru a permite utilizatorului să acceseze serviciile de transport, nivelul transport trebuie să ofere unele operații programelor aplicație, adică o interfață a serviciului transport. Fiecare serviciu de transport are interfața sa. În acest capitol, vom examina mai întâi un serviciu de transport simplu (ipotetic) și interfața sa pentru a vedea aspectele esențiale. În secțiunea următoare vom analiza un exemplu real.

Serviciul transport este similar cu cel rețea, dar există și câteva diferențe importante. Principala diferență este că serviciul rețea a fost conceput pentru a modela serviciile oferite de rețelele reale. Acestea pot pierde pachete, deci serviciile la nivel rețea sunt în general nesigure.

În schimb, serviciile de transport (orientate pe conexiune) sunt sigure. Desigur, în rețelele reale apar erori, dar este tocmai acesta este scopul nivelului transport: să furnizeze un serviciu sigur deasupra unui nivel rețea nesigur.

Ca exemplu, să considerăm două procese conectate prin ‘pipe’-uri (tuburi) în UNIX. Acestea presupun o conexiune perfectă între ele. Ele nu vor să aibă de-a face cu confirmări, pachete pierdute, congestii sau altele asemănătoare. Ele au nevoie de o conexiune sigură în proporție de 100%. Procesul A pune datele la un capăt al tubului, iar procesul B le ia de la celălalt capăt. Aceasta este exact ceea ce face un serviciu transport orientat pe conexiune: ascunde imperfecțiunile rețelei, astfel încât procesele utilizator pot să presupună existența unui flux de date fără erori.

În același timp nivelul transport furnizează și un serviciu nesigur. Totuși, sunt puține de spus în legătură cu acesta, așa că în acest capitol ne vom concentra atenția asupra serviciului orientat pe conexiune. Cu toate acestea, există unele aplicații, cum sunt programele client-server și fluxurile multimedia, care beneficiază de transport fără conexiune, deci vom vorbi puțin despre ele mai târziu.

O a doua diferență între serviciul rețea și cel de transport se referă la destinațiile lor. Serviciul rețea este folosit doar de entitățile de transport. Puțini utilizatori scriu ei înșiși entitățile de transport și, astfel, puțini utilizatori sau programe ajung să vadă vreodată serviciile rețea așa cum sunt ele. În schimb, multe programe (și programatori) folosesc primitivele de transport. De aceea, serviciul transport trebuie să fie ușor de utilizat.

Ca să ne facem o idee despre cum poate arăta un serviciu de transport, să considerăm cele cinci primitive prezentate în fig. 6-2. Această interfață este într-adevăr simplă, dar prezintă trăsăturile de bază ale oricărei interfețe orientate pe conexiune a nivelului transport. Ea permite programelor de aplicație să stabilească, să utilizeze și să elibereze conexiuni, ceea ce este suficient pentru multe aplicații.

Primitiva	Unitatea de date trimisă	Explicații
LISTEN	(nimic)	Se blochează până când un proces încearcă să se conecteze
CONNECT	CONNECTION REQ.	Încearcă să stabilească conexiunea
SEND	DATE	Transmite informație
RECEIVE	(nimic)	Se blochează până când primește date trimise
DISCONNECT	DISCONNECTION REQ.	Trimisă de partea care vrea să se deconecteze

Fig. 6-2. Primitivele unui serviciu de transport simplu.

Pentru a vedea cum pot fi utilizate aceste primitive, să considerăm o aplicație cu un server și un număr oarecare de clienți la distanță. La început, serverul apelează primitiva LISTEN, în general prin apelul unei funcții de bibliotecă care face un apel sistem pentru a bloca serverul până la apariția unei cereri client. Atunci când un client vrea să comunice cu serverul, el va executa un apel CONNECT. Entitatea de transport tratează acest apel blocând apelantul și trimițând un pachet la server. Acest pachet încapsulează un mesaj către entitatea de transport de pe server.

Este momentul să facem câteva precizări în legătură cu terminologia. În lipsa unui termen mai bun vom folosi acronimul **TPDU (Transport Protocol Data Unit -** unitate de date a protocolului de transport) pentru toate mesajele schimbate între două entități de transport corespondente. Astfel, TPDU-urile (schimbate la nivelul transport) sunt conținute în pachete (utilizate de nivelul rețea). La rândul lor, pachetele sunt conținute în cadre (utilizate la nivelul legătură de date). Atunci când este primit un cadru, nivelul legătură de date prelucrează antetul cadrului și dă conținutul util nivelului rețea. Entitatea rețea prelucrează antetul pachetului și pasează conținutul util entității de transport. Această ierarhie este ilustrată în fig. 6-3.

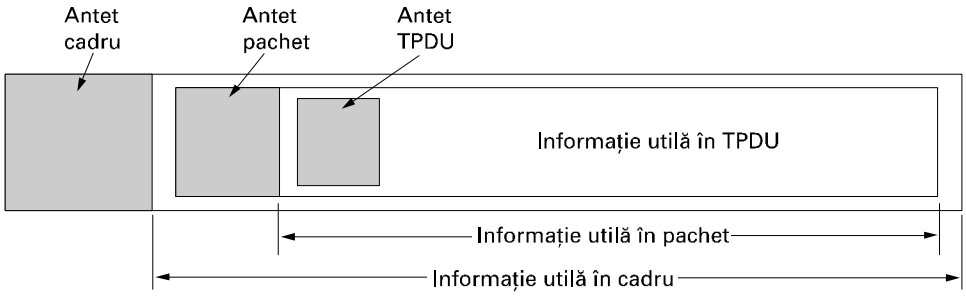


Fig. 6-3. Ierarhia de cadre, pachete și TPDU-uri.

Revenind la exemplul nostru, apelul CONNECT al clientului generează un TPDU de tip CONNECTION REQUEST care îi este trimis serverului. Atunci când acesta ajunge, entitatea de transport verifică dacă serverul este blocat într-un apel LISTEN (deci dacă așteaptă o cerere de conexiune). În acest caz, deblochează serverul și trimite înapoi clientului un TPDU CONNECTION ACCEPTED. Atunci când acest TPDU ajunge la destinație, clientul este deblocat și conexiunea este stabilită.

Acum pot fi schimbate date folosindu-se primitivele SEND și RECEIVE. Cea mai simplă posibilitate este ca una din părți să facă un apel RECEIVE (blocant) așteptând ca cealaltă parte să execute un SEND. Atunci când sosește un TPDU, receptorul este deblocat. El poate prelucra TPDU-ul și trimite o replică. Atâta vreme cât amândouă părțile știu cine este la rând să trimită mesaje și cine este la rând să recepționeze, totul merge bine.

Trebuie să observăm că la nivelul transport, chiar și un schimb de date simplu, unidirecțional, este mult mai complicat decât la nivelul rețea. Fiecare pachet de date trimis va fi (în cele din urmă) confirmat. Pachetele care conțin TPDU-uri de control sunt de asemenea confirmate, implicit sau explicit. Aceste confirmări sunt gestionate de entitățile de transport folosind protocoalele de la nivelul rețea și nu sunt vizibile utilizatorilor nivelului transport. Similar, entitățile de transport trebuie să se ocupe de ceasuri și de retransmisii. Nimic din tot acest mecanism nu este vizibil pentru utilizatorii nivelului transport, pentru care o conexiune este un tub fără pierderi: un utilizator îndeasă biți la un capăt și aceștia apar, ca prin minune, la capătul celalalt. Această capacitate de a ascunde complexitatea este motivul care face ierarhia de protocoale să fie un instrument atât de puternic.

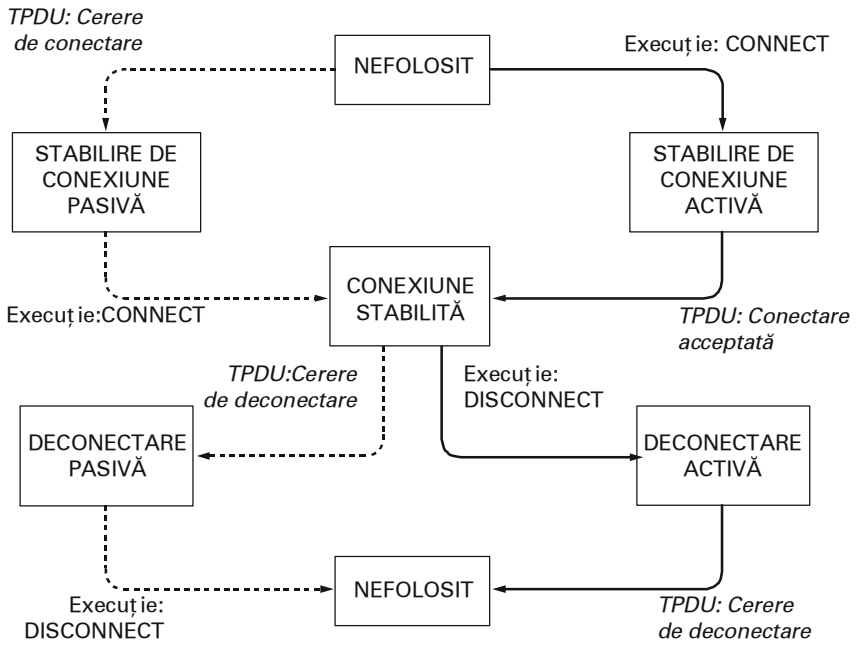


Fig. 6-4. Diagrama de stări pentru o schemă simplă de control al conexiunii.
Tranzițiile etichetate cu italice sunt cauzate de sosirea unor pachete.
Liniile continue indică secvența de stări a clientului.
Liniile punctate indică secvența de stări a serverului.

Atunci când o conexiune nu mai este necesară, ea trebuie eliberată pentru a putea elibera și spațiul alocat în tabelele corespunzătoare din cele două entități de transport. Deconectările se pot face în două variante: asimetrică sau simetrică. În varianta asimetrică, oricare dintre utilizatori poate apela o primitivă DISCONNECT, ceea ce va avea ca rezultat trimiterea unui TPDU DISCONNECT REQUEST entității de transport aflate la distanță. La sosirea acestuia conexiunea este eliberată.

În varianta simetrică fiecare direcție este închisă separat, independent de cealaltă. Atunci când una din părți face un apel DISCONNECT, însemnând că nu mai sunt date de trimis, ea va putea încă recepționa datele transmise de entitatea de transfer aflată la distanță. În acest model conexiunea este eliberată dacă ambele părți au apelat DISCONNECT.

O diagramă de stări pentru stabilirea și eliberarea conexiunilor folosind aceste primitive simple este prezentată în fig. 6-4. Fiecare tranziție este declanșată de un eveniment: fie este executată o primitivă de către utilizatorul local al nivelului transport, fie este primit un pachet. Pentru simplitate vom presupune că fiecare TPDU este confirmat separat. Vom presupune de asemenea că este folosit un model de deconectare simetric, clientul inițiind acțiunea. Trebuie reținut că acesta este un model foarte simplu, în secțiunile următoare vom analiza modele reale.

6.1.3 Socluri Berkeley

Vom trece în revistă acum un alt set de primitive de transport: primitivele pentru socluri TCP folosite în sistemul de operare Berkeley-UNIX. Primitivele sunt enumerate în fig. 6-5. În general putem spune că acestea sunt similare modelului din capitolul precedent, dar oferă mai multe caracteristici și flexibilitate. Nu vom detalia TPDU-urile existente; această discuție mai are de așteptat până în momentul când vom studia TCP, mai târziu, în acest capitol.

Primitiva	Funcția
SOCKET	Creează un nou punct de capăt al comunicației
BIND	Atașează o adresă locală la un soclu
LISTEN	Anunță capacitatea de a accepta conexiuni; determină mărimea cozii
ACCEPT	Blochează apelantul până la sosirea unei cereri de conexiune
CONNECT	Tentativă (activă) de a stabili o conexiune
SEND	Trimite date prin conexiune
RECEIVE	Recepționează date prin conexiune
CLOSE	Eliberează conexiunea

Fig. 6-5. Primitivele pentru socluri TCP

Primele patru primitive din tabel sunt executate, în această ordine, de către server. Primitiva SOCKET creează un nou capăt al conexiunii și alocă spațiu pentru el în tabelele entității de transport. În parametrii de apel se specifică formatul de adresă utilizat, tipul de serviciu dorit (de exemplu, flux sigur de octeți) și protocolul. Un apel SOCKET reușit întoarce un descriptor de fișier (la fel ca un apel OPEN) care va fi utilizat în apelurile următoare.

Soclurile nou create nu au încă nici o adresă. Atașarea unei adrese se face utilizând primitiva BIND. Odată ce un server a atașat o adresă unui soclu, clienții se pot conecta la el. Motivul pentru care apelul SOCKET nu creează adresa direct este că unor procese le pasă de adresa lor (de exemplu, unele folosesc aceeași adresă de ani de zile și oricine cunoaște această adresă), în timp ce altele nu.

Urmează apelul LISTEN, care alocă spațiu pentru a reține apelurile primite în cazul când mai mulți clienți încearcă să se conecteze în același timp. Spre deosebire de modelul din primul nostru exemplu, aici LISTEN nu mai este un apel blocant.

Pentru a se bloca și a aștepta un apel, serverul execută o primitivă ACCEPT. Atunci când sosește un TPDU care cere o conexiune, entitatea de transport creează un nou soclu cu aceleași proprietăți ca cel inițial și întoarce un descriptor de fișier pentru acesta. Serverul poate atunci să creeze un nou proces sau fir de execuție care va gestiona conexiunea de pe noul soclu și să aștepte în continuare cereri de conexiune pe soclul inițial. ACCEPT returnează un descriptor normal de fișier, care poate fi folosit pentru citirea și scrierea în mod standard, la fel ca pentru fișiere.

Să privim acum din punctul de vedere al clientului: și în acest caz, soclul trebuie creat folosind o primitivă SOCKET, dar primitiva BIND nu mai este necesară, deoarece adresa folosită nu mai este importantă pentru server. Primitiva CONNECT blochează apelantul și demarează procesul de conectare. Când acesta s-a terminat (adică atunci când TPDU-ul corespunzător a fost primit de la server), procesul client este deblocat și conexiunea este stabilită. Atât clientul cât și serverul pot utiliza acum primitivele SEND și RECEIVE pentru a transmite sau recepționa date folosind o conexiune duplex integral. Se pot folosi și apelurile de sistem READ și WRITE standard din UNIX, dacă nu sunt necesare opțiunile speciale oferite de SEND și RECV.

Eliberarea conexiunii este simetrică. Atunci când ambele părți au executat primitiva CLOSE, conexiunea este eliberată.

6.1.4 Un exemplu de programare cu socluri: server de fișiere pentru Internet

Ca exemplu de cum pot fi folosite apelurile pentru socluri, vom considera codurile client și server din fig. 6-6. Aici avem un server de Internet foarte primitiv împreună cu un exemplu de client care îl utilizează. Codul are multe limitări (discutate mai jos), dar în principiu codul server poate fi compilat și rulat pe orice sistem UNIX conectat la Internet. Codul client poate fi apoi compilat și rulat pe orie altă mașină UNIX din Internet, oriunde în lume. Codul client poate fi executat cu parametrii adecvați pentru a obține orice fișier la care serverul are acces pe mașina sa. Fișierul este scris la ieșirea standard, care, desigur, poate fi redirectată spre un fișier sau spre o conductă (pipe).

Să ne uităm mai întâi la codul server. Acesta începe incluzând niște „header”-e (antete) standard între care ultimele 3 conțin principalele definiții și structuri de date care se referă la Internet. Apoi urmează o definire a *SERVER_PORT* (portului de server) ca 12345. Acest număr a fost ales arbitrar. Orice număr între 1024 și 65535 va funcționa la fel de bine atâta timp cât nu este utilizat de alte procese. Bineînțeles, clientul și serverul trebuie sa folosească același port. Dacă serverul va deveni vreodată un succes de talie mondială (improbabil, știind cât de primitiv este) îi va fi asignat un port permanent sub 1024 și va apărea la *www.iana.org*.

Următoarele două linii în codul server definesc două constante necesare. Prima determină dimensiunea zonei de memorie folosite pentru transferul de fișiere. A doua determină cât de multe conexiuni în așteptare pot fi reținute înainte ca cele care urmează să fie înlăturate după sosire.

După declarațiile variabilelor locale începe codul server. Acesta pornește cu inițializarea unei structuri de date care va ține adresa IP a serverului. Această structură de date va fi în curând asociată cu soclul serverului. Apelul către *memset* setează structura de date la 0. Cele trei atribuiri care îi urmează completează trei din câmpurile sale. Ultima dintre ele conține portul serverului. Funcțiile *htonl* și *htons* se referă la conversia valorilor într-un format standard astfel încât codul să ruleze corect atât pe mașini „big-endian” (de exemplu SPARC) cât și mașini „little-endian” (de exemplu Pentium). Semantica lor exactă nu este relevantă aici.

```
/* Această pagină conține un program client care poate cere un fișier de la programul server
de pe pagina următoare. Serverul răspunde trimițând întregul fișier.
*/

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

#define SERVER_PORT 12345       /* arbitrar, dar clientul și serverul trebuie să fie de acord */
#define BUF_SIZE 4096                                   /* dimensiunea blocului de transfer */

int main(int argc, char **argv)
{
 int c, s, bytes;
 char buf[BUF_SIZE];          /* zona tampon de memorie pentru fișierul ce este recepționat */
 struct hostent *h;

                                                        /* informații despre server */
 struct sockaddr_in channel;
                                                            /* păstrează adresa IP */

 if (argc != 3) fatal("Usage: client server-name file-name");
 h = gethostbyname(argv[1]);
                                                        /* caută adresa IP a gazdei */
 if (!h) fatal("gethostbyname failed");

 s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
 if (s < 0) fatal("socket");
 memset(&channel, 0, sizeof(channel));
 channel.sin_family= AF_INET;
 memcpy(&channel.sin_addr.s_addr, h->h_addr, h->h_length);
 channel.sin_port= htons(SERVER_PORT);

 c = connect(s, (struct sockaddr *) &channel, sizeof(channel));
 if (c < 0) fatal("connect failed");

 /* Conexiunea este acum stabilită. Trimite numele fișierului incluzând terminatorul de șir */
 write(s, argv[2], strlen(argv[2])+1);

 /* la fișierul și-l afișează la ieșirea standard. */
 while (1) {
    bytes = read(s, buf, BUF_SIZE);                             /* citește de la soclu */
    if (bytes <= 0) exit(0);                          /* verifică dacă este sfârșit de fișier*/
    write(1, buf, bytes);
                                                        /* scrie la ieșirea standard */
 }
}

fatal(char *string)
{
 printf("%s\n", string);
 exit(1);
}
```

Fig. 6-6. Codul client folosind socluri. Codul server este pe pagina următoare.

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

#define SERVER_PORT 12345       /* arbitrar, dar clientul și serverul trebuie să fie de acord */
#define BUF_SIZE 4096                                   /* dimensiunea blocului de transfer */
#define QUEUE_SIZE 10

int main(int argc, char *argv[])
{
 int s, b, l, fd, sa, bytes, on = 1;
 char buf[BUF_SIZE];          /* zona tampon de memorie pentru fișierul care este transmis */
 struct sockaddr_in channel;
                                                                  /* păstrează adresa IP */

 /* Construiește structura adresei pentru a se lega la soclu. */
 memset(&channel, 0, sizeof(channel));                                  /* canalul zero */
 channel.sin_family = AF_INET;
 channel.sin_addr.s_addr = htonl(INADDR_ANY);
 channel.sin_port = htons(SERVER_PORT);

 /* Deschidere pasivă. Așteaptă conexiunea. */
 s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);                        /* creează soclu */
 if (s < 0) fatal("socket failed");
 setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *) &on, sizeof(on));

 b = bind(s, (struct sockaddr *) &channel, sizeof(channel));
 if (b < 0) fatal("bind failed");

 l = listen(s, QUEUE_SIZE);                                  /* specifică dimensiunea cozii */
 if (l < 0) fatal("listen failed");

 /* Soclul este acum setat și legat. Așteaptă conexiunea și o procesează. */
 while (1) {
    sa = accept(s, 0, 0);                              /* blocare pentru cererea de conexiune */
    if (sa < 0) fatal("accept failed");

    read(sa, buf, BUF_SIZE);                          /* citește numele fișierului de la soclu */

    /* Preia și returnează fișierul. */
    fd = open(buf, O_RDONLY);                          /* deschide fișierul de trimis înapoi */
    if (fd < 0) fatal("open failed");

    while (1) {
       bytes = read(fd, buf, BUF_SIZE); /* citește din fișier */
       if (bytes <= 0) break;                          /* verifică dacă este sfârșit de fișier */
       write(sa, buf, bytes);                                  /* scrie octeți la soclu */
     }
     close(fd);                                                        * închide fișierul */
    close(sa);                                                      /* închide conexiunea*/
 }
}
```

Fig. 6-6. Codul server. Codul client este pe pagina anterioară.

În continuare serverul creează un soclu și face verificare pentru erori (indicate de $s<0$). Într-o versiune de producție a codului mesajul de eroare ar putea fi mai explicit. Apelul către *setsockopt* este necesar pentru a permite portului să fie folosit de serverul care rulează la nesfârșit, răspunzând la cerere după cerere. Acum, adresa IP este legată la soclu și este făcută o verificare pentru a vedea dacă apelul către *bind* a reușit. Ultimul pas în inițializare este apelul către *listen* pentru a anunța acordul serverului de a accepta apeluri și pentru a spune sistemului să mențină un număr de până la *QUEUE_SIZE* din acestea în caz că ajung noi cereri în timp ce serverul o procesează încă pe cea curentă. Dacă coada este plină și ajung cereri suplimentare, se renunță la acestea.

În acest punct, serverul intră în bucla sa principală, pe care nu o mai părăsește niciodată. Singura cale de a-l opri este de a-l termina forțat din afară. Apelul la *accept* blochează serverul până când un client încearcă să stabilească o conexiune cu el. Dacă apelul *accept* e făcut cu succes, returnează un descriptor de fișier care poate fi folosit pentru citire și scriere, în mod asemănător descriptorilor ce sunt folosiți pentru a citi și a scrie în pipe-uri (tuburi). Cu toate acestea, spre deosebire de tuburi, care sunt unidirecționale, soclurile sunt bidirecționale, astfel că *sa* (socket address – adresa soclului) poate fi folosită și pentru citire din conexiune, și pentru a scrie pe ea.

După ce conexiunea este stabilită, serverul citește numele fișierului din ea. Dacă numele nu este încă disponibil, serverul se blochează așteptându-l. După ce ia numele fișierului, serverul deschide fișierul și intră într-o buclă care citește alternativ blocuri din fișier și le scrie pe soclu până când întregul fișier a fost copiat. Apoi serverul închide iar fișierul și conexiunea și așteaptă să apară următoarea conexiune. El repetă această buclă la infinit.

Acum să privim codul client. Pentru a înțelege cum funcționează, este necesar să înțelegem cum este invocat. Presupunând că este numit *client*, un apel tipic este

```
client flits.cs.vu.nl /usr/tom/filename >f
```

Acest apel funcționează doar dacă serverul rulează deja la *flits.cs.vu.nl* și fișierul */usr/tom/filemane* există și serverul are drept de citire pentru el. Dacă apelul are succes, fișierul este transferat prin Internet și scris în *f* , după care programul client se termină. Din moment ce serverul continuă după un transfer, clientul poate fi pornit din nou pentru a lua alte fișiere.

Codul client începe cu câteva directive *include* și declarații. Execuția începe verificând dacă a fost apelat cu număr corect de argumente (*argc*=3 înseamnă numele programului plus două argumente). Observați că *argv[1]*conține numele serverului (de exemplu *flits.cs.vu.nl*) și este convertit la o adresă IP către *gethostbyname*. Această funcție folosește DNS pentru a căuta numele. Vom studia DNS în cap. 7.

În continuare este creat și inițializat un soclu. Apoi, clientul încearcă să stabilească o conexiune TCP cu serverul, folosind *connect*. Dacă serverul funcționează pe mașina menționată și atașat la *SERVER_PORT* și este fie inactiv, fie are loc în coada sa *listen*, conexiunea va fi (în cele din urmă) stabilită. Folosind conexiunea, clientul trimite numele fișierului scriind pe soclu. Numărul de octeți trimiși este cu 1 mai mare decât numele, deoarece terminatorul de șir (un octet 0) trebuie de asemenea trimis pentru a spune serverului unde se sfârșește numele.

Acum clientul intră într-o buclă, citind fișierul bloc cu bloc de la soclu și copiindu-l la ieșirea standard. Când acestea se termină, pur și simplu iese.

Procedura *fatal* afișează un mesaj de eroare și iese. Serverul are nevoie de aceeași procedură, dar aceasta a fost omisă datorită lipsei de spațiu pe pagina. Din moment ce clientul și serverul sunt compilate separat și în mod normal rulează pe calculatoare diferite, ele nu pot partaja codul procedurii *fatal*.

Aceste două programe (la fel ca și orice material referitor la această carte) pot fi luate de la adresa de Web a cărții

http://www.prenhall.com/tanenbaum

dând clic pe link-ul către situl Web de lângă fotografia coperții. Ele pot fi descărcate și compilate pe orice sisteme UNIX (de exemplu Solaris, BSD, Linux) cu comenzile:

```
cc –o client client.c –lsocket –lnsl
cc –o server sever.c –lsocket –lnsl
```

Serverul este pornit tastând doar

```
server
```

Clientul are nevoie de două argumente, așa cum s-a discutat mai sus. O versiune de Windows este de asemenea disponibilă pe situl Web.

Ca o observație, acest server nu este ultimul cuvânt în domeniul programelor server. Verificarea erorilor este ineficientă și raportarea erorilor este mediocră. În mod clar serverul nu a auzit niciodată de securitate, și folosirea doar a apelurilor de sistem UNIX nu este ultimul cuvânt în independența de platformă. De asemenea face unele presupuneri care sunt tehnic ilegale, cum ar fi presupunerea că numele fișierului începe în zona de memorie tampon și este transmis automat. Din moment ce tratează toate cererile strict secvențial (deoarece are doar un singur fir de execuție) performanța este slabă. În ciuda acestor neajunsuri, este un server de fișiere Internet complet și funcțional. În exerciții, cititorul este invitat să le îmbunătățească. Pentru mai multe informații despre programare cu socluri, a se vedea (Stevens, 1997).

6.2 NOȚIUNI DE BAZĂ DESPRE PROTOCOALELE DE TRANSPORT

Serviciul transport este implementat prin intermediul unui **protocol de transport** folosit de cele două entități de transport. Câteva caracteristici sunt asemănătoare pentru protocoalele de transport și pentru cele de legătură de date studiate în detaliu în cap. 3. Amândouă trebuie să se ocupe, printre altele, de controlul erorilor, de secvențiere și de controlul fluxului.

Totuși, există diferențe semnificative între cele două protocoale. Aceste diferențe sunt datorate deosebirilor majore dintre mediile în care operează protocoalele, așa cum rezultă din fig. 6-7. La nivelul legăturii de date, cele două rutere comunică direct printr-un canal fizic, în timp ce la nivelul transport acest canal fizic este înlocuit de întreaga subrețea. Această deosebire are mai multe implicații importante pentru protocoale, așa cum vom vedea în acest capitol.

În cazul legăturii de date, pentru un ruter nu trebuie specificat cu care alt ruter vrea să comunice, deoarece fiecare linie specifică în mod unic o destinație. În schimb, în cazul nivelului transport este necesară adresarea explicită.

În plus, procesul stabilirii unei conexiuni prin cablul din fig. 6-7(a) este simplu: celălalt capăt este întotdeauna acolo (în afară de cazul în care nu a ‘căzut’) și în nici unul din cazuri nu sunt prea multe de făcut. Pentru nivelul transport însă, stabilirea inițială a conexiunii este mult mai complicată, așa cum vom vedea.

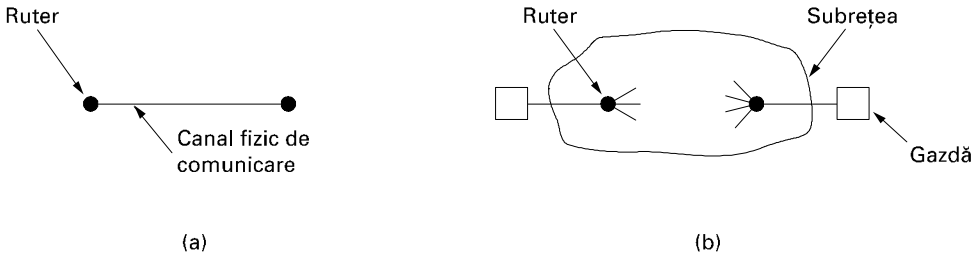


Fig. 6-7. (a) Mediul pentru nivelul legătură de date. (b) Mediul pentru nivelul transport.

O altă diferență între nivelurile legătură de date și transport, care generează multe probleme, este existența potențială a unei capacități de memorare a subrețelei. Atunci când un ruter trimite un cadru (nivel legătură de date), acesta poate să ajungă sau poate să se piardă, dar nu poate să se plimbe un timp ajungând până la capătul lumii și să apară 30 de secunde mai târziu, într-un moment nepotrivit. Dacă subrețeaua folosește datagrame și dirijare adaptivă, există o posibilitate - care nu poate fi neglijată - ca un pachet să fie păstrat pentru un număr oarecare de secunde și livrat mai târziu. Consecințele capacității de memorare a subrețelei pot fi uneori dezastruoase și necesită folosirea unor protocoale speciale.

O ultimă diferență între nivelurile legătură de date și transport este una de dimensionare și nu de proiectare. Folosirea tampoanelor și controlul fluxului sunt necesare la amândouă nivelurile, dar prezența unui număr mare de conexiuni în cazul nivelului transport necesită o abordare diferită de cea de la nivelul legătură de date. În cap. 3, unele protocoale alocau un număr fix de tampoane pentru fiecare linie, astfel încât atunci când sosea un cadru, exista întotdeauna un tampon disponibil. La nivel transport, numărul mare de conexiuni care trebuie să fie gestionate face ca ideea de a aloca tampoane dedicate să fie mai puțin atractivă. În următoarele secțiuni, vom examina atât aceste probleme importante cât și altele.

6.2.1 Adresarea

Atunci când un proces aplicație (de exemplu, un proces utilizator) dorește să stabilească o conexiune cu un proces aflat la distanță, el trebuie să specifice cu care proces dorește să se conecteze. (La protocoalele de transport neorientate pe conexiune apare aceeași problemă: cui trebuie trimis mesajul?). Metoda folosită în mod normal este de a defini adrese de transport la care procesele pot să aștepte cereri de conexiune. În Internet acestea se numesc porturi. La rețelele ATM perechile se numesc **AAL - SAP**-uri. În continuare vom folosi pentru acestea termenul generic **TSAP (Transport Service Access Point -** punct de acces la serviciul de transport). Punctele similare în cazul nivelului rețea (adică adresele la nivel rețea) sunt numite **NSAP (Network Service Access Point).** Adresele IP sunt exemple de NSAP-uri.

Fig. 6-8 ilustrează relația între TSAP, NSAP și conexiunile transport. Procesele aplicație, atât clienții cât și serverele, se pot atașa la TSAP pentru a stabili o conexiune la un TSAP la distanță. Aceste conexiuni rulează prin TSAP-uri pe fiecare gazdă așa cum se arată. Necesitatea de a avea mai multe TSAP-uri este dată de faptul că în unele rețele, fiecare calculator are un singur NSAP, deci cumva este nevoie să se distingă mai multe puncte de sfârșit de transport care partajează acel NSAP.

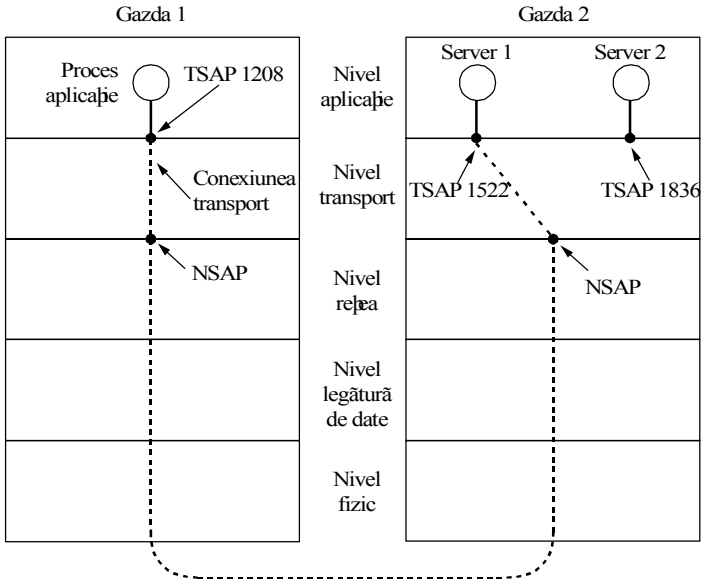


Fig. 6-8. TSAP, NSAP și conexiunile la nivel transport.

Un scenariu posibil pentru stabilirea unei conexiuni la nivel transport este următorul.

1. Un proces server care furnizează ora exactă și care rulează pe gazda 2 se atașează la TSAP 122 așteptând un apel. Felul în care un proces se atașează la un TSAP nu face parte din modelul de rețea și depinde numai de sistemul de operare local. Poate fi utilizat un apel de tip LISTEN din capitolul precedent.
2. Un proces aplicație de pe gazda 1 dorește să afle ora exactă; atunci el generează un apel CONNECT specificând TSAP 1208 ca sursă și TSAP 1522 ca destinație. Această acțiune are ca rezultat în cele din urmă stabilirea unei conexiuni la nivel transport între procesele aplicație de pe gazda 1 și serverul 1 de pe gazda 2.
3. Procesul aplicație trimite o cerere o cerere pentru timp.
4. Procesul server de timp răspunde cu timpul curent.
5. Conexiunea transport este apoi eliberată.

Trebuie reținut că foarte bine pot exista alte servere pe gazda 2 care să fie atașate la alte TSAP-uri și care să aștepte conexiuni care ajung pe același NSAP.

Fig. 6-8 explică aproape tot, cu excepția unei mici probleme: cum știe procesul utilizator de pe mașina 1 că serverul de oră exactă este atașat la TSAP 1522? O posibilitate este ca acest server de oră exactă să se atașeze la TSAP 1522 de ani de zile și, cu timpul, toți utilizatorii au aflat acest lucru. În acest model serviciile au adrese TSAP fixe, care pot fi afișate în fișiere în locuri bine cunoscute, cum este fișierul *etc/services* pe sistemele UNIX care afișează ce servere sunt atașate permanent și la ce porturi.

Dar schema cu adrese de servicii fixe funcționează doar pentru un număr mic de servicii cheie, a căror adresă nu se schimbă niciodată (de exemplu server de Web). Însă, în general, procesele utilizator vor să comunice cu alte procese care există numai pentru scurt timp și nu au o adresă TSAP dinainte cunoscută. Pe de altă parte, pot exista mai multe procese server, majoritatea utilizate foarte

rar, și ar fi neeconomic ca fiecare să fie activ și să asculte la o adresă TSAP fixă tot timpul. Pe scurt, este necesară o soluție mai bună.

O astfel de soluție este prezentată în fig. 6-9, într-o formă simplificată. Ea este cunoscută ca **protocolul de conectare inițială.** În loc ca orice server să asculte la un TSAP fixat, fiecare mașină care dorește să ofere servicii utilizatorilor aflați la distanță are un **server de procese (process server)** special care acționează ca un intermediar pentru toate serverele mai puțin utilizate. El ascultă în același timp la un număr de porturi, așteptând o cerere de conexiune. Utilizatorii potențiali ai serviciului încep prin a face o cerere de conexiune, specificând adresa TSAP a serviciului pe care îl doresc. Dacă nu există un server care să aștepte conexiuni la acel port, ele obțin o conexiune la serverul de procese, ca în fig. 6-9 (a).

După ce primește cererea, serverul de procese dă naștere serverului cerut, permițându-i să moștenească conexiunea cu procesul utilizator. Noul server execută prelucrarea cerută, în timp ce serverul de procese continuă să aștepte noi cereri, ca în fig. 6-9 (b).

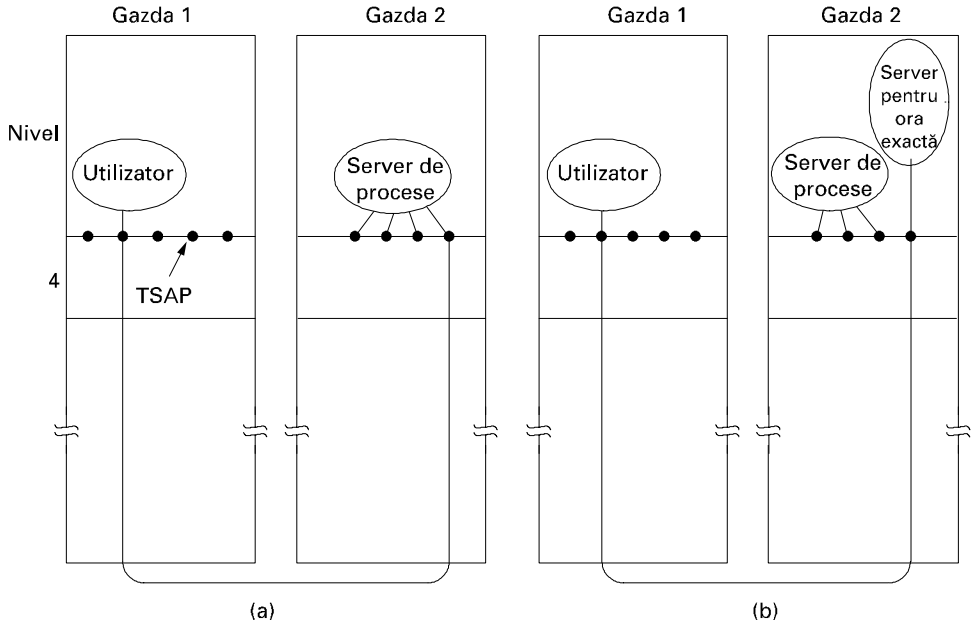


Fig. 6-9. Stabilirea unei conexiuni între calculatorul gazdă 1 și serverul pentru ora exactă.

În timp ce acest protocol funcționează bine pentru serverele care pot fi create ori de câte ori este nevoie de ele, există mai multe situații în care serviciile există independent de serverul de procese. De exemplu, un server de fișiere va rula folosind un hardware specializat (o mașina cu disc) și nu poate fi creat din mers.

Pentru a trata această situație, este des utilizată o soluție alternativă. În acest model există un proces special numit **server de nume** (**name server** sau, uneori, **directory server**). Pentru a găsi adresa TSAP corespunzătoare unui serviciu dat prin nume, așa cum este „ora exactă", utilizatorul stabilește o conexiune cu serverul de nume (care așteaptă mesaje la un TSAP cunoscut). Apoi utilizatorul trimite un mesaj specificând numele serviciului, iar serverul de nume îi trimite înapoi adresa TSAP a

acestuia. După aceasta, utilizatorul eliberează conexiunea cu serverul de nume și stabilește o nouă conexiune cu serviciul dorit.

În acest model, atunci când este creat un nou serviciu, el trebuie să se înregistreze singur la serverul de nume, furnizând atât numele serviciului oferit (în general un șir ASCII) cât și adresa TSAP. Serverul de nume înregistrează această informație într-o bază de date internă, astfel încât el va ști răspunsul atunci când vor sosi noi cereri.

Funcționarea serverului de nume este asemănătoare cu serviciul de informații de la un sistem telefonic: este furnizată corespondența dintre nume și numere de telefon. Ca și în cazul telefoanelor, este esențial ca adresa bine cunoscută a serverului de nume (sau a serverului de procese, în protocolul de conectare inițială) să fie într-adevăr bine cunoscută. Dacă nu știi numărul de la informații, nu poți afla nici un alt număr de telefon. Dacă crezi că numărul de la informații este evident pentru toți, încearcă să-l folosești și în altă țară!

6.2.2 Stabilirea conexiunii

Stabilirea unei conexiuni poate să pară ușoară dar, în realitate, este surprinzător de complicată. La prima vedere, ar părea suficient ca o entitate de transport să trimită numai un TPDU CONNECTION REQUEST și să aștepte replica CONNECTION ACCEPTED . Problema apare deoarece rețeaua poate pierde, memora sau duplica pachete. Acest comportament duce la complicații serioase.

Putem imagina o subrețea care este atât de congestionată încât confirmările ajung greu înapoi, și, din această cauză, fiecare pachet ajunge să fie retransmis de câteva ori. Putem presupune că subrețeaua folosește datagrame și fiecare pachet urmează un traseu diferit. Unele pachete pot să întâlnească o congestie locală de trafic și să întârzie foarte mult, ca și cum ar fi fost memorate de subrețea un timp și eliberate mai târziu.

Cel mai neplăcut scenariu ar fi: un utilizator stabilește o conexiune cu o bancă și trimite un mesaj cerând transferul unei sume de bani în contul unei alte persoane în care nu poate avea încredere în totalitate, și apoi eliberează conexiunea. Din nefericire, fiecare pachet din acest scenariu este duplicat și memorat în subrețea. După ce conexiunea a fost eliberată, pachetele memorate ies din subrețea și ajung la destinatar, cerând băncii să stabilească o nouă conexiune, să facă transferul (încă o dată) și să elibereze conexiunea. Banca nu poate să știe că acestea sunt duplicate, ea trebuie să presupună că este o tranzacție independentă și va transfera banii încă o dată. În continuarea acestei secțiuni, vom studia problema duplicatelor întârziate, punând accentul în mod special pe algoritmii pentru stabilirea sigură a conexiunilor, astfel încât scenarii ca cel de mai sus să nu poată să apară.

După cum am mai spus, punctul crucial al problemei este existența duplicatelor întârziate. El poate fi tratat în mai multe feluri, dar nici unul nu este într-adevăr satisfăcător. O posibilitate este de a utiliza adrese de transport valabile doar pentru o singură utilizare. În această abordare, ori de câte ori este necesară o adresă la nivel transport, va fi generată una nouă. După ce conexiunea este eliberată, adresa nu mai este folosită. Acest mecanism face însă imposibl modelul cu server de procese din fig. 6-9.

O altă posibilitate este de a atribui fiecărei conexiuni un identificator (adică, un număr de secvență incrementat pentru fiecare conexiune stabilită), ales de cel care inițiază conexiunea, și pus în fiecare TPDU, inclusiv în cel care inițiază conexiunea. După ce o conexiune este eliberată, fiecare entitate de transport va completa o tabelă cu conexiunile care nu mai sunt valide, reprezentate ca perechi (entitate de transport, identificator conexiune). Ori de câte ori apare o cerere de conexiune se va verifica în tabelă că ea nu aparține unei conexiuni care a fost eliberată anterior.

Din nefericire, această schemă are un defect important: ea necesită ca fiecare entitate de transport să mențină informația despre conexiunile precedente un timp nedefinit. Dacă o mașină cade și își pierde datele din memorie, ea nu va mai ști care identificatori de conexiune au fost deja utilizați.

Putem încerca și o altă soluție. În loc să permitem pachetelor să trăiască la nesfârșit în subrețea, putem inventa un mecanism care să elimine pachetele îmbătrânite. Dacă suntem siguri că nici un pachet nu poate să supraviețuiască mai mult de un anume interval de timp cunoscut, problema devine ceva mai ușor de rezolvat.

Durata de viață a pachetelor poate fi limitată la un maxim cunoscut, folosind una (sau mai multe) din următoarele tehnici:

1. Restricții în proiectarea subrețelei
2. Adăugarea unui contor al nodurilor parcurse în fiecare pachet
3. Adăugarea unei amprente de timp la fiecare pachet

Prima metodă include soluțiile care împiedică pachetele să stea în buclă, combinate cu modalități de a limita întârzierile datorate congestiilor, pe orice cale din rețea (indiferent de lungime). A doua metodă constă în a inițializa contorul cu o valoare adecvată și în a-l decrementa la trecerea prin orice nod. Protocolul de nivel rețea pur și simplu elimină pachetele al căror contor a devenit zero. A treia metodă presupune ca fiecare pachet să conțină timpul creării sale, ruterele acceptând să elimine pachetele mai vechi de un anumit moment de timp, asupra căruia au căzut de acord. Această metodă necesită ca ceasurile de la fiecare ruter să fie sincronizate, și această cerință în sine este destul de greu de îndeplinit (mai ușor este dacă sincronizarea ceasurilor se obține din exteriorul rețelei, de exemplu folosind GPS sau stații radio care transmit periodic ora exactă).

În practică, nu este suficient doar să garantăm că pachetul este eliminat, ci trebuie garantat și că toate confirmările sale au fost eliminate, astfel încât vom introduce T, care va fi un multiplu (mic) al duratei maxime de viață a unui pachet. Depinde de protocol de câte ori T este mai mare decât durata de viață a unui pachet. Dacă așteptăm un timp T după trimiterea unui pachet putem fi siguri că toate urmele sale au dispărut și nici el, nici vreo confirmare de-a sa nu vor apărea din senin, doar ca să complice lucrurile.

Folosind durata de viață limitată a pachetelor, există metode de a obține conexiuni sigure a căror corectitudine a fost demonstrată. Metoda descrisă în cele ce urmează este datorată lui Tomlinson (1975). Ea rezolvă problema, dar introduce câteva particularități proprii. Metoda a fost îmbunătățită de Sunshine și Dalal (1978). Variante ale sale sunt larg folosite în practică, inclusiv în TCP.

Pentru a ocoli problemele generate de pierderea tuturor datelor din memoria unei mașini după o cădere, Tomlinson propune echiparea fiecărei mașini cu un ceas. Nu este nevoie ca ceasurile de pe mașini diferite să fie sincronizate. Fiecare ceas va fi de fapt un contor binar care se autoincrementează după un anumit interval de timp. În plus, numărul de biți ai contorului trebuie să fie cel puțin egal cu numărul de biți al numerelor de secvență. În cele din urmă, și cel mai important, ceasul trebuie să continue să funcționeze chiar în cazul în care calculatorul gazdă cade.

Ideea de bază este de a fi siguri că două TPDU numerotate identic nu pot fi generate în același timp. Atunci când conexiunea este inițiată, k biți mai puțin semnificativi ai ceasului sunt folosiți ca număr inițial de secvență (tot k biți). Astfel, fiecare conexiune începe să-și numeroteze TPDU-urile sale cu un număr de secvență diferit. Spațiul numerelor de secvență ar trebui să fie suficient de mare pentru ca, în timpul scurs până când contorul ajunge din nou la același număr, toate TPDU-urile vechi cu acel număr să fi dispărut deja. Această relație liniară între timp și numărul de secvență inițial este prezentată în fig. 6-10.

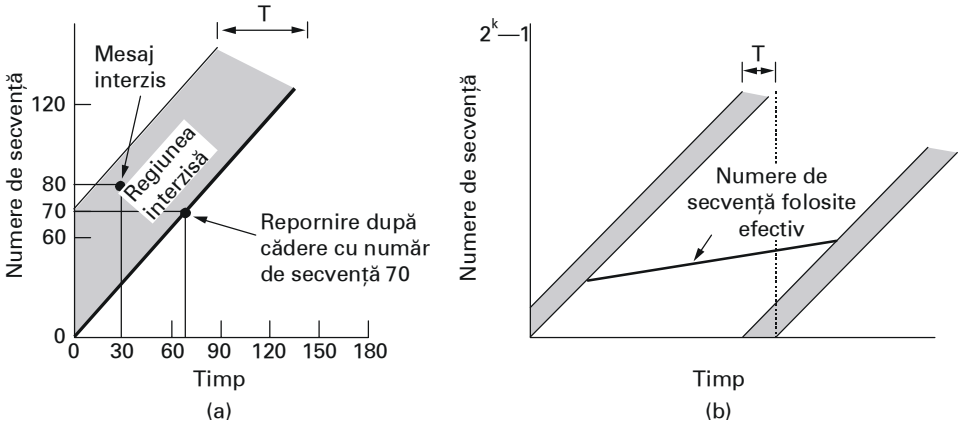


Fig. 6-10. (a) TPDU-urile nu pot să intre în regiunea interzisă. (b) Problema resincronizării.

Odată ce ambele entități de transport au căzut de acord asupra numărului de secvență inițial, pentru controlul fluxului poate fi folosit orice protocol cu fereastră glisantă. În realitate curba ce reprezintă numărul inițial de secvență (desenată cu linie îngroșată) nu este chiar liniară, ci în trepte, căci ceasul avansează în trepte. Pentru simplitate, vom ignora acest detaliu.

O problemă apare atunci când cade un calculator gazdă. Când el își revine, entitatea sa de transport nu știe unde a rămas în spațiul numerelor de secvență. O soluție este de a cere entității de transport să stea neocupată T secunde după revenire pentru ca în acest timp toate vechile TPDU să dispară. Totuși, într-o rețea complexă T poate fi destul de mare, astfel că această strategie nu este prea atrăgătoare.

Pentru a evita cele T secunde de timp nefolosit după o cădere, este necesar să introducem o nouă restricție în utilizarea numerelor de secvență. Necesitatea introducerii acestei restricții este evidentă în următorul exemplu. Fie T, timpul maxim de viață al unui pachet, egal cu 60 de secunde și să presupunem că ceasul este incrementat la fiecare secundă. După cum arată linia îngroșată din fig. 6-10(a), numărul inițial de secvență pentru o conexiune inițiată la momentul x este x. Să ne imaginăm că la t=30 sec, unui TPDU trimis pe conexiunea cu numărul 5 (deschisă anterior) i se dă numărul de secvență 80. Să numim acest TPDU *X.* Imediat după ce *X* este trimis, calculatorul gazdă cade și revine imediat. La t=60 el redeschide conexiunile de la 0 la 4. La t=70, el deschide conexiunea 5, folosind un număr de secvență inițial 70, așa cum am stabilit. În următoarele 15 secunde el va transmite TPDU-uri cu date numerotate de la 70 la 80. Astfel că la t=85, în subrețea este generat un nou TPDU cu numărul de secvență 80 și conexiunea 5. Din nefericire, TPDU *X* încă mai există. Dacă el ajunge înaintea noului TPDU 80, atunci TPDU *X* va fi acceptat și TPDU-ul corect va fi respins ca fiind un duplicat.

Pentru a preveni o astfel de problemă trebuie să luăm măsuri ca numerele de secvență să nu fie utilizate (adică atribuite unor noi TPDU-uri) un timp T înaintea utilizării lor ca noi numere de secvență. Combinațiile imposibile - timp, număr de secvență - sunt prezentate în fig. 6-10(a) ca **regiunea interzisă**. Înainte de trimiterea oricărui TPDU pe orice conexiune, entitatea de transport trebuie să citească ceasul și să verifice dacă nu cumva se află în regiunea interzisă.

Pot să apară probleme în două cazuri: dacă un calculator gazdă trimite prea multe date și prea repede pe o conexiune nou deschisă, curba numărului de secvență în funcție de timp poate să fie mult

mai abruptă decât cea inițială. Aceasta înseamnă că rata de transmisie pentru orice conexiune este de cel mult un TPDU pe unitatea de timp a ceasului. De asemenea, este necesar ca entitatea de transport să aștepte până când ceasul avansează o dată, înainte să deschidă o nouă conexiune pentru ca, la revenirea după o cădere, același număr de secvență să nu fie utilizat de două ori. Cele două observații de mai sus sunt argumente pentru ca perioada ceasului să fie cât mai mică (câteva μs sau mai mică).

Din nefericire, intrarea în regiunea interzisă prin trimitere prea rapidă nu este singura situație care creează probleme. Fig. 6-10(b) arată că la orice rată de transfer mai mică decât frecvența ceasului curba numerelor de secvență utilizate raportată la timp va ajunge până la urmă în regiunea interzisă din stânga. Cu cât curba numerelor de secvență utilizate va fi mai înclinată, cu atât mai târziu se ajunge în regiunea interzisă. Așa cum am afirmat anterior, imediat înaintea trimiterii unui TPDU, entitatea de transport trebuie să verifice dacă nu se află cumva în regiunea interzisă, și, dacă se află, să întârzie transmisia cu T secunde sau să resincronizeze numerele de secvență.

Metoda bazată pe ceasuri rezolvă problema duplicatelor întârziate pentru TPDU-urile de date, dar pentru ca această metodă să poată fi folosită, trebuie mai întâi să stabilim conexiunea. Deoarece TPDU-urile de control pot și ele să fie întârziate, pot apărea probleme atunci când entitățile de transport încercă să cadă de acord asupra numărului inițial de secvență. Să presupunem, de exemplu, că, pentru a stabili o conexiune, gazda 1 trimite un mesaj CONNECTION REQUEST conținând numărul de secvență inițial propus și portul destinație gazdei 2. Acesta va confirma mesajul trimițând înapoi un TPDU CONNECTION ACCEPTED . Dacă TPDU-ul CONNECTION REQUEST este pierdut, dar un duplicat întârziat al unui alt CONNECTION REQUEST va ajunge la gazda 2, atunci conexiunea nu va fi stabilită corect.

Pentru a rezolva aceasta problemă, Tomlinson (1975) a introdus stabilirea conexiunii cu înțelegere în trei pași (three-way handshake). Acest protocol nu necesită ca ambele părți să înceapă să trimită același număr de secvență, deci poate fi utilizat și împreună cu alte metode de sincronizare decât ceasul global. Procedura normală de inițiere a conexiunii este exemplificată în fig. 6-11(a). Gazda 1 alege un număr de secvență *x* și trimite un TPDU CONNECTION REQUEST care conține *x* gazdei 2. Gazda 2 răspunde cu CONNECTION ACK, confirmând *x* și anunțând numărul său inițial de secvență, *y*. În cele din urmă gazda 1 confirmă alegerea lui *y* gazdei 2 în primul mesaj de date pe care îl trimite.

Vom arunca acum o privire asupra stabilirii conexiunii cu înțelegere în trei pași în prezența TPDU-urilor de control duplicate întârziate. În fig. 6-11(b) primul TPDU sosit este o copie întârziată a unui CONNECTION REQUEST de la o conexiune mai veche. Acest TDU ajunge la gazda 2 fără ca gazda 1 să știe. Gazda 2 răspunde acestui TPDU trimițând gazdei 1 un TPDU ACK, verificând de fapt că gazda 1 a încercat într-adevăr să stabilească o conexiune. Atunci când gazda 1 refuză cererea gazdei 2 de a stabili conexiunea, gazda 2 își dă seama că a fost păcălită de o copie întârziată și abandonează conexiunea. În acest fel o copie întârziată nu poate să strice nimic.

În cel mai rău caz, atât CONNECTION REQUEST cât și ACK sunt copii întârziate în subrețea. Acest caz este prezentat în 6-11(c). Ca și în exemplul precedent, gazda 2 primește o comandă CONNECTION REQUEST întârziată și răspunde la ea. În acest moment este extrem de important să ne aducem aminte că gazda 2 a propus *y* ca număr inițial de secvență pentru traficul de la 2 la 1, fiind sigur că nu mai există în rețea nici un TPDU (sau confirmare) cu același număr de secvență. Atunci când al doilea TPDU întârziat ajunge la gazda 2, aceasta deduce, din faptul că a fost confirmat *z* și nu *y,* că are de-a face cu o copie mai veche. Important este că nu există nici o combinație posibilă ale unor copii vechi ale TPDU-urilor întârziate care să reușească să inițieze o conexiune atunci când nimeni nu a cerut asta.

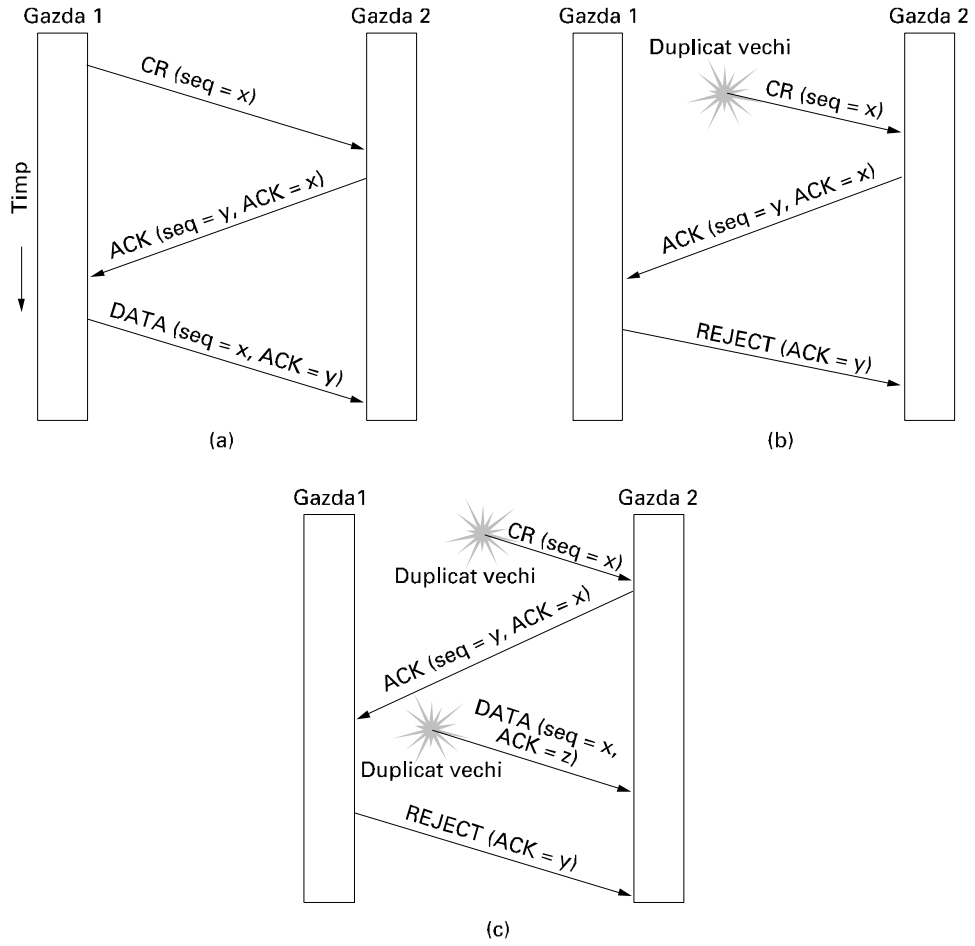


Fig. 6-11. Trei scenarii posibile de stabilire a conexiunii pentru un protocol cu înțelegere în trei pași. CR reprezintă CONNECTION REQUEST. (a) Cazul normal, (b) Un duplicat vechi al unui mesaj CONNECTION REQUEST apare când nu trebuie, (c) Sunt duplicate atât CONNECTION REQUEST cât și CONNECTION ACCEPTED.

6.2.3 Eliberarea conexiunii

Eliberarea unei conexiuni este mai ușoară decât stabilirea ei. Totuși, există mai multe dificultăți decât ne-am aștepta. Așa cum am mai amintit, există două moduri de a termina o conexiune: eliberare simetrică și eliberare asimetrică. Sistemul telefonic folosește eliberarea asimetrică: atunci când unul din interlocutori închide, conexiunea este întreruptă. Eliberarea simetrică privește conexiunea ca pe două conexiuni separate unidirecționale și cere ca fiecare să fie eliberată separat.

Eliberarea asimetrică este bruscă și poate genera pierderi de date. Să considerăm scenariul din fig. 6-12. După stabilirea conexiunii, gazda 1 trimite un TPDU care ajunge corect la gazda 2. Gazda

1 mai trimite un TPDU dar, înainte ca acesta să ajungă la destinație, gazda 2 trimite DISCONNECT REQUEST . În acest caz, conexiunea va fi eliberată și vor fi pierdute date.

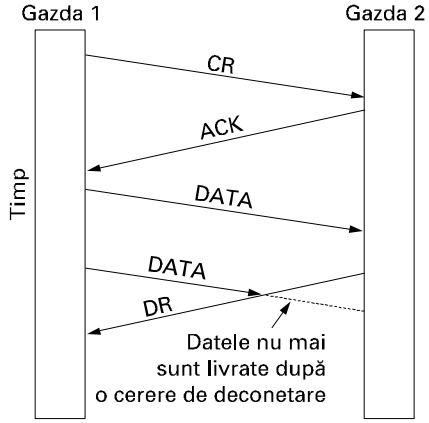


Fig. 6-12. Deconectare bruscă cu pierdere de date. CR= CONNECTION REQUEST, ACK=CONNECTION ACCEPTED , DR=DISCONNECT REQUEST.

Evident, pentru a evita pierderea de date, este necesar un protocol de eliberare a conexiunii mai sofisticat. O posibilitate este utilizarea eliberării simetrice: fiecare direcție este eliberată independent de cealaltă; un calculator gazdă poate să continue să primească date chiar și după ce a trimis un TPDU de eliberare a conexiunii.

Eliberarea simetrică este utilă atunci când fiecare proces are o cantitate fixă de date de trimis și știe bine când trebuie să transmită și când a terminat. În alte situații însă, nu este deloc ușor de determinat când trebuie eliberată conexiunea și când a fost trimis tot ce era de transmis. S-ar putea avea în vedere un protocol de tipul următor: atunci când 1 termină, trimite ceva de tipul: Am terminat. Ai terminat și tu? Dacă gazda 2 răspunde: Da, am terminat. Închidem! conexiunea poate fi eliberată în condiții bune.

Din nefericire, acest protocol nu merge întotdeauna. Binecunoscuta **problemă a celor două armate** este similară acestei situații: să ne imaginăm că armată albă și-a pus tabăra într-o vale (ca în fig. 6-13) Pe amândouă crestele care mărginesc valea sunt armatele albastre. Armata albă este mai mare decât fiecare din cele două armate albastre, dar împreună armatele albastre sunt mai puternice. Dacă oricare din armatele albastre atacă singură, ea va fi înfrântă, dar dacă ele atacă simultan, atunci vor fi victorioase.

Armatele albastre vor să-și sincronizeze atacul. Totuși singura lor posibilitate de comunicație este să trimită un mesager care să străbată valea. Mesagerul poate fi capturat de armata albă și mesajul poate fi pierdut (adică vor trebui să utilizeze un canal de comunicație nesigur). Problema este următoarea: există vreun protocol care să permită armatelor albastre să învingă?

Să presupunem că comandantul primei armate albastre trimite un mesaj: „Propun să atacăm pe 29 martie”, mesajul ajunge la armata 2 al cărei comandant răspunde: „De acord” iar răspunsul ajunge înapoi la armata 1. Va avea loc atacul în acest caz? Probabil că nu, deoarece comandantul armatei 2 nu știe dacă răspunsul său a ajuns sau nu la destinație. Dacă nu a ajuns, armata 1 nu va ataca, deci ar fi o prostie din partea lui să intre în luptă.

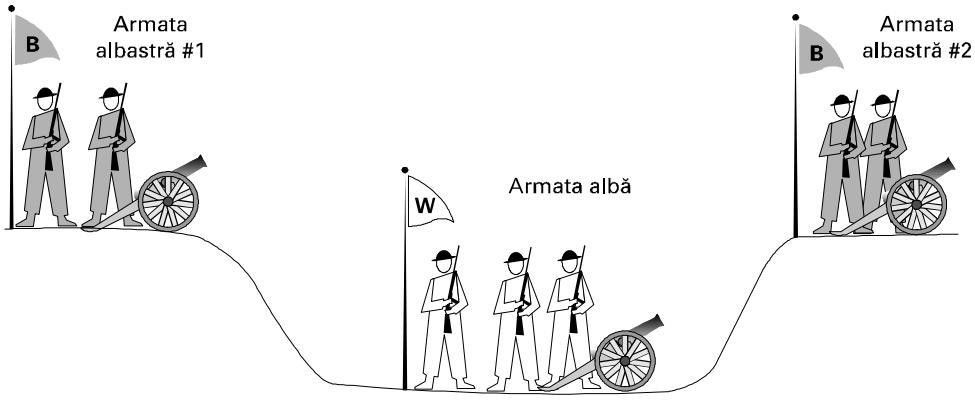


Fig. 6-13. Problema celor două armate.

Să încercăm să îmbunătățim protocolul, transformându-l într-unul cu înțelegere în trei pași. Inițiatorul propunerii de atac trebuie să confirme răspunsul. Presupunând că nici un mesaj nu este pierdut, armata 2 va avea confirmarea, dar comandantul armatei 1 va ezita acum. Până la urmă, el nu știe dacă confirmarea sa a ajuns la destinație și este sigur că dacă aceasta nu a ajuns, armata 2 nu va ataca. Am putea să încercăm un protocol cu confirmare în patru timpi, dar ne-am lovi de aceleași probleme.

De fapt, poate fi demonstrat că nu există un protocol care să funcționeze. Să presupunem că ar exista un asemenea protocol: decizia finală poate să depindă sau nu de ultimul mesaj al unui asemenea protocol. Dacă nu depinde, putem elimina acest mesaj (și oricare altul la fel) până ajungem la un protocol în care orice mesaj este vital. Ce se va întâmpla dacă ultimul mesaj este interceptat? Tocmai am hotărât că acest mesaj era unul vital, deci dacă este pierdut, atacul nu va avea loc. Deoarece cel care trimite ultimul mesaj nu poate fi niciodată sigur că mesajul a ajuns, el nu va risca atacând. Mai rău chiar, cealaltă armată albastră știe și ea acest lucru, deci nu va ataca nici ea.

Pentru a vedea legătura problemei celor două armate cu problema eliberării conexiunii este suficient să înlocuim 'atac' cu 'deconectare'. Dacă niciuna din părți nu se deconectează până nu este sigură că cealaltă parte este gata să se deconecteze la rândul ei, atunci deconectarea nu va mai avea loc niciodată.

În practică suntem dispuși să ne asumăm mai multe riscuri atunci când este vorba de eliberarea conexiunii decât atunci când este vorba de atacarea armatei albe, așa încât situația nu este întru totul fără speranță. Fig. 6-14 prezintă patru scenarii de eliberare a conexiunii folosind un protocol cu confirmare în trei timpi. Deși acest protocol nu este infailibil, el este în general adecvat.

În fig. 6-14(a) apare cazul normal în care unul dintre utilizatori trimite un TPDU de tip DR (DISCONNECT REQUEST) pentru a iniția eliberarea conexiunii. Atunci când acesta sosește, receptorul trimite înapoi tot un TPDU DR și pornește un ceas pentru a trata cazul în care mesajul său este pierdut. Când primește mesajul înapoi, inițiatorul trimite o confirmare și eliberează conexiunea. În sfârșit, la primirea confirmării, receptorul eliberează și el conexiunea. Eliberarea conexiunii înseamnă de fapt că entitatea de transport șterge din tabelele sale informația despre conexiunea respectivă din tabela de conexiuni deschise în momentul curent și semnalează acest lucru utilizatorului nivelului transport. Această acțiune nu este același lucru cu apelul unei primitive DISCONNECT de către un utilizator al nivelului transport.

Dacă ultima confirmare este pierdută, ca în fig. 6-14(b), putem salva situația cu ajutorul ceasului: după scurgerea unui anumit interval de timp conexiunea este eliberată oricum.

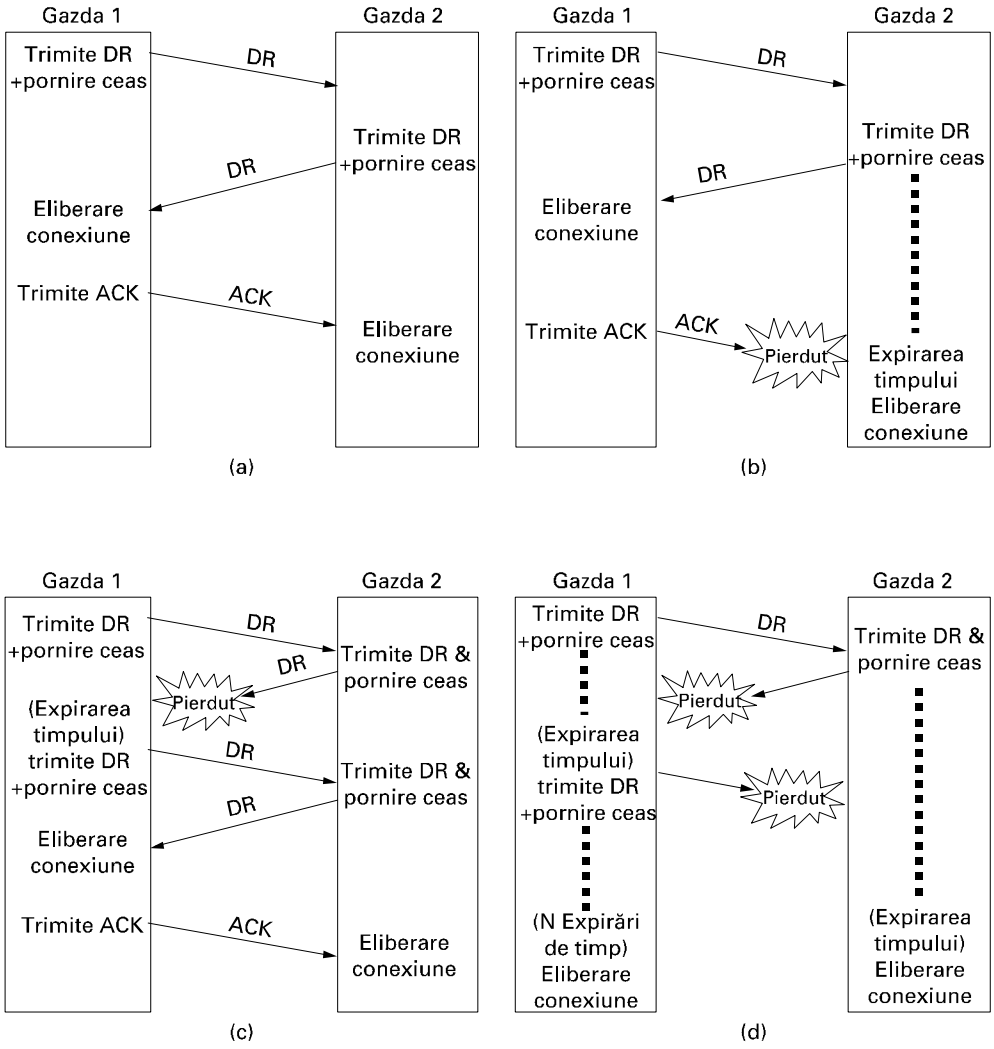


Fig. 6-14. Patru cazuri posibile la eliberarea conexiunii: (a)Cazul normal cu confirmare în trei timpi. (b)Ultima confirmare este pierdută. (c) Răspunsul este pierdut. (d) Răspunsul și următoarele cereri de deconectare sunt pierdute. (DR=DISCONNECT REQUEST).

Să considerăm acum cazul în care cel de-al doilea DR este pierdut: utilizatorul care a inițiat deconectarea nu va primi răspunsul așteptat, va aștepta un anumit timp și va trimite din nou un DR. În fig. 6-14(c), putem vedea cum se petrec lucrurile în acest caz, presupunând că la a doua încercare toate TPDU-urile ajung corect și la timp.

Ultima posibilitate pe care o studiem, prezentată în fig. 6-14(d), este similară cu cea din 6-14(c), cu următoarea diferență: de aceasta dată niciuna din încercările următoare de a retransmite DR nu reușește. După *N* încercări, emițătorul va elibera pur și simplu conexiunea. În același timp, și receptorul va elibera conexiunea după expirarea timpului.

Deși acest protocol este, în general, destul de bun, în teorie el poate să dea greș dacă atât mesajul DR inițial cât și *N* retransmisii ale sale se pierd. Emițătorul va renunța și va elibera conexiunea, în timp ce la celălalt capăt nu se știe nimic despre încercările de deconectare și aceasta va rămâne în continuare activă. În această situație rezultă o conexiune deschisă pe jumătate.

Am putea evita această problemă nepermițând emițătorului să cedeze după *N* reîncercări nereușite, ci cerându-i să continue până primește un răspuns. Totuși, dacă celeilalte părți i se permite să elibereze conexiunea după un interval de timp, este posibil ca inițiatorul să ajungă să aștepte la infinit. Dacă însă nu s-ar permite eliberarea conexiunii după expirarea unui interval de timp, atunci în cazul din fig. 6-14 (b) protocolul s-ar bloca.

O altă posibilitate de a scăpa de conexiunile pe jumătate deschise este de a aplica o regulă de tipul: dacă nici un TPDU nu sosește într-un anumit interval de timp, atunci conexiunea este eliberată automat. În acest fel, dacă una din părți se deconectează, cealaltă parte va detecta lipsa de activitate și se va deconecta și ea. Desigur, pentru a implementa aceasta regulă este nevoie ca fiecare entitate de transport să aibă un ceas care va fi repornit la trimiterea oricărui TPDU. La expirarea timpului, se transmite un TPDU vid, doar pentru a menține conexiunea deschisă. Pe de altă parte, dacă este aleasă această soluție, și câteva TPDU-uri vide sunt pierdute la rând pe o conexiune altfel liberă, este posibil ca, mai întâi una din părți, apoi cealaltă să se deconecteze automat.

Nu vom mai continua să detaliem acest subiect, dar probabil că acum este clar că eliberarea unei conexiuni fără pierderi de date nu este atât de simplă cum părea la început.

6.2.4 Controlul fluxului și memorarea temporară (buffering)

După ce am studiat în detaliu stabilirea și eliberarea conexiunii, vom arunca o privire asupra modului în care sunt tratate conexiunile cât timp sunt utilizate. Una din problemele cheie a apărut și până acum: controlul fluxului. La nivel transport există asemănări cu problema controlului fluxului la nivel legătură de date, dar există și deosebiri. Principala asemănare: la ambele niveluri este necesar un mecanism (fereastră glisantă sau altceva) pentru a împiedica un emițător prea rapid să depășească capacitatea de recepție a unui receptor prea lent. Principala deosebire: un ruter are în general puține linii, dar poate să aibă numeroase conexiuni. Această diferență face nepractică implementarea la nivel transport a strategiei de memorare temporară a mesajelor folosită la nivel legătură de date.

În protocoalele pentru legătura de date prezentate în cap. 3, cadrele sunt memorate temporar atât de ruterul care emite cât și de cel care recepționează. În protocolul 6, de exemplu, atât emițătorul cât și receptorul au alocate un număr de *MAXSEQ+1* tampoane pentru fiecare linie, jumătate pentru intrări și jumătate pentru ieșiri. Pentru un calculator gazdă cu, să spunem, 64 de conexiuni și numere de secvență de 4 biți, acest protocol ar necesita 1024 tampoane.

La nivel legătură de date, emițătorul trebuie să memoreze cadrele transmise, pentru că poate fi necesară retransmiterea acestora. Dacă subrețeaua oferă un serviciu datagramă, atunci entitatea de transport emițătoare va trebui să memoreze pachetele trimise din aceleași motive. Dacă receptorul știe că emițătorul stochează toate TPDU-urile până când acestea sunt confirmate, el poate să aloce sau nu tampoane specifice fiecărei conexiuni, după cum i se pare mai bine.

Receptorul poate, de exemplu, să rezerve un singur grup de tampoane pentru toate conexiunile. La sosirea unui TPDU se face o încercare de a obține dinamic un nou tampon. Dacă un tampon este liber, atunci TPDU-ul este acceptat, altfel, este refuzat. Cum emițătorul este gata să retransmită TPDU-urile pierdute de subrețea, faptul că unele TPDU-uri sunt refuzate nu produce nici o daună, deși în acest fel sunt risipite resurse. Emițătorul va retransmite până când va primi confirmarea.

Pe scurt, dacă serviciul rețea nu este sigur, emițătorul va trebui să memoreze toate TPDU-urile trimise, la fel ca la nivel legătură de date. Totuși, folosind un serviciu la nivel rețea sigur sunt posibile unele compromisuri. În particular, dacă emițătorul știe că receptorul are întotdeauna tampoane disponibile, atunci nu trebuie să păstreze copiile TPDU-urilor trimise. Totuși, dacă receptorul nu poate garanta că orice TPDU primit va fi acceptat, emițătorul va trebui să păstreze copii. În ultimul caz, emițătorul nu poate avea încredere în confirmarea primită la nivel rețea, deoarece aceasta confirmă sosirea TPDU-ului la destinație, dar nu și acceptarea lui. Vom reveni asupra acestui punct important mai târziu.

Chiar dacă receptorul va realiza memorarea temporară a mesajelor primite, mai rămâne problema dimensiunii tamponului. Dacă cea mai mare parte a TPDU-urilor au aceeași dimensiune, este naturală organizarea tampoanelor într-o resursă comună care conține tampoane de aceeași dimensiune, cu un TPDU per tampon, ca în fig. 6-15(a). Dacă însă dimensiunea TPDU-urilor variază de la câteva caractere tipărite la un terminal, la mii de caractere pentru un transfer de fișiere, organizarea ca o resursă comună cu tampoane de aceeași dimensiune va pune probleme. Dacă dimensiunea tampoanelor ar fi constantă, egală cu cel mai mare TPDU posibil, atunci va apărea o risipă de spațiu ori de câte ori este primit un TPDU mai scurt. Dacă dimensiunea tampoanelor este aleasă mai mică decât cel mai mare TPDU posibil, atunci pentru memorarea unui TPDU mai lung vor fi necesare mai multe tampoane, iar complexitatea operației va crește.

O altă soluție este utilizarea unor tampoane de dimensiune variabilă, ca în fig. 6-15(b). Avantajul este o mai bună utilizare a memoriei, cu prețul unei gestiuni a tampoanelor mai complicată. O a treia posibilitate este alocarea unui singur tampon circular pentru fiecare conexiune, ca în fig. 6-15(c). Această soluție are de asemenea avantajul unei utilizări eficiente a memoriei, dar numai în situația în care conexiunile sunt relativ încărcate.

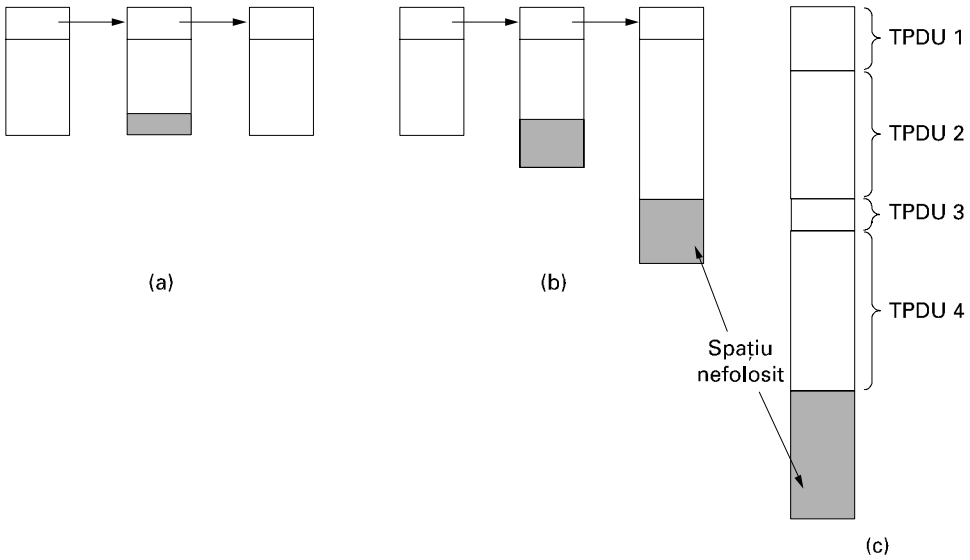


Fig. 6-15. (a) Tampoane de dimensiune fixă înlănțuite. (b) Tampoane de dimensiune variabilă înlănțuite. (c) Un tampon circular pentru fiecare conexiune.

Compromisul optim între memorarea temporară la sursă sau la destinație depinde de tipul traficului prin conexiune. Pentru un trafic în rafală cu o bandă de transfer îngustă, ca traficul produs de

un terminal interactiv, este mai bine ca tampoanele să nu fie prealocate, ci mai curând, alocate dinamic. Întrucât emițătorul nu poate să fie sigur că receptorul va reuși să aloce un tampon la sosirea unui pachet, emițătorul va fi nevoit să rețină copia unui TPDU transmis până când acesta va fi confirmat. Pe de altă parte, pentru un transfer de fișiere sau pentru orice alt trafic care necesită o bandă de transfer largă este mai bine dacă receptorul alocă un set întreg de tampoane, pentru a permite un flux de date la viteză maximă. Cu alte cuvinte, pentru un trafic în rafală cu o bandă de transfer îngustă este mai bine să fie folosite tampoane la emițător, în timp ce pentru un trafic continuu cu o bandă de transfer largă, este mai eficientă folosirea tampoanelor la receptor.

Pe măsură ce conexiunile sunt create și eliberate de trafic, iar șablonul se schimbă, emițătorul și receptorul trebuie să își ajusteze dinamic politica de alocare a tampoanelor. În consecință, protocolul de transport trebuie să permită emițătorului să ceară spațiu pentru tampoane la capătul celălalt al conexiunii. Tampoanele pot fi alocate pentru o anumită conexiune sau pot fi comune pentru toate conexiunile între două calculatoare gazdă. O alternativă este ca receptorul, cunoscând situația tampoanelor sale (dar necunoscând șablonul traficului) să poată spune emițătorului: „Am rezervat X tampoane pentru tine”. Dacă numărul conexiunilor deschise trebuie să crească, poate fi necesar ca spațiul alocat unei singure conexiuni să scadă, deci protocolul trebuie să furnizeze și această facilitate.

O modalitate înțeleaptă de a trata alocarea dinamică a tampoanelor este separarea stocării în tampoane de confirmarea mesajelor, spre deosebire de protocolul cu fereastră glisantă din cap. 3. Alocarea dinamică a tampoanelor înseamnă, de fapt, o fereastră cu dimensiune variabilă. La început, emițătorul trimite cereri pentru un anumit număr de tampoane bazându-se pe o estimare a necesităților. Receptorul îi alocă atâtea tampoane cât își poate permite. De fiecare dată când emițătorul trimite un TPDU, el decrementează numărul de tampoane pe care le are alocate la receptor, oprindu-se când acest număr devine zero. Receptorul trimite înapoi confirmări și situația tampoanelor alocate, împreună cu traficul în sens invers.

Fig. 6-16 este un exemplu pentru modul în care administrarea dinamică a ferestrelor poate fi folosită într-o subrețea cu datagrame, cu numere de secvență pe 4 biți. Să presupunem că informația despre alocarea tampoanelor este împachetată în TPDU-uri distincte și că este separată de traficul în sens invers. La început A dorește opt tampoane, dar nu i se acordă decât patru. După aceea trimite trei TPDU-uri, iar al treilea este pierdut. TPDU-ul 6 confirmă recepția tuturor TPDU-urilor cu numere de secvență mai mici sau egale cu 1, permițând lui A să elibereze acele tampoane și, mai mult, îl informează pe A că B poate să mai recepționeze trei TPDU-uri (adică TPDU-urile cu numere de secvență 2, 3, 4). A știe că a trimis deja numărul 2, deci se gândește că ar putea trimite 3 și 4, ceea ce și încearcă să facă. În acest moment, el este blocat și trebuie să aștepte alocarea unui tampon. Expirarea timpului pentru primirea confirmării determină retransmisia mesajului 2 (linia 9), care poate avea loc, deși emițătorul este blocat, deoarece se vor utiliza tampoane deja alocate. În linia 10, B confirmă primirea tuturor TPDU-urilor până la 4, dar îl ține pe A încă blocat. O astfel de situație ar fi fost imposibilă cu protocolul cu fereastră glisantă prezentat în cap. 3. Următorul TPDU de la B la A alocă încă un tampon și îi permite lui A să continue.

În cazul strategiilor pentru alocarea tampoanelor de tipul celei de mai sus, eventuale probleme pot să apară în rețele neorientate pe conexiune dacă sunt pierdute TPDU-uri de control. Să privim linia 16 din fig. 6-16: B a mai alocat tampoane pentru A, dar TPDU-ul care transmitea această informație a fost pierdut. Deoarece TPDU-urile de control nu sunt numerotate sau retransmise, A va fi blocat. Pentru a preveni această situație, fiecare calculator gazdă trebuie să trimită periodic pe fiecare conexiune TPDU-uri de control ce pot conține confirmări și informații despre starea tampoanelor. În acest fel, A va fi deblocat mai devreme sau mai târziu.

	A	Mesajul	B	Comentarii
1	→	< cere 8 tampoane>	→	A cere 8 tampoane
2	←	<ack=15, buf=4>	←	B îi acordă tampoane numai de la 0 la 3
3	→	<seq = 0, data = m0>	→	A mai are 3 tampoane libere
4	→	<seq = 1, data = m1>	→	A mai are 2 tampoane libere
5	→	<seq = 2, data = m2>	•••	Mesaj pierdut, dar A crede că mai are un singur tampon liber
6	←	<ack = 1, buf = 3>	←	B confirmă 0 și 1 și permite 2-4
7	→	<seq = 3, data = m3>	→	A mai are tampoane
8	→	<seq = 4, data = m4>	→	A nu mai are tampoane libere și trebuie să se oprească
9	→	<seq = 2, data = m2>	→	A retransmite la expirarea intervalului de timp
10	←	<ack = 4, buf = 0>	←	Toate mesajele sunt confirmate, dar A este în continuare blocat
11	←	<ack = 4, buf = 1>	←	A poate să îl trimită acum pe 5
12	←	<ack = 4, buf = 2>	←	B a mai găsit un tampon
13	→	<seq = 5, data = m5>	→	A mai are un tampon liber
14	→	<seq = 6, data = m6>	→	A este blocat din nou
15	←	<ack = 6, buf = 0>	←	A este blocat în continuare
16	•••	<ack = 6, buf = 4>	←	Posibilă interblocare

Fig. 6-16. Alocarea dinamică a tampoanelor. Săgețile indică direcția transmisiei. Punctele de suspensie (...) indică pierderea unui TPDU.

Până acum am presupus tacit că singura limită impusă ratei de transfer a emițătorului este legată de dimensiunea spațiului alocat la receptor pentru tampoane. Deoarece prețul memoriei continuă să scadă vertiginos, ar putea deveni posibilă echiparea unei gazde cu suficient de multă memorie, astfel încât lipsa tampoanelor să pună rar probleme, dacă le va pune vreodată.

Atunci când spațiul de memorie alocat pentru tampoane nu limitează fluxul maxim, va apărea o altă limitare: capacitatea de transport a subrețelei. Dacă două rutere adiacente pot să schimbe cel mult x pachete pe secundă și există k căi distincte între două calculatoare gazdă, atunci este imposibil transferul la o rată mai mare de $k * x$ TPDU-uri pe secundă, oricât de multe tampoane ar fi alocate la cele două capete ale conexiunii. Dacă emițătorul se grăbește prea tare (adică trimite mai mult de $k * x$ TPDU-uri pe secundă), rețeaua se va congestiona, deoarece nu va putea să livreze datele le fel de repede cum le primește.

Este necesar un mecanism bazat pe capacitatea de transport a subrețelei și nu pe capacitatea de memorare în tampoane a receptorului. Evident, mecanismul de control al fluxului trebuie aplicat la emițător pentru a preveni existența prea multor TPDU-uri neconfirmate la acesta. Belsens (1975) a propus folosirea unei scheme cu fereastră glisantă în care emițătorul modifică dinamic dimensiunea ferestrei pentru a o potrivi la capacitatea de transport a rețelei. Dacă rețeaua poate să transporte c TPDU-uri pe secundă și durata unui ciclu (incluzând transmisia, propagarea, timpul petrecut în cozi, prelucrarea la destinație și revenirea confirmării) este r, atunci dimensiunea ferestrei la emițător trebuie să fie $c * r$. Folosind o fereastră cu această dimensiune, emițătorul va putea lucra la capacitate maximă. Orice mică scădere a performanțelor rețelei va genera blocări ale emițătorului.

Pentru a ajusta periodic dimensiunea ferestrei, emițătorul poate urmări cei doi parametri, după care poate calcula dimensiunea dorită a ferestrei. Capacitatea de transport a subrețelei poate fi de-

terminată pur și simplu numărând TPDU-urile confirmate într-o anumită perioadă de timp și raportând la acea perioadă. În timpul măsurătorii, emițătorul trebuie să transmită cât mai repede pentru a fi sigur că factorul care limitează numărul confirmărilor este capacitatea de transport a subrețelei și nu rata mică de emisie. Timpul necesar pentru ca un TPDU transmis să fie confirmat poate fi măsurat cu exactitate și media poate fi calculată continuu. Deoarece capacitatea de transport a rețelei disponibilă pentru orice flux dat variază în timp, dimensiunea ferestrei trebuie ajustată frecvent pentru a urmări schimbările în capacitatea de transport. Așa cum vom vedea mai departe, în Internet se folosește un mecanism similar.

6.2.5 Multiplexarea

Multiplexarea mai multor conversații pe conexiuni, circuite virtuale și legături fizice joacă un rol important în mai multe niveluri ale arhitecturii rețelei. În cazul nivelului transport, multiplexarea poate fi necesară din mai multe motive. De exemplu, dacă doar o singură adresă de rețea este disponibilă pe o gazdă, toate conexiunile transport de pe acea mașină trebuie să o folosească. Când un TDPU sosește este necesar un mod de a spune cărui proces trebuie dat. Această situație numită **multiplexare în sus**, este prezentată în fig. 6-17(a). În această figură, patru conexiuni transport diferite folosesc în comun aceeași conexiune rețea (de exemplu, adresa IP) către calculatorul gazdă de la distanță.

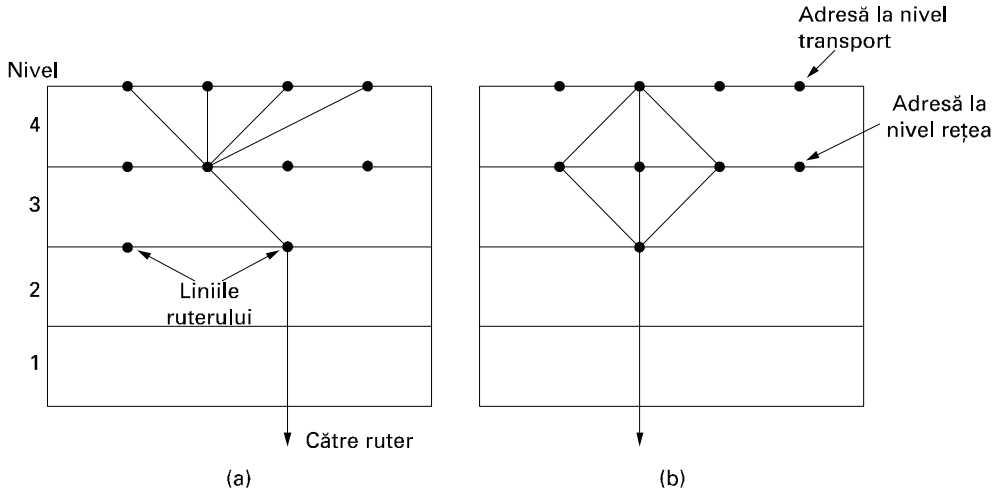


Fig. 6-17. (a) Multiplexare în sus. (b) Multiplexare în jos.

Multiplexarea poate să fie utilă nivelului transport și din alt motiv, legat de deciziile tehnice și nu de politica de prețuri ca până acum. Să presupunem, de exemplu, că o subrețea folosește intern circuite virtuale și impune rată de date maximă p fiecare dintre ele. Dacă un utilizator are nevoie de mai multă lățime de bandă decât poate oferi un circuit virtual, o soluție este ca nivelul transport să deschidă mai multe conexiuni rețea și să distribuie traficul prin acestea (într-un sistem round-robin), la fel ca în fig. 6-17(b). Acest mod de operare se numește **multiplexare în jos.** În cazul a *k* conexiuni rețea deschise lățimea de bandă reală este multiplicată cu un factor *k*. Un exemplu obișnuit de multiplexare în jos apare la utilizatorii care au acasă o linie ISDN. Această linie pune la dispoziție două conexiuni separate de 64 Kbps. Folosirea ambelor pentru apelul unui distribuitor de Internet și împărțirea traficului pe ambele linii, face posibilă atingerea unei lățimi de bandă efectivă de 128 Kbps.

6.2.6 Refacerea după cădere

În cazul în care calculatoarele gazdă sau ruterele se întâmplă să cadă, recuperarea după o astfel de cădere devine o problemă. Dacă entitatea de transport este în întregime conținută de calculatorul gazdă, atunci revenirea după o cădere a rețelei sau a unui ruter este simplă. Dacă nivelul rețea furnizează un serviciu datagramă, atunci entitatea de transport știe să rezolve problema TPDU-urilor pierdute. Dacă nivelul rețea furnizează un serviciu orientat pe conexiune, atunci pierderea unui circuit virtual este tratată stabilind un circuit virtual nou și apoi întrebând entitatea de transport aflată la distanță care TPDU-uri a primit deja și ce TPDU-uri nu. Acestea din urmă pot fi retransmise.

Căderea unui calculator gazdă pune o problemă mult mai supărătoare. În particular, clienții pot dori să continue să lucreze imediat după ce serverul cade și repornește. Pentru a ilustra această dificultate, să presupunem că o gazdă (clientul) trimite un fișier lung unei alte gazde (serverul) folosind un protocol simplu de tip pas-cu-pas (stop-and-wait). Nivelul transport de pe server nu face decât să paseze utilizatorului TPDU-urile primite, unul câte unul. Dacă în timpul transmisiei serverul cade, la revenirea acestuia tabelele sunt reinițializate și serverul nu va mai ști precis unde a rămas.

Într-o încercare de a reveni în starea sa inițială, serverul ar putea să trimită cereri tuturor celorlalte gazde anunțând că tocmai s-a refăcut după o cădere și cerând clienților să-l informeze despre situația tuturor conexiunilor deschise. Fiecare client poate fi în una din următoarele stări: un TPDU neconfirmat (starea S1) sau nici un TPDU neconfirmat (starea S0). Bazându-se numai pe această informație, clientul trebuie să decidă dacă să retransmită sau nu cel mai recent TPDU.

La prima vedere pare evident: atunci când află de căderea și revenirea serverului, clientul trebuie să retransmită doar dacă are TPDU-uri neconfirmate (adică este în starea S1). Totuși, o privire mai atentă descoperă dificultățile care apar în această abordare naivă. Să considerăm, ca exemplu, situația în care entitatea de transport de pe server trimite mai întâi confirmarea și apoi, după ce confirmarea a fost trimisă, pasează datele procesului aplicație. Trimiterea confirmării și transferul datelor procesului aplicație sunt două evenimente distincte care nu pot avea loc simultan. Dacă o cădere a serverului are loc după trimiterea confirmării, dar înainte de transferul datelor, clientul va primi confirmarea și se va afla în starea S0, atunci când anunțul despre cădere ajunge la el. În acest caz, clientul nu va mai retransmite (ceea ce este incorect), crezând că TPDU-ul respectiv a fost recepționat. Această decizie a clientului va conduce la lipsa unui TPDU.

În acest moment s-ar putea spune: „Nimic mai simplu! Tot ceea ce trebuie făcut este să reprogramăm entitatea de transport astfel, încât să transfere datele mai întâi și să trimită confirmarea după aceea!”. Să vedem: dacă transferul datelor a avut loc, dar serverul cade înainte să trimită confirmarea, clientul va fi în starea S1, va retransmite și astfel vom avea un TPDU duplicat în fluxul de date către procesul aplicație.

Oricum ar fi proiectați clientul și serverul, întotdeauna vor exista situații în care revenirea după o cădere nu se va face corect. Serverul poate fi programat în două feluri: să facă mai întâi confirmarea sau să transfere mai întâi datele. Clientul poate fi programat în patru feluri: să retransmită întotdeauna ultimul TPDU, să nu retransmită niciodată, să retransmită numai în starea S0, să retransmită numai în starea S1. Rezultă astfel opt combinații, dar, așa cum vom vedea, pentru fiecare combinație există o anumită succesiune de evenimente pentru care protocolul nu funcționează corect.

La server sunt posibile trei evenimente: trimiterea unei confirmări (A), transferul datelor la procesul aplicație (T) și căderea (C). Aceste trei evenimente pot să fie ordonate în 6 feluri: AC(T), ATC, C(AT), C(TA), TAC și TC(A). Parantezele sunt folosite pentru a indica faptul că nici A, nici

T nu pot urma după C (odată ce serverul a căzut, e bun căzut). Fig. 6-18 prezintă toate cele opt combinații ale strategiilor clientului și serverului și prezintă secvențele valide pentru fiecare din ele. Se observă că pentru orice strategie există o secvență de evenimente pentru care protocolul nu funcționează corect. De exemplu, dacă clientul retransmite întotdeauna, secvența ATC va genera un duplicat care nu poate fi detectat, în timp ce pentru celelalte două secvențe totul este în regulă.

	Strategia folosită de receptor					
	Mai întâi confirmă, apoi transferă			Mai întâi transferă, apoi confirmă		
Strategia folosită de emițător	AC(T)	ATC	C(AT)	C(TA)	T AC	TC(A)
Retransmite întotdeauna	OK	DUP	OK	OK	DUP	DUP
Nu retransmite niciodată	LOST	OK	LOST	LOST	OK	OK
Retransmite în S0	OK	DUP	LOST	LOST	DUP	OK
Retransmite în S1	LOST	OK	OK	OK	OK	DUP

OK = Protocolul funcționează corect
DUP = Protocolul generează un mesaj duplicat
LOST = Protocol pierde un mesaj

Fig. 6-18. Combinațiile diferitelor strategii posibile pentru server și client.

Încercarea de a reproiecta mai minuțios protocolul nu ajută. Chiar dacă clientul și serverul schimbă mai multe TPDU-uri înainte ca serverul să transfere datele, astfel încât clientul știe exact ceea ce se petrece pe server, nu poate afla dacă serverul a căzut imediat înainte sau imediat după transferul datelor. Concluzia se impune de la sine: dată fiind condiția de bază ca două evenimente să nu fie simultane, revenirea după o cădere nu poate fi făcută transparent pentru nivelurile superioare.

În termeni mai generali, acest rezultat poate fi reformulat astfel: restartarea după o cădere a nivelului N nu poate fi făcută decât de către nivelul N+1, și aceasta doar dacă nivelul superior reține suficientă informație de stare. Așa cum am arătat mai sus, nivelul transport poate să revină după erorile nivelului rețea numai dacă fiecare capăt al conexiunii ține minte unde a rămas.

Am ajuns astfel la problema definirii precise a ceea ce înseamnă o confirmare capăt-la-capăt. În principiu, protocolul de transport este unul capăt-la-capăt și nu înlănțuit ca la nivelurile de mai jos. Să considerăm cazul unui utilizator care generează cereri de tranzacții pentru o bază de date de la distanță. Să presupunem că entitatea de transport aflată la distanță este programată să trimită mai întâi TPDU-ul nivelului superior și apoi să confirme. Chiar și în acest caz, primirea unei confirmări de către mașina utilizator nu înseamnă neapărat că mașina gazdă de la distanță a avut timp să actualizeze baza de date. De fapt, o adevărată confirmare capăt-la-capăt, a cărei primire arată că s-au efectuat toate prelucrările de către mașina de la distanță, este probabil imposibil de obținut. Acest subiect este discutat în detaliu de Saltser ș.a. (1984).

6.3 UN PROTOCOL SIMPLU DE TRANSPORT

Pentru a concretiza ideile discutate, în această secțiune vom studia în detaliu un exemplu de nivel transport. Vom utiliza setul abstract de primitive orientate pe conexiune prezentat în fig. 6-2. Alegerea acestor primitive orientate pe conexiune face exemplul ales similar cu (dar mai simplu decât) popularul protocol TCP.

6.3.1 Primitivele serviciului ales ca exemplu

Prima problemă constă în definirea exactă a primitivelor de transport. Pentru CONNECT este ușor: vom avea o rutină de bibliotecă *connect* care poate fi apelată cu parametri potriviți pentru stabilirea unei conexiuni. Parametrii sunt TSAP-ul local și cel aflat la distanță. În timpul apelului, apelantul este blocat în timp ce entitatea de transport încearcă să stabilească conexiunea. Dacă conexiunea reușește, apelantul este deblocat și poate să înceapă să transmită date.

Atunci când un proces dorește să accepte conexiuni, el face un apel *listen* specificând un anumit TSAP pe care îl ascultă. După aceasta, procesul este blocat până când un proces aflat la distanță încearcă să stabilească o conexiune cu TSAP-ul la care așteaptă.

De observat că acest model este asimetric. O parte este pasivă, executând *listen* și așteptând ca ceva să se întâmple. Cealaltă parte este activă și inițiază conexiunea. O întrebare interesantă este: ce trebuie făcut dacă partea activă începe prima? O posibilitate este ca încercarea de conectare să nu reușească dacă nu există nici un proces care să asculte la TSAP-ul aflat la distanță. O altă posibilitate este ca inițiatorul să se blocheze (eventual pentru totdeauna) până când apare un proces care să asculte la TSAP-ul aflat la distanță.

Un compromis folosit în exemplul nostru este păstrarea cererii de conexiune la receptor pentru un anumit interval de timp. Dacă un proces de pe acest calculator gazdă apelează *listen* înainte ca timpul să expire, atunci conexiunea este stabilită, altfel conexiunea este refuzată și apelantul este deblocat întorcând un cod de eroare.

Pentru a elibera o conexiune vom folosi un apel *disconnect.* Atunci când ambele părți s-au deconectat, conexiunea este eliberată. Cu alte cuvinte, folosim un model de deconectare simetric.

Transmisia de date are exact aceeași problemă ca și stabilirea conexiunii: emițătorul este activ, dar receptorul este pasiv. Vom folosi aceeași soluție pentru transmisia de date ca și pentru stabilirea conexiunii, adică un apel activ *send* care trimite datele și un apel pasiv *receive* care blochează până când sosește un TPDU.

Definiția concretă a serviciului constă astfel din cinci primitive: CONNECT, LISTEN, DISCONNECT, SEND și RECEIVE. Fiecare primitivă corespunde unei funcții de bibliotecă care execută acea primitivă. Parametrii pentru primitive și funcțiile de bibliotecă sunt:

```
connum = LISTEN(local)
connum = CONNECT (local, remote)
status = SEND(connum, buffer, bytes)
status = RECEIVE(connum, buffer, bytes)
status = DISCONNECT(connum)
```

Primitiva LISTEN anunță disponibilitatea serverului de a accepta cereri de conexiune la TSAP-ul indicat. Utilizatorul primitivei este blocat până când se face o încercare de conectare la TSAP-ul specificat. Blocarea poate să fie definitivă.

Primitiva CONNECT are doi parametri, un TSAP local (adică o adresă la nivel transport) și un TSAP aflat la distanță și încearcă să stabilească o conexiune între acestea două. Dacă reușește, ea întoarce *connum,* un număr nenegativ utilizat pentru a identifica conexiunea. Dacă nu reușește, motivul este pus în *connum* ca un număr negativ. În modelul nostru (destul de simplu), fiecare TSAP poate să participe la cel mult o singură conexiune de transport, deci un motiv pentru refuzul unei cereri de conectare poate fi că adresa la nivel transport este deja folosită. Alte câteva motive pot fi: gazda de la distanță căzută, adresă locală incorectă sau adresă de la distanță incorectă.

Primitiva SEND trimite conținutul unui tampon ca un mesaj pe conexiunea indicată, eventual divizându-l în mai multe unități, dacă este nevoie. Erorile posibile, întoarse în *status*, pot fi: nu există conexiune, adresă de tampon invalidă sau număr de biți negativ.

Primitiva RECEIVE indică faptul că apelantul așteaptă să recepționeze date. Dimensiunea mesajului este plasată în câmpul *bytes.* Dacă procesul aflat la distanță a eliberat conexiunea sau dacă adresa pentru tampon este incorectă (de exemplu, în afara spațiului de adrese al programului utilizator), funcția va întoarce un cod de eroare indicând natura problemei.

Primitiva DISCONNECT pune capăt unei conexiunii transport indicate prin parametrul *connum*. Erori posibile sunt: *connum* aparține de fapt altui proces sau *connum* nu este un identificator valid de conexiune. Codul de eroare sau 0 pentru succes sunt returnate în *status.*

6.3.2 Entitatea de transport aleasă ca exemplu

Înainte de a studia programul aferent entității de transport, trebuie spus că acesta este un exemplu similar celor din cap. 3: este prezentat mai mult în scop pedagogic decât pentru a fi utilizat. Mai multe detalii tehnice (precum detectarea extensivă a erorilor) care ar fi necesare unui produs real au fost lăsate deoparte pentru simplitate.

Nivelul transport utilizează primitivele serviciului rețea pentru a trimite și recepționa TPDU-uri. Trebuie să alegem primitivele serviciului rețea pe care le vom utiliza pentru acest exemplu. O posibilitate ar fi fost: un serviciu datagramă nesigur. Pentru a păstra simplitatea exemplului, nu am făcut această alegere. Folosind un serviciu datagramă nesigur, codul pentru entitatea de transport ar fi devenit foarte mare și complicat, în cea mai mare parte legat de pachete pierdute sau întârziate. Și în plus, cea mai mare parte a acestor idei au fost deja discutate în cap. 3.

Am ales în schimb un serviciu rețea sigur, orientat pe conexiune. În acest fel ne putem concentra asupra problemelor puse de nivelul transport care nu apar în nivelurile inferioare. Acestea includ, între altele, stabilirea conexiunii, eliberarea conexiunii și gestiunea tampoanelor. Un serviciu de transport simplu, construit peste un nivel rețea ATM, ar putea să semene cu acesta.

În general, entitatea de transport poate să fie parte a sistemului de operare al calculatorului gazdă sau poate să fie un pachet de funcții de bibliotecă în spațiul de adrese utilizator. Pentru simplitate, exemplul nostru a fost programat ca și cum entitatea de transport ar fi un pachet de funcții de bibliotecă, dar schimbările necesare pentru a o face parte a sistemului de operare sunt minime (fiind în principal legate de modul cum sunt adresate tampoanele).

Totuși merită remarcat faptul că, în acest exemplu, „entitatea de transport” nu este o entitate separată, ci este o parte a procesului utilizator. Mai precis, atunci când utilizatorul folosește o primitivă blocantă, de exemplu LISTEN, se blochează toată entitatea de transport. În timp ce această arhitectură este potrivită pentru un calculator gazdă cu un singur proces utilizator, pentru un calculator gazdă cu mai mulți utilizatori este normal ca entitatea de transport să fie un proces separat, distinct de toate celelalte procese.

Interfața cu nivelul rețea se face prin intermediul procedurilor *to_net* și *from_net.* Fiecare are șase parametri. Primul este identificatorul conexiunii, care este în corespondență bijectivă cu circuitul virtual la nivel rețea. Apoi urmează biții *Q* și *M* care, atunci când au valoarea 1, indică mesaje de control și, respectiv, faptul că mesajul continuă și în următorul pachet. După acestea urmează tipul pachetului, ales dintr-un set de șase tipuri de pachete prezentate în fig. 6-19. La sfârșit se găsește un indicator către zona de date și un întreg care indică numărul de octeți de date.

Tip pachet	Explicații
CALL_ REQUEST	Trimis pentru a stabili conexiunea
CALL_ACCEPTED	Răspuns la CALL_ REQUEST
CLEAR_REQUEST	Trimis pentru a elibera conexiunea
CLEAR_CONFIRMATION	Răspuns la CLEAR_REQUEST
DATA	Pentru transport de date
CREDIT	Pachet de control pentru gestionarea ferestrei

Fig. 6-19. Tipurile de pachete folosite la nivel rețea.

La apelurile *to_net,* entitatea de transport completează toți parametrii pentru ca nivelul rețea să-i poată citi; la apelurile *from_net* nivelul rețea dezasamblează un pachet sosit pentru entitatea de transport. Transferul informației sub forma unor parametri ai unei proceduri și nu a pachetului de trimis sau de recepționat protejează nivelul transport de toate detaliile protocolului nivelului rețea. Dacă entitatea de transport încearcă să trimită un pachet atunci când fereastra glisantă corespunzătoare a circuitului virtual este plină, ea va fi blocată într-un apel *to_net* până când va fi spațiu în fereastră. Mecanismul este transparent pentru entitatea de transport și este controlat de nivelul rețea prin comenzi ca *enable_transport_layer* și *disable_transport_layer,* analoage celor descrise în protocoalele din cap. 3. Gestionarea ferestrei de pachete este de asemenea făcută de către nivelul rețea.

Pe lângă acest mecanism transparent de suspendare mai există două proceduri (care nu sunt prezentate aici), *sleep* și *wakeup,* apelate de entitatea de transport. Procedura *sleep* este apelată atunci când entitatea de transport este blocată logic în așteptarea unui eveniment extern, în general sosirea unui pachet. După ce a fost apelat *sleep*, entitatea de transport (și procesul utilizator, bineînțeles) sunt blocate.

Codul entității de transport este prezentat în fig. 6-20. Orice conexiune este într-una din următoarele șapte stări:

1. *IDLE* - conexiunea nu a fost încă stabilită
2. *WAITING* - a fost executat CONNECT și trimis: CALL_REQUEST
3. *QUEUED* - a sosit un CALL_REQUEST, nu a fost executat încă nici un apel LISTEN
4. *ESTABLISHED* - conexiunea a fost stabilită
5. *SENDING* - utilizatorul așteaptă permisiunea de a trimite un pachet
6. *RECEIVING* - a fost apelat RECEIVE
7. *DISCONNECTING* - un apel DISCONNECT a fost făcut local

Pot să apară tranziții între stări ori de câte ori are loc unul din următoarele evenimente: este executată o primitivă, sosește un pachet sau expiră un interval de timp stabilit.

Procedurile prezentate în fig. 6-20 sunt de două tipuri. Majoritatea sunt apelate direct de către programele utilizator, totuși *packet_arrival* și *clock* sunt diferite. Execuția lor este declanșată de evenimente externe: sosirea unui pachet și, respectiv, avansul ceasului. Ele sunt de fapt rutine de întrerupere. Vom presupune că acestea nu sunt niciodată apelate atât timp cât rulează o altă procedură a entității de transport. Acestea pot fi executate numai atunci când procesul utilizator este în așteptare sau atât timp cât el rulează în afara entității de transport. Această proprietate este crucială pentru funcționarea corectă a entității de transport.

```
#define MAX_CONN 32                    /* numărul maxim de conexiuni deschise simultan */
#define MAX_MSG_SIZE 8192              /* dimensiunea maximă a unui mesaj în octeți */
#define MAX_PKT_SIZE 512               /* dimensiunea maximă a unui pachet în octeți */
#define TIMEOUT 20
#define CRED 1
#define OK 0

#define ERR_FULL -1
#define ERR_REJECT -2
#define ERR_CLOSED -3
#define LOW_ERR -3

typedef int transport_address;
typedef enum {CALL_REQ,CALL_ACC, CLEAR_REQ, CLEAR_CONF, DATA_PKT,CREDIT}
pkt_type;
typedef enum {IDLE, WAITING, QUEUED, ESTABLISHED, SENDING, RECEIVING, DISCONN}
cstate;

/* Variabile globale */
transport_address listen_address;              /* adresa locală care este ascultată */
int listen_conn;                               /* identificatorul de conexiune pentru listen */
unsigned char data[MAX_PKT_SIZE]               /* zona pentru pachetele de date */

struct conn {
    transport_address local_address, remote_address;
    cstate state;                                          /* starea conexiunii */
    unsigned char *user_buf_addr;                          /* pointer la tamponul de recepție */
    int byte_count;                                        /* contor de emisie/recepție*/
    int clr_req_received;          /* setat atunci când este primit un pachet CLEAR_REQ*/
    int timer;                                 /* folosit pentru a evita așteptările infinite */
    int credits;                               /* numărul de mesaje care poate fi trimis */
}; conn[MAX_CONN + 1]                                      /* poziția 0 nu e folosită */

/* prototipuri */
void sleep(void);
void wakeup(void);
void to_net(int cid, int q, int *m, pkt_type pt, unsigned char *p, int bytes);
void from_net(int *cid, int *q, int *m, pkt_type *pt,unsigned char *p, int *bytes);

int listen (transport_address t)
{
/* Utilizatorul vrea stabilească o conexiune. Trebuie văzut daca CALL_REQ a sosit deja */
    int i, found=0;

    for (i=1; i<= MAX_CONN; i++)               /* caută în tabela de conexiuni CALL_REQ*/
        if (conn[i].state == QUEUED && conn[i].locall_address == t) {
            found = i;
            break;
        }
        if (found == 0) {          /* nu a găsit nici un CALL_REQ. Așteaptă până când sosește
                                      unul sau până când expira intervalul de timp de așteptare */
            listen_address = t; sleep(); i = listen_conn;
        }
    conn[i].state = ESTABLISHED;                           /* conexiunea este stabilită */
```

```
    conn[i].timer = 0                                   /* ceasul nu este folosit */
    listen_conn = 0;                                    /* 0 este presupus a fi o adresă invalidă */
    to_net(i, 0, 0, CALL_ACC, 0);          /* spune niv. rețea să accepte cererea de conexiune */
    return(i);                                          /* întoarce identificatorul conexiunii */
}

int connect (transport_address l, transport_address r)
{                                  /* Utilizatorul dorește să stabilească o conexiune cu un proces aflat
                                                                    la distanță; se trimite un pachet CALL_REQ */
    int i;
    struct conn *cptr;

    data[0] = r;                                    /* pachetul CALL_REQ are nevoie de acestea */
    data[1] = l;
    i = MAX_CONN;                                                   /* caută în tabelă înapoi */
    while (conn[i].state != IDLE && i>1) i = i-1;
    if (conn[i].state == IDLE) {                                    /* Face o intrare în tabelă */
        cptr = &conn[i];
        cptr->local_address = l;;
        cptr->remote_address = r;
        cptr->state = WAITING;
        cptr->clr_req_received = 0;
        cptr->credits = 0;
        cptr->timer = 0;
        to_net(i, 0, 0, CALL_REQ, data, 2);
        sleep();                                    /* așteaptă CALL_ACC sau CLEAR_REQ */
        if (cptr->state == ESTABLISHED) return (i);
        if (cptr->clr_req_received ) {                  /* cererea de conexiune este refuzată */
            cptr->state = IDLE                                      /* înapoi în starea IDLE */
            to_net(i, 0, 0, CLEAR_CONF, data, 0);
            return(ERR_REJECT);
        }
    }
    else return (ERR_FULL)          /* conexiunea este refuzată: insuficient spațiu în tabele */
}

int send (int cid, unsigned char bufptr[], int bytes)
{                                                       /* Utilizatorul dorește să trimită un mesaj */
    int i, count, m;
    struct conn *cptr = &conn[cid];

    /* Intră în starea SENDING */
    cptr->state = SENDING;
    cptr->byte_count = 0;
    if (cptr->clr_req_received == 0 && cptr->credits == 0) sleep();
    if (cptr->clr_req_received == 0) {
                                    /* Există credite; împarte mesajul în pachete dacă este cazul */
        do {
            if (bytes - cptr->byte_count > MAX_PKT_SIZE) {
                /* mesaj format din mai multe pachete */
                count = MAX_PKT_SIZE;
                m = 1;
                /* mai urmează pachete */
```

```
            } else { /* un mesaj format dintr-un pachet */
                count = bytes — cptr->byte->count;
                m=0;                                    /* ultimul pachet al acestui mesaj */
            }
            for (i=0; i<count; i++) data[i]= bufptr[cptr->byte_count+1];
            to_net(cid, 0, m,DATA_PKT, data, count);          /* trimite un pachet */
            cptr->byte_count=cptr->byte_count + count;     /* incrementează nr. octeților trimiși */
        } while (cptr->byte_count < bytes);  /* ciclează până când întregul mesaj este trimis */
        cptr->credits--;                                  /* fiecare mesaj folosește un credit */
        cptr->state = ESTABLISHED;
        return(OK);
    } else { cptr->state = ESTABLISHED;
        return(ERR_CLOSED);     /* întoarce insucces: celălalt capăt vrea să se deconecteze */
    }
}

int receive (int cid, unsigned char bufptr[], int *bytes)
{
    /* Utilizatorul este gata să primească un mesaj */
    struct conn *cptr = &conn[cid];

    if (cptr->clr_req_received == 0) {   /* Conexiunea încă stabilită; încearcă să recepționeze */
        cptr->state = RECEIVING;
        cptr->user_buf_addr = bufptr;
        cptr->byte_count = 0;
        data[0] = CRED;
        data[1] = 1;
        to_net(cid, 1, 0, CREDIT, data, 2);                    /* trimite CREDIT */
        sleep();                                  /* se blochează în așteptarea datelor */
        *bytes = cptr->byte_count;
    }
    cptr->state = ESTABLISHED;
    return(cptr->clr_req_received ? ERR_CLOSED : OK);
}

int disconnect (int cid)
{   /* Utilizatorul vrea să se deconecteze */
    struct conn *cptr = &conn[cid];

    if (cptr -> clr_req_received) {                /* cealaltă parte a inițiat deconectarea */
        cptr->state = IDLE;                                 /* conexiunea este eliberată */
        to_net(cid, 0, 0, CLEAR_CONF, data, 0);
    } else {                                                /* se inițiază terminarea */
        cptr->state = DISCONNECT;                  /* conexiunea nu este eliberată până când
                                                              cealaltă parte nu-și dă acordul */
        to_net(cid, 0, 0, CLEAR_REQ, data, 0);
    }
    return (OK);
}

void packet_arrival (void)
{   /* A sosit un pachet; urmează prelucrarea lui */
    int cid;                                    /* conexiunea pe care a sosit pachetul */
```

```
    int count, i q, m;
    pkt_type ptype;                                  /* CALL_REQ, CALL_ACC, CLEAR_REQ,
                                                       CLEAR_CONF, DATA_PKT, CREDIT */
    unsigned char data [MAX_MKT_SIZE];               /* date din pachetul care sosește */
     struct conn *cptr;

    from_net(&cid, &q, &ptype, data, &count);                          /* preia pachetul */
    cptr = &conn[cid];
    switch (ptype) {
        case CALL_REQ:      /* utilizatorul de la distanță vrea să stabilească o conexiune */
            cptr->local_address = data[0];
            cptr->remote_address = data[1];
            if (cptr->local_address == listen_address) {
                listen_conn = cid;
                cptr->state = ESTABLISHED;
                wakeup();
            } else {
                cptr->state = QUEUED;
                cptr->timer = TIMEOUT;
            }
            cptr->clr_req_received = 0;
            cptr->credits = 0;
            break;

        case CALL_ACC:          /* utilizatorul de la distanță a acceptat CALL_REQ trimis */
            cptr->state = ESTABLISHED;
            wakeup();
            break;

        case CLEAR_REQ:
                  /* utilizatorul de la distanță vrea să se deconecteze sau să refuze un apel */
            cptr->clr_req_received = 1;
            if (cptr->state == DISCONN) cptr->state = IDLE;
            if (cptr->state == WAITING || cptr->state == RECEIVING
                        || cptr->state == SENDING) wakeup();
            break;

        case CLEAR_CONF:            /* utilizatorul de la distanță acceptă deconectarea */
            cptr->state = IDLE;
            break;

        case CREDIT:                          /* utilizatorul de la distanță așteaptă date */
            cptr->credits +=data[1];
             if (cptr->state == SENDING) wakeup();
             break;

        case DATA_PKT:                                                  /* au fost trimise date */
            for (i=0; i<count; i++)
                cptr->user_buff_addr[cptr->byte_count + i] = data[i];
            cptr->byte_count += count;
            if (m == 0) wakeup();
    }
}
```

```
void clock( void)
{                   /* la fiecare interval de timp al ceasului se verifică eventualele
                       depășiri ale timpilor de așteptare pentru cererile aflate în starea QUEUED */
     int i;
     struct conn *cptr;

     for (i=1; i<=MAX_CONN, i++) {
          cptr = &conn[i];
          if (cptr->timer > 0) {                                   /* ceasul funcționa */
               cptr->timer --;
               if (cptr->timer == 0) {                             /* timpul a expirat */
                    cptr->state = IDLE;
                    to_net(i, 0, 0, CLEAR_REQ, data, 0);
               }
          }
     }
}
```

Fig. 6-20. Entitatea de transport aleasă ca exemplu.

Existența bitului *Q* în antetul pachetului ne permite să evităm timpul suplimentar introdus de antetul protocolului de transport. Mesajele de date obișnuite sunt trimise ca pachete de date cu $Q=0$. Mesajele legate de protocolul de transport, dintre care există numai unul (CREDIT) în exemplul nostru, sunt trimise ca pachete de date cu Q=1. Aceste mesaje de control sunt detectate și prelucrate de entitatea de transport receptoare.

Structura de date de bază folosită de entitatea de transport este vectorul *conn,* care are câte o înregistrare pentru fiecare conexiune posibilă. În înregistrare sunt menținute starea conexiunii, incluzând adresa de nivel transport la fiecare capăt, numărul de mesaje trimise și recepționate pe conexiune, starea curentă, un indicator (pointer) la tamponul utilizator, numărul de octeți ai mesajului trimis sau recepționat, un bit indicând dacă utilizatorul aflat la distanță a apelat DISCONNECT, un ceas, și un contor folosit pentru a permite transmiterea mesajelor. Nu toate aceste câmpuri sunt folosite în exemplul nostru simplu, dar o entitate de transport completă ar avea nevoie de toate, poate chiar de mai multe. Fiecare element din *conn* este inițializat cu starea *IDLE* .

Atunci când un utilizator apelează CONNECT, nivelului rețea i se va cere să trimită un pachet CALL_REQUEST către mașina de la distanță și utilizatorul este blocat. Atunci când pachetul CALL_REQUEST ajunge de partea cealaltă, entitatea de transport este întreruptă pentru a executa *packet_arrival,* care verifică dacă utilizatorul local ascultă la adresa specificată. Dacă da, atunci este trimis înapoi un pachet CALL_ACCEPTED și utilizatorul de la distanță este trezit; dacă nu, atunci cererea CALL_REQUEST este pusă într-o coadă pentru un interval de timp TIMEOUT. Dacă în acest interval este făcut un apel LISTEN, atunci conexiunea este stabilită; dacă nu, conexiunea este refuzată la sfârșitul intervalului de timp cu un pachet CLEAR_REQUEST ca să nu fie blocată pe timp nedefinit.

Deși am eliminat antetul pentru protocolul de transport, tot mai trebuie găsită o modalitate pentru a determina cărei conexiuni transport îi aparține un anumit pachet, deoarece pot exista simultan mai multe conexiuni. Cea mai simplă soluție este folosirea numărului de circuit virtual de la nivel rețea și ca număr de conexiune transport. În plus, numărul de circuit virtual poate fi folosit și ca index în vectorul *conn.* Atunci când un pachet sosește pe circuitul virtual *k* la nivel rețea, el îi va aparține conexiunii *k* a cărei stare este păstrată în *conn[k].* La inițierea unei conexiuni, numărul de cone-

xiune este ales de entitatea de transport care a inițiat-o. Pentru cererile de conexiune, nivelul rețea alege ca număr de conexiune orice număr de circuit virtual care nu a fost încă folosit.

Pentru a evita gestionarea tampoanelor în interiorul entității de transport, este folosit un mecanism de gestiune a fluxului diferit de fereastra glisantă tradițională. Când un utilizator apelează RECEIVE, este trimis un **mesaj de credit** entității de transport de pe mașina care va transmite și este înregistrat în vectorul *conn.* Atunci când este apelat SEND, entitatea de transport verifică dacă a sosit vreun credit pe conexiunea respectivă. Dacă da, mesajul este trimis (chiar și în mai multe pachete, dacă este cazul) și credit este decrementat; dacă nu, atunci entitatea de transport se blochează până la sosirea unui credit. Mecanismul garantează că nici un mesaj nu este trimis decât dacă cealaltă parte a apelat deja RECEIVE. Ca rezultat, ori de câte ori sosește un mesaj, există cu siguranță un tampon disponibil pentru el. Această schemă poate fi ușor generalizată pentru a permite receptorilor să ofere tampoane multiple și să ceară mai multe mesaje.

Trebuie reținută simplitatea codului din fig. 6-20. O entitate de transport reală ar verifica în mod normal validitatea tuturor parametrilor furnizați de utilizator, ar putea să revină după căderea rețelei, ar putea rezolva coliziunile la apel și ar susține un serviciu transport mult mai general, care ar include facilități cum sunt întreruperile, datagramele și versiunile nonblocante ale primitivelor SEND și RECEIVE.

6.3.3 Exemplul văzut ca un automat finit

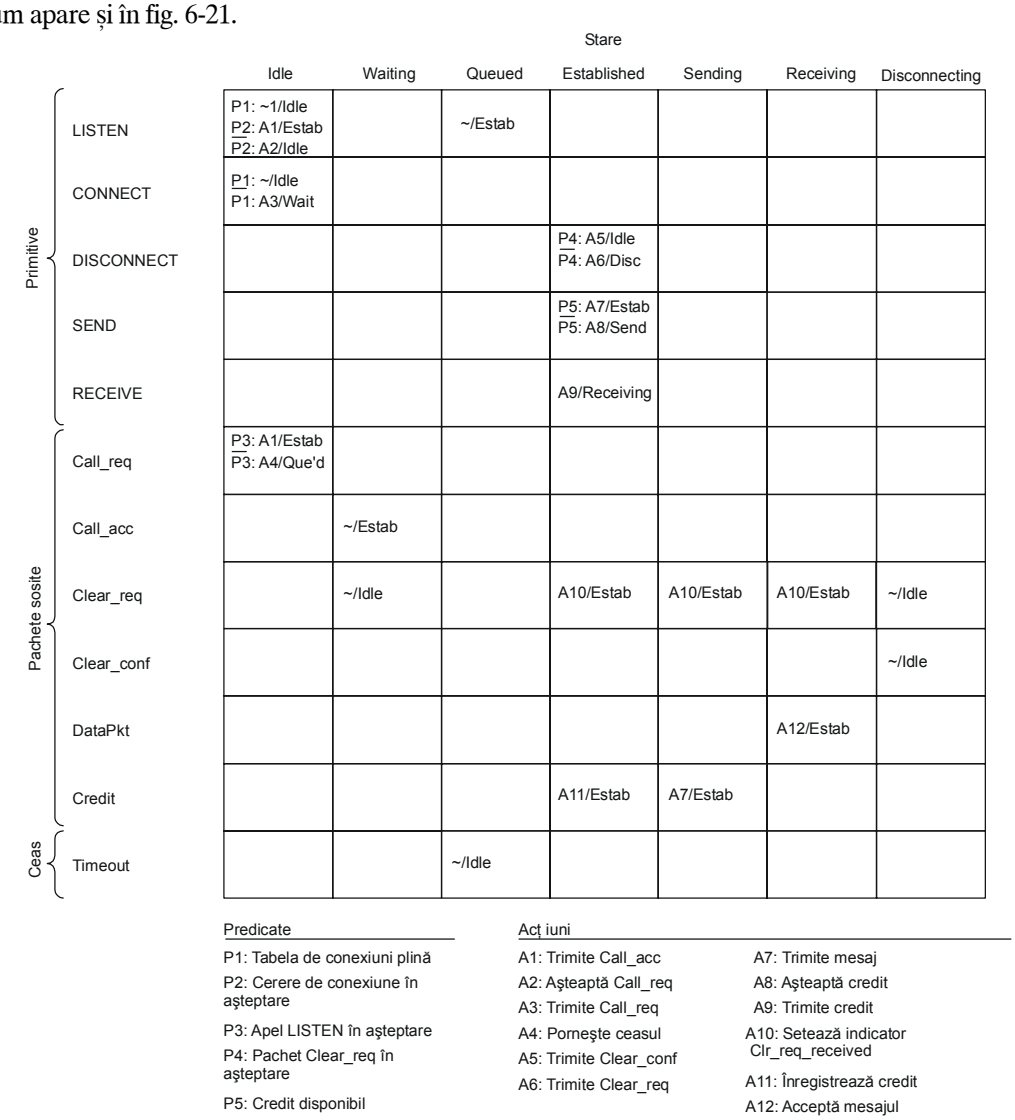
Scrierea unei entități de transport este o muncă dificilă și riguroasă, în special pentru protocoalele reale. Pentru a reduce probabilitatea de apariție a unei erori, adeseori este util să reprezentăm protocolul ca un automat finit.

Am văzut deja că protocolul nostru folosit ca exemplu are șapte stări pentru fiecare conexiune. Este de asemenea posibil să izolăm cele douăsprezece evenimente care pot să apară pentru a schimba starea unei conexiuni. Cinci dintre aceste evenimente sunt cele cinci primitive ale serviciului de transport. Alte șase sunt reprezentate de sosirile celor șase tipuri distincte de pachete. Ultimul este expirarea intervalului de timp stabilit. Fig. 6-21 arată acțiunile principale ale protocolului sub forma unei matrice. Pe coloane sunt prezentate stările, iar pe linii cele 12 evenimente.

Fiecare intrare în matrice (adică în automatul finit) din fig. 6-21 are până la trei câmpuri: un predicat, o acțiune și o nouă stare. Predicatul indică condițiile în care acțiunea este executată. De exemplu, pentru intrarea din colțul stânga-sus, dacă este executat un LISTEN și nu mai există spațiu în tabele (predicatul P1), atunci LISTEN eșuează și starea rămâne aceeași. Pe de altă parte, dacă un pachet CALL_REQUEST a sosit deja la adresa de nivel transport la care se face LISTEN (predicatul P2), atunci conexiunea este stabilită imediat. O altă posibilitate este ca P2 să fie fals, adică nici un CALL_REQUEST nu a fost primit, caz în care conexiunea rămâne în starea *IDLE* în așteptarea unui pachet CALL_REQUEST.

Merită să subliniem că alegerea stărilor folosite în matrice nu este în întregime determinată de protocolul însuși. În acest exemplu, nu există nici o stare *LISTENING*, care ar fi putut urma în mod normal apelului LISTEN. Nu a fost introdusă o stare *LISTENING* deoarece o stare este asociată cu o intrare în tabela de conexiuni și la un apel LISTEN nu se creează nici o intrare în tabelă. De ce? Pentru că am decis să folosim identificatorii circuitelor virtuale de la nivel rețea ca identificatori pentru conexiune și, pentru un apel LISTEN, numărul circuitului virtual este în cele din urmă ales de nivelul rețea atunci când sosește un CALL_REQUEST.

Acțiunile de la A1 la A12 sunt acțiuni importante, precum trimiterea pachetelor sau rearmarea ceasurilor. Nu sunt menționate toate acțiunile minore, cum ar fi inițializarea unor câmpuri ale înregistrării atașate conexiunii. Dacă o acțiune implică trezirea unui proces în așteptare, atunci acțiunile care urmează trezirii procesului sunt considerate și ele. De exemplu, dacă un pachet CALL_REQUEST sosește și există un proces adormit care îl așteaptă, atunci transmiterea pachetului CALL_ACCEPT este considerată ca făcând parte din acțiunea care urmează recepției lui CALL_REQUEST. După ce este efectuată fiecare acțiune, conexiunea ajunge într-o nouă stare, așa cum apare și în fig. 6-21.



		Stare						
		Idle	Waiting	Queued	Established	Sending	Receiving	Disconnecting
Primitive	LISTEN	P1: ~1/Idle P2: A1/Estab P2: A2/Idle		~/Estab				
	CONNECT	P1: ~/Idle P1: A3/Wait						
	DISCONNECT				P4: A5/Idle P4: A6/Disc			
	SEND				P5: A7/Estab P5: A8/Send			
	RECEIVE				A9/Receiving			
Pachete sosite	Call_req	P3: A1/Estab P3: A4/Que'd						
	Call_acc		~/Estab					
	Clear_req		~/Idle		A10/Estab	A10/Estab	A10/Estab	~/Idle
	Clear_conf							~/Idle
	DataPkt						A12/Estab	
	Credit				A11/Estab	A7/Estab		
Ceas	Timeout			~/Idle				

Predicate

P1: Tabela de conexiuni plină

P2: Cerere de conexiune în așteptare

P3: Apel LISTEN în așteptare

P4: Pachet Clear_req în așteptare

P5: Credit disponibil

Acțiuni

A1: Trimite Call_acc

A2: Așteaptă Call_req

A3: Trimite Call_req

A4: Pornește ceasul

A5: Trimite Clear_conf

A6: Trimite Clear_req

A7: Trimite mesaj

A8: Așteaptă credit

A9: Trimite credit

A10: Setează indicator Clr_req_received

A11: Înregistrează credit

A12: Acceptă mesajul

Fig. 6-21. Protocolul ales ca exemplu, reprezentat ca un automat finit. Fiecare intrare are un predicat opțional, o acțiune opțională și o nouă stare. Caracterul tilda indică faptul că nici o acțiune importantă nu este efectuată. Bara deasupra predicatului este reprezentarea pentru predicatul negat. Intrările vide corespund unor secvențe de evenimente imposibile sau eronate.

Avantajul reprezentării protocolului ca o matrice este întreit. În primul rând, în această formă este mult mai simplu pentru programator să verifice sistematic fiecare combinație stare/ eveniment/ acțiune. În implementările reale, unele combinații vor fi folosite pentru tratarea erorilor. În fig. 6-21 nu s-a făcut nici o distincție între situațiile imposibile și cele care sunt numai ilegale. De exemplu, dacă o conexiune este în starea *WAITING*, evenimentul DISCONNECT este imposibil deoarece utilizatorul este blocat și nu poate să facă nici un apel. Pe de altă parte, în starea *SENDING* nu pot sosi date deoarece nu a fost generat nici un credit. Sosirea unui pachet de date va fi o eroare a protocolului.

Al doilea avantaj al reprezentării protocolului ca o matrice este legat de implementarea acestuia. Într-un vector bidimensional elementul *act[i][j]* ar putea fi văzut ca un indicator la procedura care tratează evenimentul i atunci când starea este j. O implementare posibilă este codificarea entității de transport ca o buclă în care se așteaptă la început producerea unui eveniment. După producerea evenimentului, este identificată conexiunea implicată și este extrasă starea ei. Cunoscând acum starea și evenimentul, entitatea de transport nu face decât să acceseze vectorul *act* și să apeleze procedura adecvată. Această abordare ar conduce la o arhitectură mult mai regulată și mai simetrică decât cea a entității de transport implementate de noi.

Al treilea avantaj al abordării folosind un automat finit este legat de descrierea protocolului. În unele standarde, protocoalele sunt specificate ca un automat finit de tipul celui din fig. 6-21. Trecerea de la acest tip de descriere la o entitate de transport funcțională este mult mai ușoară dacă entitatea de transport este și ea modelată cu ajutorul unui automat finit.

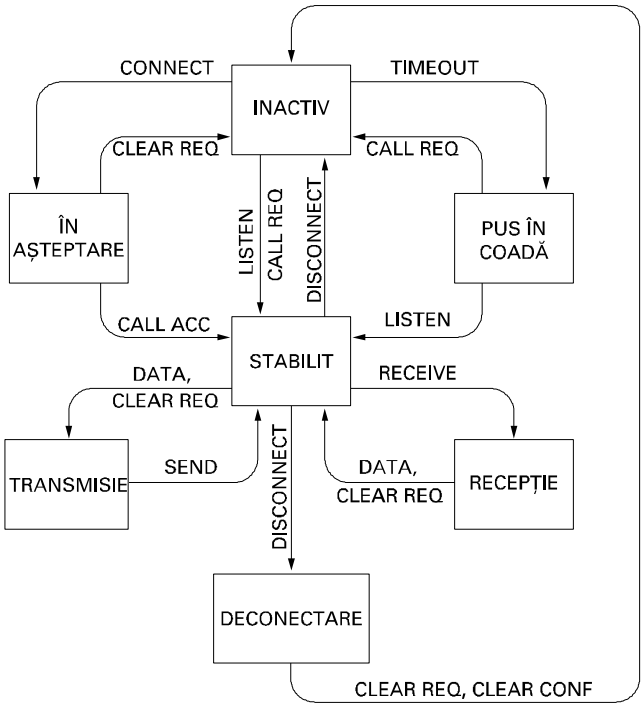


Fig. 6-22. Protocolul dat ca exemplu, în reprezentare grafică. Pentru simplificare, tranzițiile care lasă starea conexiunii nemodificată au fost omise.

Cel mai important dezavantaj este că această abordare poate să fie mai dificil de înțeles decât metoda directă pe care am utilizat-o la început. Totuși, această problemă poate fi rezolvată, fie și parțial, desenând automatul finit ca un graf, așa cum am făcut-o în fig. 6-22.

6.4 PROTOCOALE DE TRANSPORT PRIN INTERNET: UDP

Internet-ul are două protocoale principale în nivelul de transport: unul neorientat pe conexiune și unul orientat pe conexiune. În următoarele secțiuni o să le studiem pe ambele. Protocolul neorientat pe conexiune se numește UDP. Protocolul orientat pe conexiune se numește TCP. O să începem cu UDP-ul deoarece în esență este la fel ca IP-ul cu un mic antet adăugat. De asemeni, o să studiem și două aplicații ale UDP-ului.

6.4.1 Introducere în UDP

Setul de protocoale Internet suportă un protocol de transport fără conexiune, **UDP** (**User Protocol** – Protocol cu Datagrame Utilizator). UDP oferă aplicațiilor o modalitate de a trimite datagrame IP încapsulate și de a le transmite fără a fi nevoie să stabilească o conexiune. UDP este descris în RFC 768.

UDP transmite **segmente** constând într-un antet de 8 octeți urmat de informația utilă . Antetul este prezentat în fig. 6-23. Cele două porturi servesc la identificarea punctelor terminale ale mașinilor sursă și destinație. Când ajunge un pachet UDP, conținutul său este predat procesului atașat portului destinație. Această atașare are loc atunci când se folosește o simplă procedură de nume sau ceva asemănător, așa cum am văzut în fig. 6-6 pentru TCP (procesul de legătură este același pentru UDP). De fapt, valoarea cea mai importantă dată de existența UDP-ului față de folosirea doar a IP-ului simplu, este aceea a adăugării porturilor sursă și destinație. Fără câmpurile portului, nivelul de transport nu ar ști ce să facă cu pachetul. Cu ajutorul lor, segmentele se livrează corect.

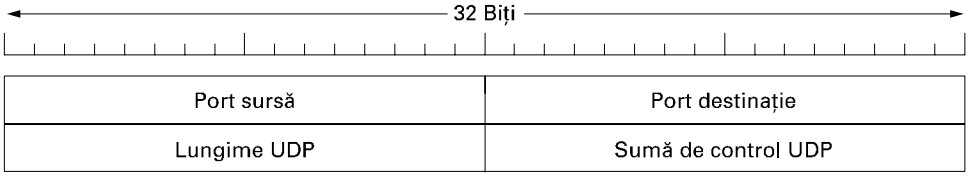


Fig. 6-23. Antetul UDP.

Portul sursă este în primul rând necesar atunci când un răspuns trebuie transmis înapoi la sursă. Prin copierea câmpului *port sursă* din segmentul care sosește în câmpul *port destinație* al segmentului care pleacă, procesul ce trimite răspunsul specifică ce proces de pe mașina de trimitere urmează să-l primească.

Câmpul l*ungime UDP* include antetul de 8 octeți și datele. Câmpul *sumă de control UDP* este opțional și stocat ca 0 (zero) dacă nu este calculat (o valoare de adevăr 0 rezultată în urma calculelor

este memorată ca un șir de biți 1). Dezactivarea acestuia este o prostie, excepție făcând cazul în care calitatea informației chiar nu contează (de exemplu, transmisia vocală digitalizată).

Merită probabil menționate, în mod explicit, unele dintre lucrurile pe care UDP-ul *nu le face*. Nu realizează controlul fluxului, controlul erorii, sau retransmiterea unui segment incorect primit. Toate acestea depind de procesele utilizatorului. Ceea ce face este să ofere protocolului IP o interfață cu facilități adăugate de demultiplexare a mai multor procese, folosind porturi. Aceasta este tot ceea ce face UDP-ul. Pentru aplicațiile care trebuie să aibă un control precis asupra fluxului de pachete, controlului erorii sau cronometrarea, UDP-ul furnizează doar ceea ce "a ordonat doctorul".

Un domeniu unde UDP-ul este în mod special util este acela al situațiilor client-server. Deseori, clientul trimite o cerință scurtă server-ului și așteaptă înapoi un răspuns scurt. Dacă se pierde ori cererea ori răspunsul, clientul poate pur și simplu să încerce din nou după ce a expirat timpul. Nu numai că va fi mai simplu codul, dar sunt necesare și mai puține mesaje (câte unul în fiecare direcție) decât la un protocol care solicită o inițializare inițială.

O aplicație care folosește UDP-ul în acest fel este DNS (Domain Name System, rom: Sistem de rezolvare de nume), pe care îl vom studia în cap. 7. Pe scurt, un program care trebuie să caute adresele de IP ale unor nume gazdă, de exemplu *www.cs.berkley.edu* , poate trimite un pachet UDP, conținând numele gazdă, către un server DNS. Serverul răspunde cu un pachet UDP conținând adresa de IP a gazdei. Nu este necesară nici o inițializare în avans și nici o închidere de sesiune. Doar două mesaje traversează rețeaua.

6.4.2 Apel de procedură la distanță (Remote Procedure Call)

Într-un anume sens, trimiterea unui mesaj către o stație la distanță și primirea înapoi a unui răspuns seamănă mult cu realizarea unei funcții de apel într-un limbaj de programare. În ambele cazuri se începe cu unul sau mai mulți parametri și se primește înapoi un rezultat. Această observație le-a făcut pe unele persoane să încerce să organizeze interacțiunile cerere-răspuns în rețele pentru a fi puse împreună sub forma apelurilor procedurale. Un astfel de aranjament face aplicațiile de rețea mai ușor programabile și mai abordabile. De exemplu, imaginați-vă doar procedura numită *get_IP_address(host_name)* care funcționează prin trimiterea unui pachet UDP către un server DNS și așteptarea răspunsului, cronometrând și încercând încă o dată, dacă răspunsul nu apare suficient de rapid. În acest fel, toate detaliile de rețea pot fi ascunse programatorului.

Efortul cel mai important în acest domeniu a fost depus de către Birell și Nelson (1984). Rezumând, ce au sugerat Birell și Nelson a fost să permită programelor să apeleze proceduri localizate pe stații aflate la distanță. Când procesul de pe mașina 1 invocă o procedură de pe mașina 2, procesul apelant de pe prima mașină este suspendat și execuția procedurii invocate are loc pe cea de-a doua. Informația poate fi transportată de la cel care apelează la cel care este apelat în parametri și se poate întoarce în rezultatul procedurii. Nici un transfer de mesaje nu este vizibil pentru programator. Tehnica este cunoscută sub numele de **RPC** (**Remote Procedure Call** , rom: Apel de procedură la distanță), și a devenit baza pentru multe aplicații de rețea. În mod tradițional, procedura care apelează este cunoscută ca fiind clientul și procedura apelată ca fiind serverul și vom folosi aceste denumiri și aici.

Ideea din spatele RPC-ului este aceea de a face un apel de procedură la distanță să arate pe cât posibil ca unul local. În forma cea mai simplă, pentru apelarea unei proceduri la distanță, programul client trebuie sa fie legat cu o mică procedură de bibliotecă, numită **client stub** (rom: ciot), care reprezintă procedura server-ului în spațiul de adresă al clientului. În mod similar, serverul este legat cu

o procedură numită **server stub** . Aceste proceduri ascund faptul că apelul de procedură de la client la server nu este local.

Pașii efectivi ai realizării unui RPC sunt prezentați în fig. 6-24. Pasul 1 este cel în care clientul apelează stub-ul client. Acest apel este un apel de procedură locală, cu parametrii introduși în stivă în modul obișnuit. Pasul 2 constă în împachetarea parametrilor de către stub-ul client într-un mesaj și realizarea unui apel de sistem pentru a trimite mesajul. Împachetarea parametrilor este denumită **marshaling** (rom: împachetare). Pasul 3 constă în faptul că nucleul sistemului de operare trimite un mesaj de la mașina client la mașina server. Pasul 4 constă în trimiterea de către nucleu a pachetelor care sosesc la stub-ul server. În sfârșit, pasul 5 constă în faptul că stub-ul server apelează procedura server cu parametrii despachetați. Răspunsul urmează aceeași cale și în cealaltă direcție.

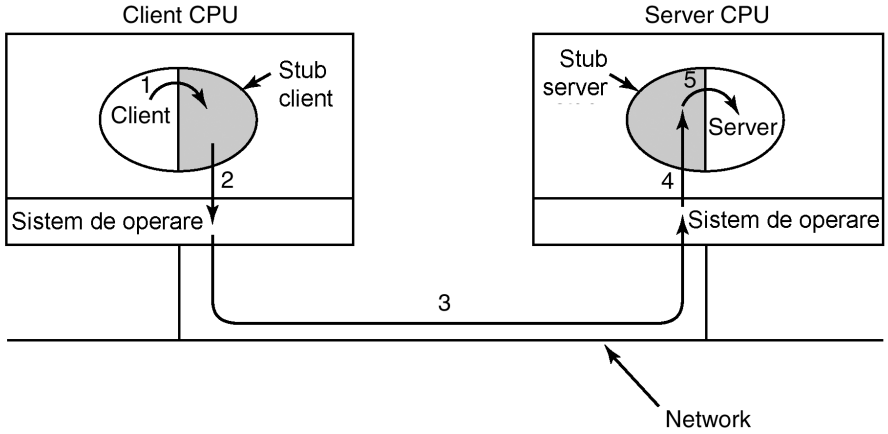


Fig. 6-24. Pașii pentru a crea un apel de procedură la distanță. Stub-urile sunt hașurate.

Elementul cheie de reținut aici este acela că procedura client, scrisă de către utilizator, doar face un apel normal de procedură (adică local) către stub-ul client, care are același nume cu procedura server. Cum procedura client și stub-ul client se găsesc în același spațiu de adresă, parametrii sunt transferați în modul obișnuit. În mod asemănător, procedura server este apelată de o procedură din spațiul său de adresă cu parametrii pe care îi așteaptă. Pentru procedura server nimic nu este neobișnuit. În felul acesta, în loc ca intrarea/ieșirea să se facă pe socluri, comunicația de rețea este realizată imitând o procedură de apelare normală.

În ciuda eleganței conceptului de RPC, „sunt câțiva șerpi care se ascund prin iarbă”. Unul mare este utilizarea parametrilor pointer. În mod normal, transmiterea unui pointer către procedură nu este o problemă. Procedura apelată poate folosi pointer-ul în același mod în care poate să o facă și cel care o apelează, deoarece ambele proceduri se găsesc în același spațiu de adrese virtuale. Cu RPC-ul, transmiterea pointer-ilor este imposibilă deoarece clientul și server-ul se găsesc în spații de adresă diferite.

În anumite cazuri, se pot folosi unele trucuri pentru a face posibilă transmiterea pointer-ilor. Să presupunem că primul parametru este un pointer către un întreg, *k*. Stub-ul client poate împacheta variabila *k* și să o trimită server-ului. Atunci, server stub creează un pointer către *k* și-l transmite procedurii server, exact așa cum aceasta se așteaptă. Când procedura server cedează controlul server stub, acesta din urmă trimite variabila *k* înapoi clientului, unde noul *k* este copiat peste cel vechi, în caz că serverul l-a schimbat. În fapt, secvența standard de apelare apel-prin-referință a fost înlocuită

de copiază-restaurează (eng.: copy-restore). Din păcate, acest truc nu funcționează întotdeauna, de exemplu dacă un pointer este către un grafic sau altă structură complexă de date. Din acest motiv, trebuie puse anumite restricții asupra parametrilor procedurilor apelate la distanță.

O a doua problemă este aceea că în limbajelor mai puțin bazate pe tipuri, cum ar fi C-ul, este perfect legal să scrii o procedură care calculează produsul scalar a doi vectori, fără a specifica dimensiunea vreunuia dintre ei. Fiecare poate fi terminat printr-o valoare specială cunoscută doar de către procedura apelată și de cea apelantă. În aceste condiții, în mod cert este imposibil pentru stub-ul client să împacheteze parametrii: nu are nici o modalitate de a determina cât de mult spațiu ocupă aceștia.

O a treia problemă este aceea că nu întotdeauna este posibilă deducerea tipurilor parametrilor, nici măcar dintr-o specificație formală sau din cod în sine. Un exemplu este *printf,* care poate avea orice număr de parametri (cel puțin unul), iar parametrii pot fi o combinație arbitrară a tipurilor întregi, short, long, caractere, șiruri, numere în virgulă mobilă de diferite lungimi și alte tipuri. A încerca să invoci *printf* ca procedură cu apel la distanță ar fi practic imposibil, deoarece C-ul este prea permisiv. Totuși, o regulă care să spună că RPC-ul poate fi folosit cu condiția să nu programezi în C (sau C++) nu ar fi prea populară.

O a patra problemă este legată de utilizarea variabilelor globale. În mod normal, procedura de apelare și cea care este apelată pot comunica folosind variabilele globale, în plus față de comunicarea prin parametri. Dacă procedura apelată este mutată acum pe o mașină la distanță, codul va da erori deoarece variabilele globale nu mai sunt partajate.

Aceste probleme nu sunt menite să sugereze că RPC-ul este lipsit de șanse. De fapt, este larg folosit, dar sunt necesare anumite restricții pentru a-l face să funcționeze bine în practică.

Desigur, RPC-ul nu are nevoie să folosească pachete UDP, dar RPC și UDP se potrivesc bine, și UDP este uzual folosit pentru RPC. Totuși, când parametrii sau rezultatele pot să fie mai mari decât pachetul maxim UDP sau atunci când operația cerută nu este idempotentă (adică nu poate fi repetată în siguranță, ca de exemplu atunci când se incrementează un contor), poate fi necesară stabilirea unei conexiuni TCP și trimiterea cererii prin aceasta, în loc să se folosească UDP-ul.

6.4.3 Protocolul de transport în timp real – Real-Time Transport Protocol

RPC-ul client-server este un domeniu în care UDP este mult folosit. Un alt domeniu este acela al aplicațiilor multimedia în timp real. În particular, având în vedere că radioul pe internet, telefonia pe Internet, muzica la cerere, video-conferințele, video la cerere și alte aplicații multimedia au devenit mai răspândite, oamenii au descoperit că fiecare aplicație a folosit, mai mult sau mai puțin, același protocol de transport în timp real. Treptat a devenit clar faptul că un protocol generic de transport în timp real, pentru aplicații multimedia, ar fi o idee bună. Așa a luat naștere **RTP**-ul (**Real-time Transport Protocol**, rom: Protocol de transport în timp real). Este descris în RFC 1889 și acum este folosit pe scară largă.

Poziția RTP-ului în stiva de protocoale este oarecum ciudată. S-a hotărât să se pună RTP-ul în spațiul utilizator și să se ruleze (în mod normal) peste UDP. El funcționează după cum urmează. Aplicațiile multimedia constau în aplicații audio, video, text și posibil alte fluxuri. Acestea sunt trimise bibliotecii RTP, care se află în spațiul utilizator împreună cu aplicația. Apoi, această bibliotecă multiplexează fluxurile și le codează în pachete RTP, pe care apoi le trimite printr-un soclu. La celălalt capăt al soclului (în nucleul sistemului de operare), pachete UDP sunt generate și încapsulate în pachete IP. Dacă computer-ul se găsește într-o rețea Ethernet, pachetele IP sunt puse apoi în cadre

Ethernet, pentru transmisie. Stiva de protocoale pentru această situație este prezentată în fig. 6-25 (a). Încapsularea pachetului este prezentată în fig. 6-25 (b).

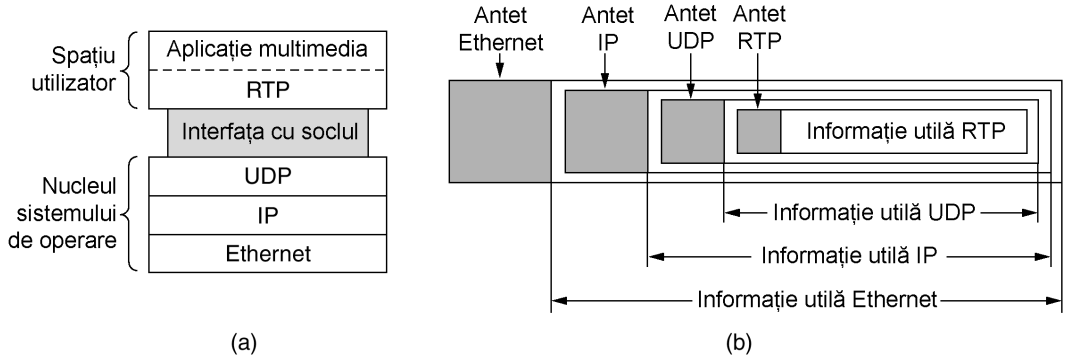


Fig. 6-25. (a)Poziționarea Protocolului RTP în stiva de protocoale. (b) Încapsularea pachetului.

Ca o consecință a acestei proiectări, este cam dificil de spus în ce nivel este RTP-ul. Cum rulează în spațiul utilizator și este legat la programul aplicație, în mod cert arată ca un protocol de aplicație. Pe de altă parte, este un protocol generic independent de aplicație, care doar furnizează facilități de transport, astfel încât arată totodată ca un protocol de transport. Probabil că cea mai potrivită descriere este aceea că este un protocol de transport care este implementat la nivelul aplicație.

Funcția de bază a RTP-ului este multiplexarea mai multor fluxuri de date în timp real într-un singur flux de pachete UDP. Fluxul UDP poate fi transmis către o singură destinație (unicasting) sau către destinații multiple (multicasting). Deoarece RTP-ul folosește numai UDP normal, pachetele sale nu sunt tratate în mod special de către rutere, decât dacă sunt activate anumite facilități de calitate a serviciilor din IP. În particular, nu există garanții speciale referitoare la livrare, bruiaj etc.

Fiecărui pachet trimis în fluxul RTP i se dă un număr cu unu mai mare decât al predecesorului său. Această numerotare permite destinației să stabilească dacă lipsesc unele pachete. Dacă un pachet lipsește, cea mai bună decizie ce poate fi luată de către destinație este de a aproxima valoarea lipsă prin interpolare. Retransmiterea nu este o opțiune practică având în vedere că pachetul retransmis va ajunge probabil prea târziu pentru a fi util. Ca o consecință, RTP-ul nu are control al fluxului, control al erorii, nu are confirmări și nu are mecanism pentru a cere retransmiterea.

Fiecare informație utilă din RTP poate să conțină mostre multiple și ele pot fi codate în orice mod dorește aplicația. Pentru a permite compatibilitatea , RTP-ul definește mai multe profiluri (de exemplu un singur flux audio) și pentru fiecare profil pot fi permise multiple formate de codare. De exemplu, un singur flux audio poate fi codat ca mostre de 8 biți PCM la 8 KHz, codare delta, codare previzibilă, codare GSM, MP3 și așa mai departe. RTP-ul furnizează un câmp antet în care sursa poate specifica codarea, dar altfel nu este implicat în modul în care este făcută codarea.

O altă facilitate de care au nevoie multe aplicații multimedia este stabilirea amprentei de timp. Aici ideea este de a permite sursei să asocieze o amprentă de timp cu prima mostră din fiecare pachet. Amprentele de timp sunt relative la începutul fluxului, așa că numai diferențele dintre acestea sunt semnificative. Valorile absolute nu au nici o semnificație. Acest mecanism permite destinației să folosească zone tampon de dimensiuni mici și să reproducă fiecare eșantion la numărul corect de milisecunde după începutul fluxului, independent de momentul în care ajunge pachetul ce conține eșantionul. Stabilirea amprentelor de timp nu numai că reduce efectele bruiajului, ci permite de asemenea mai multor fluxuri să se sincronizeze între ele. De exemplu, un program de televiziune

digital poate avea un flux video și două fluxuri audio. Cele două fluxuri audio pot fi pentru emisiuni stereo sau pentru a permite filmelor să fie manipulate cu o coloană sonoră în limba originală și o coloană sonoră dublată în limba locală, oferind o alegere celui care vede. Fiecare flux provine dintr-un dispozitiv fizic diferit, dar dacă sunt stabilite amprentele de timp de către un singur contor, ele pot fi redate sincronizat, chiar dacă fluxurile sunt transmise într-un mod dezordonat.

Antetul RTP este ilustrat în fig. 6-26. Acesta constă din trei cuvinte de 32 biți și eventual unele extensii. Primul cuvânt conține câmpul *Versiune*, care este deja la 2. Să sperăm că această versiune este foarte asemănătoare cu ultima versiune deoarece a mai rămas aici doar un punct de cod (deși 3 ar putea fi definit ca semnificând faptul că versiunea reală se găsește într-un cuvânt extins).

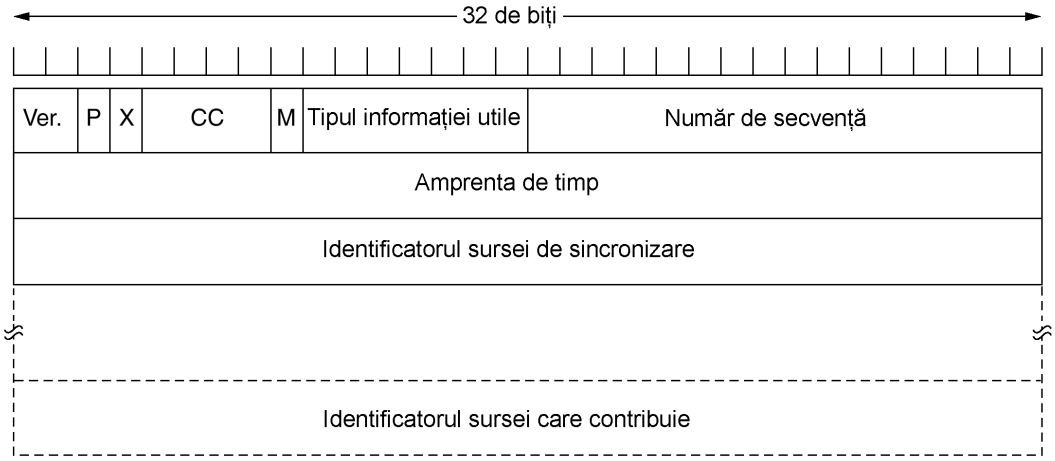


Fig. 6-26. Antetul RTP-ului.

Bitul *P* indică faptul că pachetul a fost extins la un multiplu de 4 octeți. Ultimul octet extins ne spune câți octeți au fost adăugați. Bitul *X* indică prezența unui antet extins. Formatul și semnificația antetului extins nu sunt definite. Singurul lucru care este definit este acela că primul cuvânt al extensiei dă lungimea. Aceasta este o cale de scăpare pentru orice cerințe neprevăzute.

Câmpul *CC* arată câte surse contribuabile sunt prezente, de la 0 la 15 (vezi mai jos). Bitul *M* este un bit de marcare specific aplicației. Poate fi folosit pentru a marca începutul unui cadru video, începutul unui cuvânt într-un canal audio sau altceva ce aplicația înțelege. Câmpul *Tip informație utilă* indică ce algoritm de codare a fost folosit (de exemplu 8 biți audio necompresați, MP3, etc). Din moment ce fiecare pachet transportă acest câmp, codarea se poate schimba în timpul transmisiei. *Numărul de secvență* este doar un contor care este incrementat pe fiecare pachet RTP trimis. Este folosit pentru a detecta pachetele pierdute.

Amprenta de timp este stabilită de către sursa fluxului pentru a se ști când este făcut primul eșantion din pachet. Această valoare poate să ajute la reducerea bruiajului la receptor prin separarea redării de momentul ajungerii pachetului. *Identificatorul sursei de sincronizare* spune cărui flux îi aparține pachetul. Este metoda utilizată pentru a multiplexa și demultiplexa mai multe fluxuri de date într-un singur flux de pachete UDP. În sfârșit, dacă există, *identificatorii sursei care contribuie* sunt folosiți când în studio există mixere. În acest caz, mixerul este sursa de sincronizare și fluxurile, fiind amestecate, apar aici.

RTP are un protocol înrudit numit **RTCP** (**Real Time Transport Control Protocol**, rom: Protocol de control al transportului în timp real). Acesta se ocupă de răspuns, sincronizare și de interfața cu utilizatorul, dar nu transportă date. Prima funcție poate fi folosită pentru a oferi surselor reacție (eng.: feedback) la întârzieri, bruiaj, lățime de bandă, congestie și alte proprietăți ale rețelei. Această informație poate fi folosită de către procesul de codare pentru a crește rata de transfer a datelor (și să ofere o calitate mai bună) când rețeaua merge bine și să reducă rata de transfer când apar probleme pe rețea. Prin furnizarea continuă de răspunsuri, algoritmii de codare pot fi în continuu adaptați pentru a oferi cea mai bună calitate posibilă în circumstanțele curente. De exemplu, dacă lățimea de bandă crește sau scade în timpul transmisiei, codarea poate să se schimbe de la MP3 la PCM pe 8 biți la codare delta, așa cum se cere. Câmpul *Tip informație utilă* este folosit pentru a spune destinației ce algoritm de codare este folosit pentru pachetul curent, făcând posibilă schimbarea acestuia la cerere.

De asemenea, RTCP-ul se ocupă de sincronizarea între fluxuri. Problema este că fluxuri diferite pot folosi ceasuri diferite, cu granularități diferite și devieri de flux diferite. RTCP poate fi folosit pentru a le menține sincronizate.

În sfârșit, RTCP oferă un mod pentru a numi diversele surse (de exemplu în text ASCII). Această informație poate fi afișată pe ecranul receptorului pentru a indica cine vorbește în acel moment.

Mai multe informații despre RTP pot fi găsite în (Perkins, 2002).

6.5. PROTOCOALE DE TRANSPORT PRIN INTERNET: TCP

UDP-ul este un protocol simplu și are anumite nișe de utilizare, cum ar fi interacțiunile client-server și cele multimedia, dar pentru cele mai multe aplicații de Internet este necesar un transport de încredere, secvențial al informației. UDP-ul nu poate oferi acest lucru, deci este nevoie de un alt protocol. Acesta este TCP și este pionul principal de lucru al Internet-ului. Să-l studiem acum în amănunt.

6.5.1 Introducere în TCP

TCP (**Transport Communication Protocol** - protocol de comunicație de nivel transport) a fost proiectat explicit pentru a asigura un flux sigur de octeți de la un capăt la celălalt al conexiunii într-o inter-rețea nesigură. O inter-rețea diferă de o rețea propriu-zisă prin faptul că diferite părți ale sale pot diferi substanțial în topologie, lărgime de bandă, întârzieri, dimensiunea pachetelor și alți parametri. TCP a fost proiectat să se adapteze în mod dinamic la proprietățile inter-rețelei și să fie robust în ceea ce privește mai multe tipuri de defecte.

TCP a fost definit în mod oficial în RFC 793. O dată cu trecerea timpului, au fost detectate diverse erori și inconsistențe și au fost modificate cerințele în anumite subdomenii. Aceste clarificări, precum și corectarea câtorva erori sunt detaliate în RFC 1122. Extensiile sunt furnizate în RFC 1323.

Fiecare mașină care suportă TCP dispune de o entitate de transport TCP, fie ca proces utilizator, fie ca procedură de bibliotecă, fie ca parte a nucleului. În toate aceste cazuri, ea care gestionează fluxurile TCP și interfețele către nivelul IP. O entitate TCP acceptă fluxuri de date utilizator de la procesele locale, le împarte în fragmente care nu depășesc 64K octeți (de regulă în fragmente de aproximativ 1460 de octeți, pentru a încăpea într-un singur cadru Ethernet împreună cu antetele

TCP și IP) și expediază fiecare fragment ca o datagramă IP separată. Atunci când datagramele IP conținând informație TCP sosesc la o mașină, ele sunt furnizate entității TCP, care reconstruiește fluxul original de octeți. Pentru simplificare, vom folosi câteodată doar TCP , subînțelegând prin aceasta sau entitatea TCP de transport (o porțiune de program) sau protocolul TCP (un set de reguli). Din context va fi clar care din cele două noțiuni este referită. De exemplu, în „Utilizatorul furnizează date TCP-ului" este clară referirea la entitatea TCP de transport.

Nivelul IP nu oferă nici o garanție că datagramele vor fi livrate corect, astfel că este sarcina TCP-ului să detecteze eroarea și să efectueze o retransmisie atunci când situația o impune. Datagramele care ajung (totuși) la destinație pot sosi într-o ordine eronată; este, de asemenea, sarcina TCP-ului să le reasambleze în mesaje respectând ordinea corectă (de secvență). Pe scurt, TCP-ul trebuie să ofere fiabilitatea pe care cei mai mulți utilizatori o doresc și pe care IP-ul nu o oferă.

6.5.2 Modelul serviciului TCP

Serviciul TCP este obținut prin crearea atât de către emițător, cât și de către receptor, a unor puncte finale, numite socluri (sockets), așa cum s-a discutat în Sec. 6.1.3. Fiecare soclu are un număr de soclu (adresă) format din adresa IP a mașinii gazdă și un număr de 16 biți, local gazdei respective, numit **port**. Port este numele TCP pentru un TSAP. Pentru a obține o conexiune TCP, trebuie stabilită explicit o conexiune între un soclu de pe mașina emițătoare și un soclu de pe mașina receptoare. Apelurile de soclu sunt prezentate în fig. 6-5.

Un soclu poate fi folosit la un moment dat pentru mai multe conexiuni. Altfel spus, două sau mai multe conexiuni se pot termina la același soclu. Conexiunile sunt identificate prin identificatorii soclurilor de la ambele capete, adică *(soclu 1, soclu 2)*. Nu este folosit nici un alt număr sau identificator de circuit virtual.

Numerele de port mai mici decât 256 se numesc **porturi general cunoscute** și sunt rezervate serviciilor standard. De exemplu, orice proces care dorește să stabilească o conexiune cu o mașină gazdă pentru a transfera un fișier utilizând FTP, se poate conecta la portul 21 al mașinii destinație pentru a contacta demonul său FTP. Similar, portul 23 este folosit pentru a stabili o sesiune de lucru la distanță utilizând TELNET. Lista porturilor general cunoscute se găsește la *www.iana.org*. Câteva dintre cele foarte cunoscute sunt prezentate în fig. 6-27.

Port	Protocol	Utilitate
21	FTP	Transfer de fișiere
23	Telnet	Login la distanță
25	SMTP	E-mail
69	TFTP	Protocol de transfer de fișiere trivial
79	Finger	Căutare de informații despre un utilizator
80	HTTP	World Wide Web
110	POP-3	Acces prin e-mail la distanță
119	NNTP	Știri USENET

Fig. 6-27. Câteva porturi asignate.

Cu siguranță ar fi posibil ca, în momentul încărcării, demonul de FTP să se autoatașeze la portul 21, demonul telnet la portul 23 și tot așa. Totuși, dacă s-ar proceda astfel s-ar umple memoria cu demoni inactivi în majoritatea timpului. În schimb, în general se folosește un singur demon, numit **inetd** (**Internet daemon**, rom: demon de Internet) în UNIX, care să se autoatașeze la mai multe porturi și să aștepte prima conexiune care vine. Când acest lucru se întâmplă, inetd creează un nou pro-

ces și execută în el demonul adecvat, lăsând acel demon să se ocupe de cerere. Astfel, demonii, în afară de inetd, sunt activi doar când au de lucru. Inetd află ce porturi să folosească dintr-un fișier de configurare. În consecință, administratorul de sistem poate seta sistemul să aibă demoni permanenți pe cele mai ocupate porturi (de exemplu portul 80) și inetd pe restul.

Toate conexiunile TCP sunt duplex integral și punct-la-punct. Duplex integral înseamnă că traficul se poate desfășura în ambele sensuri în același timp. Punct-la-punct indică faptul că fiecare conexiune are exact două puncte finale. TCP nu suportă difuzarea parțială sau totală.

O conexiune TCP este un flux de octeți și nu un flux de mesaje. Dimensiunile mesajelor nu se conservă de la un capăt la celălalt. De exemplu, dacă procesul emițător execută patru scrieri de câte 512 octeți pe un canal TCP, aceste date pot fi livrate procesului receptor ca patru fragmente (chunks) de 512 octeți, două fragmente de 1024 octeți, un singur fragment de 2048 octeți (vezi fig. 6-28) sau în orice alt mod. Nu există posibilitatea ca receptorul să determine numărul de unități în care a fost scrisă informația.

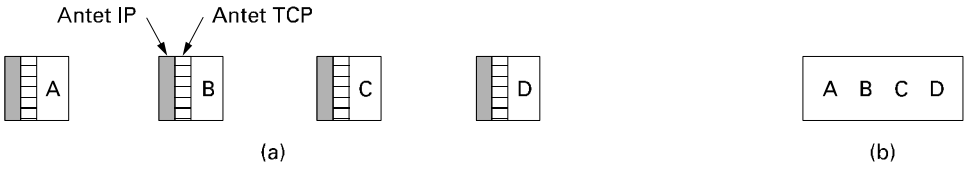


Fig. 6-28. (a) Patru segmente de 512 octeți au fost trimise ca datagrame IP separate. (b) Livrarea celor 2048 octeți către aplicație, printr-un singur apel *read*.

În UNIX, aceeași proprietate o au și fișierele. Cititorul unui fișier nu poate spune dacă fișierul a fost scris bloc cu bloc, octet cu octet sau tot dintr-o dată. Ca și un fișier UNIX, programele TCP nu au nici cea mai vagă idee despre semnificația octeților și nici cel mai mic interes pentru a afla acest lucru. Un octet este pur și simplu un octet.

Atunci când o aplicație trimite date către TCP, TCP-ul le poate expedia imediat sau le poate reține într-un tampon (în scopul colectării unei cantități mai mari de informație pe care să o expedieze toată odată), după bunul său plac. Cu toate acestea, câteodată, aplicația dorește ca informația să fie expediată imediat. De exemplu, să presupunem că un utilizator este conectat la o mașină de la distanță. După ce a fost terminată o linie de comandă și s-a tastat Return, este esențial ca linia să fie imediat expediată către mașina de la distanță și să nu fie memorată până la terminarea următoarei linii. Pentru a forța expedierea, aplicația poate folosi indicatorul PUSH, care îi semnalează TCP-ului să nu întârzie procesul de transmisie.

Unele din primele aplicații foloseau indicatorul PUSH ca un fel de marcaj pentru a delimita marginile mesajelor. Deși acest truc funcționează câteodată, uneori el eșuează datorită faptului că, la recepție, nu toate implementările TCP-ului transmit aplicației indicatorul PUSH. Mai mult decât atât, dacă mai multe indicatoare PUSH apar înainte ca primul să fi fost transmis (de exemplu, pentru că linia de legătură este ocupată), TCP-ul este liber să colecteze toată informația referită de către aceste indicatoare într-o singură datagramă IP, fără să includă nici un separator între diferitele sale părți.

O ultimă caracteristică a serviciului TCP care merită menționată aici constă în **informația urgentă**. Atunci când un utilizator apasă tasta DEL sau CTRL-C pentru a întrerupe o prelucrare la distanță, aflată deja în execuție, aplicația emițător plasează o informație de control în fluxul de date și o furnizează TCP-ului împreună cu indicatorul URGENT. Acest eveniment impune TCP-ului întreruperea acumulării de informație și transmisia imediată a întregii informații disponibile deja pentru conexiunea respectivă.

Atunci când informația urgentă este recepționată la destinație, aplicația receptoare este întreruptă (de ex. prin emisia unui semnal, în terminologie UNIX), astfel încât, eliberată de orice altă activitate, aplicația să poată citi fluxul de date și să poată regăsi informația urgentă. Sfârșitul informației urgente este marcat, astfel încât aplicația să știe când se termină informația. Începutul informației urgente nu este marcat. Este sarcina aplicației să determine acest început. Această schemă furnizează de fapt un rudiment de mecanism de semnalizare, orice alte detalii fiind lăsate la latitudinea aplicației.

6.5.3 Protocolul TCP

În această secțiune vom prezenta un punct de vedere general asupra protocolului TCP, pentru a ne concentra apoi, în secțiunea care îi urmează, asupra antetului protocolului, câmp cu câmp.

O caracteristică importantă a TCP, care domină structura protocolului, este aceea că fiecare octet al unei conexiuni TCP are propriul său număr de secvență, reprezentat pe 32 biți. Când a luat ființă Internetul, liniile dintre rutere erau în cel mai bun caz linii închiriate de 56 Kbps, deci unei gazde funcționând la viteză maximă îi lua mai mult de o săptămână să utilizeze toate numerele de secvență. La vitezele rețelelor moderne, numerele de secvență pot fi consumate intr-un ritm alarmant, după cum vom vedea mai târziu. Numerele de secvență sunt utilizate atât pentru confirmări cât și pentru mecanismul de secvențiere, acesta din urmă utilizând câmpuri separate de 32 de biți din antet.

Entitățile TCP de transmisie și de recepție interschimbă informație sub formă de segmente. Un **segment TCP** constă dintr-un antet de exact 20 de octeți (plus o parte opțională) urmat de zero sau mai mulți octeți de date. Programul TCP este cel care decide cât de mari trebuie să fie aceste segmente. El poate acumula informație provenită din mai multe scrieri într-un singur segment sau poate fragmenta informația provenind dintr-o singură scriere în mai multe segmente. Există două limite care restricționează dimensiunea unui segment. În primul rând, fiecare segment, inclusiv antetul TCP, trebuie să încapă în cei 65.535 de octeți de informație utilă IP. În al doilea rând, fiecare rețea are o **unitate maximă de transfer** sau **MTU (Maximum Transfer Unit)**, deci fiecare segment trebuie să încapă în acest MTU. În realitate, MTU este în general de 1500 octeți (dimensiunea informației utile din Ethernet), definind astfel o limită superioară a dimensiunii unui segment.

Protocolul de bază utilizat de către entitățile TCP este protocolul cu fereastră glisantă. Atunci când un emițător transmite un segment, el pornește un cronometru. Atunci când un segment ajunge la destinație, entitatea TCP receptoare trimite înapoi un segment (cu informație utilă, dacă aceasta există sau fără, în caz contrar) care conține totodată și numărul de secvență următor pe care aceasta se așteaptă să-l recepționeze. Dacă cronometrul emițătorului depășește o anumită valoare înaintea primirii confirmării, emițătorul retransmite segmentul neconfirmat.

Deși acest protocol pare simplu, pot apărea multe situații particulare pe care le vom prezenta mai jos. Segmentele pot ajunge într-o ordine arbitrară, deci octeții 3072-4095 pot fi recepționați, dar nu pot fi confirmați datorită absenței octeților 2048-3071. Segmentele pot de asemenea întârzia pe drum un interval de timp suficient de mare pentru ca emițătorul să detecteze o depășire a cronometrului și să le retransmită. Retransmisiile pot include porțiuni de mesaj fragmentate altfel decât în transmisia inițială, ceea ce impune o tratare atentă, astfel încât să se țină evidența octeților primiți corect. Totuși, deoarece fiecare octet din flux are un deplasament unic față de începutul mesajului, acest lucru se poate realiza.

TCP trebuie să fie pregătit să facă față unor astfel de situații și să le rezolve într-o manieră eficientă. Un efort considerabil a fost dedicat optimizării performanțelor fluxurilor TCP, ținându-se cont inclusiv de probleme legate de rețea. În continuare vor fi prezentați un număr de algoritmi utilizați de numeroase implementări TCP.

6.5.4 Antetul segmentului TCP

În fig. 6-29 este prezentată structura unui segment TCP. Fiecare segment începe cu un antet format dintr-o structură fixă de 20 de octeți. Antetul fix poate fi urmat de un set de opțiuni asociate antetului. În continuarea opțiunilor, dacă ele există, pot urma până la 65.535 - 20 - 20 = 65.495 de octeți de date, unde primul 20 reprezintă antetul IP, iar al doilea antetul TCP. Segmente care nu conțin octeți de date sunt nu numai permise, dar și utilizate în mod frecvent pentru confirmări și mesaje de control.

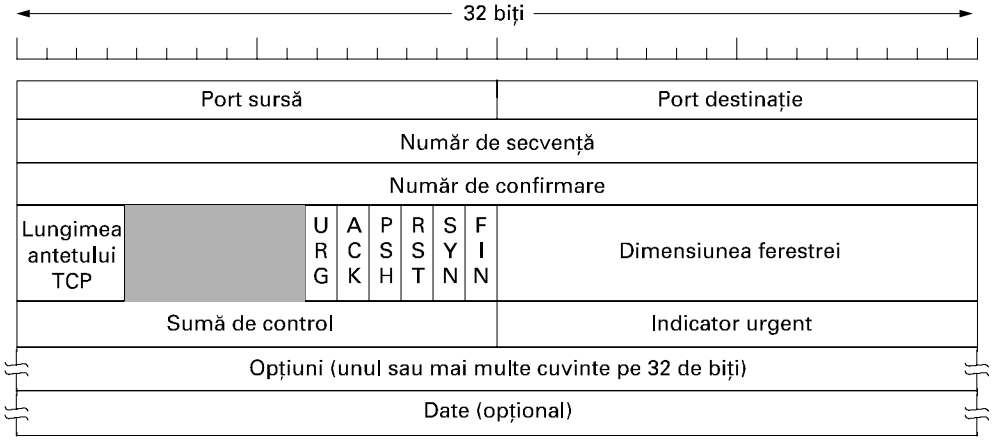


Fig. 6-29. Antetul TCP.

Să disecăm acum structura antetului TCP, câmp cu câmp. Câmpurile *Port sursă* și *Port destinație* identifică punctele finale ale conexiunii. Porturile general cunoscute sunt definite la *www.iana.org*, dar fiecare gazdă le poate aloca pe celelalte după cum dorește. Un port formează împreună cu adresa IP a mașinii sale un unic punct de capăt (eng.: end point) de 48 de biți. Conexiunea este identificată de punctele de capăt ale sursei și destinației.

Câmpurile *Număr de secvență* și *Număr de confirmare* au semnificația funcțiilor lor uzuale. Trebuie observat că cel din urmă indică octetul următor așteptat și nu ultimul octet recepționat în mod corect. Ambele câmpuri au lungimea de 32 de biți, deoarece într-un flux TCP fiecare bit de informație este numerotat.

Lungimea antetului TCP indică numărul de cuvinte de 32 de biți care sunt conținute în antetul TCP. Această informație este utilă, deoarece câmpul *Opțiuni* este de lungime variabilă, proprietate pe care o transmite astfel și antetului. Tehnic vorbind, acest câmp indică în realitate începutul datelor din segment, măsurat în cuvinte de 32 de biți, dar cum acest număr este identic cu lungimea antetului în cuvinte, efectul este același.

Urmează un câmp de șase biți care este neutilizat. Faptul că acest câmp a supraviețuit intact mai mult de un sfert de secol este o mărturie despre cât de bine a fost proiectat TCP-ul. Protocoale mai prost concepute ar fi avut nevoie de el pentru a corecta erori ale proiectării inițiale.

Urmează acum șase indicatori de câte un bit. URG este poziționat pe 1 dacă Indicatorul Urgent este valid. Indicatorul Urgent este folosit pentru a indica deplasamentul în octeți față de numărul curent de secvență la care se găsește informația urgentă. O astfel de facilitate ține locul mesajelor de

întrerupere. Așa cum am menționat deja anterior, această facilitate reprezintă esența modului în care emițătorul poate transmite un semnal receptorului fără ca TCP-ul în sine să fie cauza întreruperii.

Bitul *ACK* este poziționat pe 1 pentru a indica faptul că *Numărul de confirmare* este valid. În cazul în care *ACK* este poziționat pe 0, segmentul în discuție nu conține o confirmare și câmpul *Număr de confirmare* este ignorat.

Bitul *PSH* indică informația FORȚATĂ. Receptorul este rugat respectuos să livreze aplicației informația respectivă imediat ce este recepționată și să nu o memoreze în așteptarea umplerii tampoanelor de comunicație (lucru care, altminteri, ar fi făcut din rațiuni de eficiență).

Bitul *RST* este folosit pentru a desființa o conexiune care a devenit inutilizabilă datorită defecțiunii unei mașini sau oricărui alt motiv. El este de asemenea utilizat pentru a refuza un segment invalid sau o încercare de deschidere a unei conexiuni. În general, recepționarea unui segment având acest bit poziționat indică o problemă care trebuie tratată în funcție de context.

Bitul *SYN* este utilizat pentru stabilirea unei conexiuni. Cererea de conexiune conține $SYN = 1$ și $ACK = 0$ pentru a indica faptul că acel câmp suplimentar de confirmare nu este utilizat. Răspunsul la o astfel de cerere conține o confirmare, având deci $SYN = 1$ și $ACK = 1$. În esență, bitul *SYN* este utilizat pentru a indica o CERERE DE CONEXIUNE și o CONEXIUNE ACCEPTATĂ, bitul *ACK* făcând distincția între cele două posibilități.

Bitul *FIN* este folosit pentru a încheia o conexiune. El indică faptul că emițătorul nu mai are nici o informație de transmis. Cu toate acestea, după închiderea conexiunii, un proces poate recepționa în continuare date pe o durată nedefinită. Ambele segmente, *SYN* și *FIN,* conțin numere de secvență și astfel este garantat faptul că ele vor fi prelucrate în ordinea corectă.

În TCP, fluxul de control este tratat prin ferestre glisante de dimensiune variabilă. Câmpul *Fereastră* indică numărul de octeți care pot fi trimiși, începând de la octetul confirmat. Un câmp *Fereastră* de valoare 0 este perfect legal și spune că octeții până la *Număr de confirmare* - 1 inclusiv au fost recepționați, dar receptorul dorește cu ardoare o pauză, așa că mulțumește frumos, dar pentru moment nu dorește continuarea transferului. Permisiunea de expediere poate fi acordată ulterior de către receptor prin trimiterea unui segment având același *Număr de confirmare,* dar un câmp *Fereastră* cu o valoare nenulă.

În protocoalele din cap. 3, confirmările pentru cadrele primite și permisiunea de a trimite noi cadre erau legate una de alta. Aceasta era o consecință a dimensiunii fixe a ferestrei pentru fiecare protocol. În TCP, confirmările și permisiunea de a trimite noi date sunt total decuplate. De fapt, receptorul poate spune: Am primit octeții până la al *k*-lea, dar în acest moment nu mai doresc să primesc alții. Această decuplare (care de fapt reprezintă o fereastră de dimensiune variabilă) oferă mai multă flexibilitate. O vom studia detaliat mai jos.

Este de asemenea prevăzută o *Sumă de control,* în scopul obținerii unei fiabilități extreme. Această sumă de control este calculată pentru antet, informație și pseudo-antetul conceptual prezentat în fig. 6-30. În momentul calculului, *Suma de control* TCP este poziționată pe zero, iar câmpul de date este completat cu un octet suplimentar nul, dacă lungimea sa este un număr impar. Algoritmul de calcul al sumei de control este simplu, el adunând toate cuvintele de 16 biți în complement față de 1 și aplicând apoi încă o dată complementul față de 1 asupra sumei. În acest mod, atunci când receptorul aplică același calcul asupra întregului segment, inclusiv asupra *Sumei de control*, rezultatul ar trebui să fie 0.

Pseudo-antetul conține adresele IP ale mașinii sursă și destinație, de 32 de biți fiecare, numărul de protocol pentru TCP (6) și numărul de octeți al segmentului TCP (incluzând și antetul). Prin includerea pseudo-antetului în calculul sumei de control TCP se pot detecta pachetele care au fost

preluate eronat, dar procedând astfel, este negată însăși ierarhia protocolului, deoarece adresa IP aparține nivelului IP și nu nivelului TCP.

Câmpul *Opțiuni* a fost proiectat pentru a permite adăugarea unor facilități suplimentare neacoperite de antetul obișnuit. Cea mai importantă opțiune este aceea care permite fiecărei mașini să specifice încărcarea maximă de informație utilă TCP pe care este dispusă să o accepte. Utilizarea segmentelor de dimensiune mare este mai eficientă decât utilizarea segmentelor de dimensiune mică datorită amortizării antetului de 20 de octeți prin cantitatea mai mare de informație utilă. Cu toate acestea, este posibil ca mașini mai puțin performante să nu fie capabile să manevreze segmente foarte mari. În timpul inițializării conexiunii, fiecare parte anunță dimensiunea maximă acceptată și așteaptă de la partener aceeași informație. Câștigă cel mai mic dintre cele două numere. Dacă o mașină nu folosește această opțiune, cantitatea implicită de informație utilă este de 536 octeți. Toate mașinile din Internet trebuie să accepte segmente de dimensiune 536 + 20 = 556 octeți. Dimensiunea maximă a segmentului nu trebuie să fie aceeași în cele două direcții.

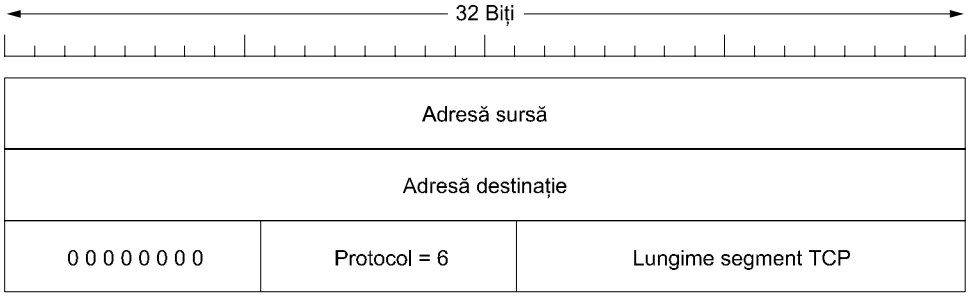


Fig. 6-30. Pseudo-antetul inclus în suma de control TCP.

O fereastră de 64 K octeți reprezintă adesea o problemă pentru liniile cu o lărgime de bandă mare și/sau cu întârzieri mari. Pe o linie T3 (44.736 Mbps) trimiterea unei ferestre întregi de 64 K octeți durează doar 12 ms. Dacă întârzierea propagării dus-întors este de 50 ms (care este valoarea tipică pentru o linie trans-continentală), emițătorul va aștepta confirmări - fiind deci inactiv ¾ din timp. Pe o conexiune prin satelit, situația este chiar mai rea. O fereastră de dimensiune mare ar permite emițătorului să continue trimiterea informației, însă o astfel de dimensiune nu poate fi reprezentată în cei 16 biți ai câmpului Fereastră. În RFC 1323 se propune o opțiune Scală a ferestrei, permițând emițătorului și receptorului să negocieze un factor de scalare a ferestrei. Acest număr permite ambelor părți să deplaseze câmpul Fereastră cu până la 14 biți spre stânga, permițând astfel ferestre de până la 230 octeți. Această opțiune este suportată în prezent de cele mai multe implementări ale TCP-ului.

O altă opțiune propusă de RFC 1106, și care este în prezent implementată pe scară largă, constă în utilizarea unei repetări selective în locul unui protocol cu întoarcere de n pași (eng.: go back n protocol). Dacă receptorul primește un segment eronat urmat de un număr mare de segmente corecte, protocolul TCP clasic va constata într-un final o depășire de timp și va retrimite toate segmentele neconfirmate, deci și pe acelea care au fost recepționate corect (adică se face o întoarcere de n pași). RFC 1106 introduce NAK-urile pentru a permite receptorului să ceară un anumit segment (sau segmente). După obținerea acestora, el poate confirma toată informația memorată reducând astfel cantitatea de informație retransmisă.

6.5.5 Stabilirea conexiunii TCP

În TCP conexiunile sunt stabilite utilizând „înțelegerea în trei pași", discutată în Sec. 6.2.2. Pentru a stabili o conexiune, una din părți - să spunem serverul - așteaptă în mod pasiv o cerere de conexiune prin execuția primitivelor LISTEN și ACCEPT, putând specifica o sursă anume sau nici o sursă în mod particular.

Cealaltă parte - să spunem clientul - execută o primitivă CONNECT, indicând adresa IP și numărul de port la care dorește să se conecteze, dimensiunea maximă a segmentului TCP pe care este dispusă să o accepte și, opțional, o informație utilizator (de exemplu o parolă). Primitiva CONNECT trimite un segment TCP având bitul *SYN* poziționat și bitul *ACK* nepoziționat, după care așteaptă un răspuns.

Atunci când sosește la destinație un segment, entitatea TCP receptoare verifică dacă nu cumva există un proces care a executat LISTEN pe numărul de port specificat în câmpul *Port destinație*. În caz contrar, trimite un răspuns cu bitul *RST* poziționat, pentru a refuza conexiunea.

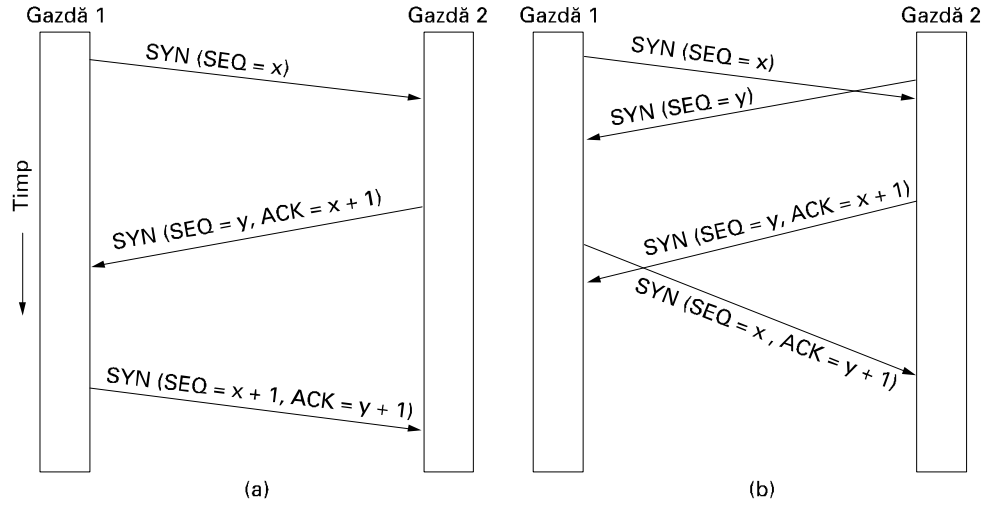


Fig. 6-31. (a) Stabilirea unei conexiuni TCP în cazul normal. (b) Coliziunea apelurilor.

Dacă există vreun proces care ascultă la acel port, segmentul TCP recepționat va fi dirijat către procesul respectiv. Acesta poate accepta sau refuza conexiunea. Dacă o acceptă, trimite înapoi expeditorului un segment de confirmare. În fig. 6-31(a) este reprezentată secvența de segmente TCP transferate în caz de funcționare normală. De notat că un segment *SYN* consumă un octet din spațiul numerelor de secvență, astfel încât confirmarea să poată fi făcută fără ambiguități.

Secvența de evenimente ilustrată în fig. 6-31(b) reprezintă cazul în care două mașini încearcă simultan să stabilească o conexiune între aceleași două porturi. Rezultă că este suficient să fie stabilită o singură conexiune și nu două, deoarece conexiunile sunt identificate prin punctele lor terminale. Dacă prima inițializare conduce la crearea unei conexiuni identificată prin *(x, y)* și același lucru îl face și cea de-a doua inițializare, atunci este construită o singură intrare de tabel, în speță pentru *(x, y)*.

Numărul inițial de secvență asociat unei conexiuni nu este 0, din motivele discutate anterior. Se utilizează o schemă bazată pe un ceas cu o bătaie la fiecare 4 μs. Pentru mai multă siguranță , atunci când

o mașină se defectează, este posibil ca ea să nu fie reinițializată în timpul de viață maxim al unui pachet, garantându-se astfel că pachetele unei conexiuni anterioare nu se plimbă încă pe undeva prin Internet.

6.5.6 Eliberarea conexiunii TCP

Deși conexiunile TCP sunt bidirecționale, pentru a înțelege cum sunt desființate conexiunile, cel mai bine este să ni le imaginăm sub forma unei perechi de legături unidirecționale. Fiecare legătură unidirecțională este eliberată independent de perechea sa. Pentru eliberarea unei conexiuni, orice partener poate expedia un segment TCP având bitul *FIN* setat, lucru care indică faptul că nici o informație nu mai urmează să fie transmisă. Atunci când *FIN*-ul este confirmat, sensul respectiv de comunicare este efectiv oprit pentru noi date. Cu toate acestea, informația poate fi transferată în continuare, pentru un timp nedefinit, în celălalt sens. Conexiunea este desființată atunci când ambele direcții au fost oprite. În mod normal, pentru a elibera o conexiune sunt necesare patru segmente TCP: câte un *FIN* și un *ACK* pentru fiecare sens. Cu toate acestea, este posibil ca primul *ACK* și cel de-al doilea *FIN* să fie cuprinse în același segment reducând astfel numărul total la trei.

La fel ca în conversațiile telefonice, în care ambele persoane pot spune „la revedere” și pot închide telefonul simultan, ambele capete ale unei conexiuni TCP pot expedia segmente *FIN* în același timp. Acestea sunt confirmate ca de obicei, conexiunea fiind astfel eliberată. Nu există de fapt nici o diferență esențială între cazurile în care mașinile eliberează conexiunea secvențial respectiv simultan.

Pentru a evita problema celor două armate, sunt utilizate cronometre. Dacă un răspuns la un *FIN* nu este recepționat pe durata a cel mult două cicluri de maxime de viață ale unui pachet, emițătorul *FIN*-ului eliberează conexiunea. Cealaltă parte va observa în final că nimeni nu mai pare să asculte la celălalt capăt al conexiunii, și va elibera conexiunea în urma expirării unui interval de timp. Această soluție nu este perfectă, dar având în vedere faptul că o soluție perfectă este teoretic imposibilă, va trebui să ne mulțumim cu ce avem. În realitate astfel de probleme apar foarte rar.

6.5.7 Modelarea administrării conexiunii TCP

Pașii necesari stabilirii unei conexiuni pot fi reprezentați printr-un automat cu stări finite, cele 11 stări ale acestuia fiind prezentate în fig. 6-32. În fiecare stare pot apărea doar anumite evenimente. Atunci când are loc un astfel de eveniment, este îndeplinită o acțiune specifică. Atunci când se produce un eveniment a cărui apariție nu este legală în starea curentă, este semnalată o eroare.

Stare	Descriere
CLOSED (ÎNCHIS)	Nici o conexiune nu este activă sau în așteptare
LISTEN (ASCULTARE)	Serverul așteaptă recepționarea unui apel
SYN RCVD (Recepție SYN)	S-a recepționat o cerere de conexiune; aștept ACK
SYN SENT (Transmisie SYN)	Aplicația a început deschiderea unei conexiuni
ESTABLISHED (STABILIT)	Starea normală de transfer a datelor
FIN WAIT 1 (Așteptare FIN 1)	Aplicația a anunțat că termină
FIN WAIT 2 (Așteptare FIN 2)	Partenerul este de acord cu eliberarea conexiunii
TIMED WAIT (Așteptare Temporizată)	Se așteaptă „moartea” tuturor pachetelor
CLOSING (În curs de INCHIDERE)	Ambele părți încearcă simultan închiderea
CLOSE WAIT (ÎNCHIDE și AȘTEAPTĂ)	Partenerul a inițiat eliberarea conexiunii
LAST ACK (CONFIRMARE FINALĂ)	Se așteaptă „moartea” tuturor pachetelor

Fig. 6-32. Stările utilizate în automatul cu stări finite pentru controlul conexiunii TCP.

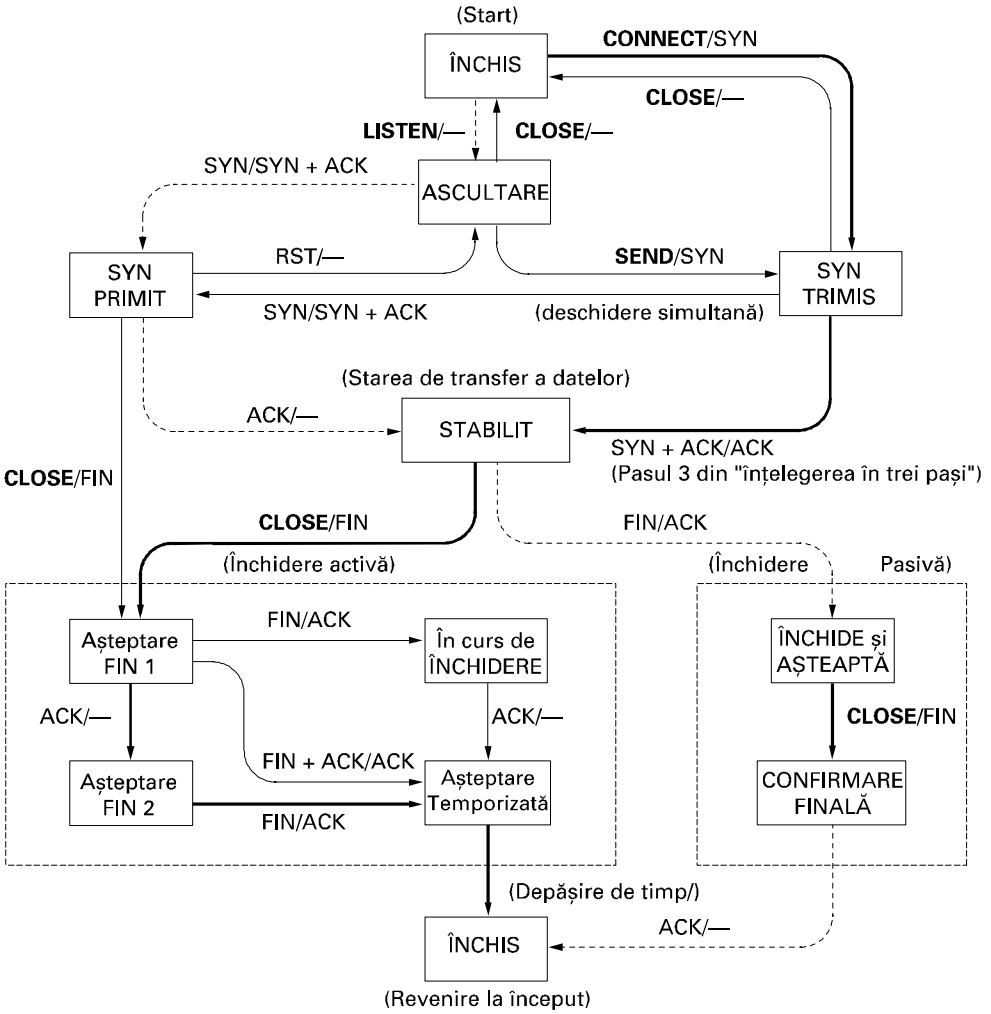


Fig. 6-33. Automatul cu stări finite pentru controlul conexiunii TCP. Linia groasă continuă este calea normală pentru client. Linia groasă întreruptă este calea normală pentru server. Liniile subțiri sunt evenimente neuzuale. Fiecare tranziție este etichetată de evenimentul care a creat-o și acțiunea care rezultă din el, separate de slash.

Fiecare conexiune începe în starea *ÎNCHIS*. Această stare este părăsită dacă urmează să se stabilească o conexiune pasivă (LISTEN) sau activă (CONNECT). Dacă partenerul stabilește o conexiune de tipul opus, starea devine *STABILIT*. Desființarea conexiunii poate fi inițiată de oricare din parteneri, o dată cu eliberarea conexiunii revenindu-se în starea *ÎNCHIS*.

Automatul cu stări finite este reprezentat în fig. 6-33. Cazul cel mai comun, al unui client conectându-se activ la un server pasiv, este reprezentat prin linii groase - continue pentru client și întrerupte pentru server. Liniile subțiri reprezintă secvențe de evenimente mai puțin obișnuite, dar posibile. Fiecare linie din fig. 6-33 este etichetată cu o pereche *eveniment/acțiune*. Evenimentul poate fi unul inițiat de către utilizator printr-un apel sistem (CONNECT, LISTEN, SEND sau CLOSE), recepțio-

narea unui segment (*SYN, FIN, ACK* sau *RST*) sau, într-un singur caz, expirarea unui interval de timp egal cu dublul ciclului de viață a unui pachet. Acțiunea constă în expedierea unui segment de control (*SYN, FIN* sau *RST*) sau „nici o acțiune", lucru reprezentat prin —. Comentariile sunt incluse între paranteze.

Diagrama poate fi înțeleasă cel mai bine urmărind de la bun început calea urmată de un client (linia groasă continuă) și apoi calea urmată de un server (linia groasă întreruptă). Atunci când un program aplicație de pe mașina client generează o cerere CONNECT, entitatea TCP locală creează o înregistrare de conexiune, o marchează ca fiind în starea *SYN SENT* și trimite un segment *SYN*. De observat că mai multe conexiuni pot fi deschise (sau în curs de a fi deschise) în același timp spre folosul mai multor aplicații, astfel încât o stare este asociată unei conexiuni și este înregistrată în înregistrarea asociată acesteia. La recepția unui *SYN + ACK*, TCP expediază ultima confirmare (*ACK*) din „înțelegerea în trei pași" și comută în starea *STABILIT*. Din acest moment, informația poate fi atât expediată cât și recepționată.

Atunci când se termină o aplicație, se apelează primitiva CLOSE care impune entității TCP locale expedierea unui segment *FIN* și așteptarea *ACK*-ului corespunzător (dreptunghiul figurat cu linie întreruptă și etichetat „închidere activă"). Atunci când *ACK*-ul este recepționat, se trece în starea *AȘTEPTARE FIN 2,* unul din sensuri fiind în acest moment închis. Atunci când celălalt sens este la rândul său închis de partenerul de conexiune, se recepționează un *FIN* care este totodată și confirmat. În acest moment, ambele sensuri sunt închise, dar TCP-ul așteaptă un interval de timp egal cu dublul duratei de viață a unui pachet, garantând astfel că toate pachetele acestei conexiuni au murit și că nici o confirmare nu a fost pierdută. Odată ce acest interval de timp expiră, TCP-ul șterge înregistrarea asociată conexiunii.

Să examinăm acum gestiunea conexiunii din punctul de vedere al server-ului. Acesta execută LISTEN și se „așează" fiind totodată atent pentru a vedea cine „se ridică în picioare". La recepționarea unui *SYN*, acesta este confirmat și serverul comută în starea *SYN RCVD*. Atunci când *SYN*-ul server-ului este la rândul său confirmat, „înțelegerea în trei pași" este completă, serverul comutând în starea *STABILIT*. De acum, transferul informației poate începe.

Atunci când clientul a terminat, execută CLOSE, ceea ce conduce la atenționarea server-ului prin recepționarea unui *FIN* (dreptunghiul figurat cu linie întreruptă și etichetat „închidere pasivă"). Atunci când și acesta execută un CLOSE, se trimite un *FIN* către client. O dată cu primirea confirmării clientului, serverul desființează conexiunea și șterge înregistrarea asociată.

6.5.8 Politica TCP de transmisie a datelor

Cum s-a menționat anterior, administrarea ferestrei în TCP nu este direct legată de confirmări, așa cum se întâmplă la cele mai multe protocoale de nivel legătură de date. De exemplu, să presupunem că receptorul are un tampon de 4096 octeți, așa cum se vede în fig. 6-34. Dacă emițătorul transmite un segment de 2048 de octeți care este recepționat corect, receptorul va confirma segmentul. Deoarece acum tamponul acestuia din urmă mai are liberi doar 2048 octeți (până când aplicația șterge niște date din acest tampon), receptorul va anunța o fereastră de 2048 octeți începând de la următorul octet așteptat.

Acum, emițătorul transmite alți 2048 octeți, care sunt confirmați, dar fereastra oferită este 0. Emițătorul trebuie să se oprească până când procesul aplicație de pe mașina receptoare a șters niște date din tampon, moment în care TCP poate oferi o fereastră mai mare.

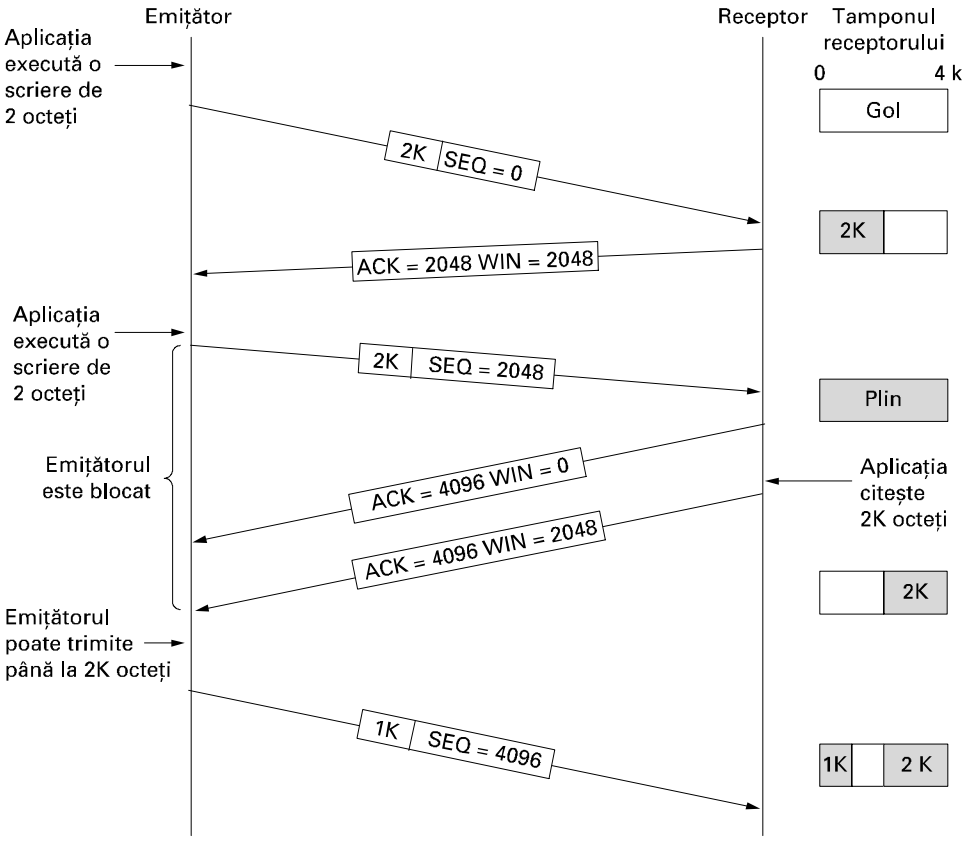


Fig. 6-34. Controlul ferestrei în TCP.

Atunci când fereastra este 0, în mod normal emițătorul nu poate să transmită segmente, cu două excepții. În primul rând, informația urgentă poate fi trimisă, de exemplu pentru a permite utilizatorului să oprească procesele rulând pe mașina de la distanță. În al doilea rând, emițătorul poate trimite un segment de un octet pentru a determina receptorul să renunțe următorul octet așteptat și dimensiunea ferestrei. Standardul TCP prevede în mod explicit această opțiune pentru a preveni interblocarea în cazul în care se întâmplă ca anunțarea unei ferestre să fie vreodată pierdută.

Emițătorii nu transmit în mod obligatoriu date de îndată ce acest lucru este cerut de către aplicație. Nici receptorii nu trimit în mod obligatoriu confirmările de îndată ce acest lucru este posibil. De exemplu, în fig. 6-34, atunci când sunt disponibili primii 2K octeți, TCP, știind că dispune de o fereastră de 4K octeți, va memora informația în tampon până când alți 2K octeți devin disponibili și astfel se va putea transmite un segment cu o încărcare utilă de 4K octeți. Această facilitate poate fi folosită pentru îmbunătățirea performanțelor.

Să considerăm o conexiune TELNET cu un editor interactiv care reacționează la fiecare apăsare a tastelor. În cel mai rău caz, atunci când un caracter sosește la entitatea TCP emițătoare, TCP creează un segment TCP de 21 octeți, pe care îl furnizează IP-ului pentru a fi transmis ca o datagramă IP de 41 octeți. De partea receptorului, TCP transmite imediat o confirmare de 40 octeți (20 octeți antet TCP și 20 octeți antet IP). Mai târziu, când editorul a citit caracterul, TCP transmite o

actualizare a ferestrei, deplasând fereastra cu un octet la dreapta. Acest pachet este de asemenea de 40 octeți. În final, când editorul a prelucrat caracterul, transmite ecoul sub forma unui pachet de 41 octeți. Cu totul, sunt folosiți 162 octeți din lărgimea de bandă și sunt trimise patru segmente pentru orice caracter tipărit. Atunci când lărgimea de bandă este redusă, această metodă de lucru nu este recomandată.

O abordare folosită de multe implementări TCP pentru optimizarea acestei situații constă în întârzierea confirmărilor și actualizărilor de fereastră timp de 500 ms, în speranța apariției unor informații la care să se atașeze pentru o călătorie pe gratis. Presupunând că editorul are un ecou de 50 ms, este necesar acum un singur pachet de 41 octeți pentru a fi trimis utilizatorului de la distanță, reducând numărul pachetelor și utilizarea lărgimii de bandă la jumătate.

Deși această regulă reduce încărcarea rețelei de către receptor, emițătorul operează încă ineficient trimițând pachete de 41 octeți care conțin un singur octet de date. O modalitate de a reduce această deficiență este cunoscută ca **algoritmul lui Nagle** (Nagle, 1984). Sugestia lui Nagle este simplă: atunci când emițătorul dispune de date, în secvență, de câte un octet, el va trimite doar primul octet, restul octeților fiind memorați până la confirmarea primului octet. Apoi vor fi trimise toate caracterele memorate într-un segment TCP și va continua memorarea până la confirmarea tuturor octeților. Dacă utilizatorul tastează repede și rețeaua este lentă, un număr substanțial de caractere poate fi plasat în fiecare segment, reducând cu mult lărgimea de bandă folosită. În plus, algoritmul permite transmisia unui nou pachet, dacă s-a dispus de suficientă informație pentru a umple jumătate dintr-o fereastră sau pentru a completa un segment.

Implementările TCP folosesc pe scară largă algoritmul lui Nagle, dar există situații când este mai bine ca el să fie dezactivat. În particular, când o aplicație X-Windows rulează prin Internet, deplasările mausului trebuie transmise mașinii de la distanță. (Sistemul X Window este sistemul de ferestre utilizat pe majoritatea sistemelor UNIX.) Gruparea lor pentru a fi transmise în rafală provoacă o mișcare imprevizibilă a cursorului, lucru care nemulțumește profund utilizatorii.

O altă problemă care poate degrada performanța TCP este **sindromul ferestrei stupide**. (Clark, 1982). Această problemă apare atunci când informația este furnizată entității TCP emițătoare în blocuri mari, dar la partea receptoare o aplicație interactivă citește datele octet cu octet. Pentru a înțelege problema, să analizăm fig. 6-35. Inițial, tamponul TCP al receptorului este plin și emițătorul știe acest fapt (adică are o fereastră de dimensiune 0). Apoi, aplicația interactivă citește un caracter de pe canalul TCP. Această acțiune face fericită entitatea TCP receptoare, deci ea va trimite o actualizare de fereastră către emițător dându-i astfel dreptul de a mai trimite un octet. Îndatorat, emițătorul trimite un octet. Cu acesta, tamponul este plin și receptorul confirmă segmentul de 1 octet, dar repoziționează dimensiunea ferestrei la 0. Acest comportament poate continua la nesfârșit.

Soluția lui Clark este de a nu permite receptorului să trimită o actualizare de fereastră la fiecare octet. În schimb, el este forțat să aștepte până când spațiul disponibil are o dimensiune decentă, urmând să-l ofere pe acesta din urmă. Mai precis, receptorul nu ar trebui să trimită o actualizare de fereastră până când nu va putea gestiona minimul dintre dimensiunea maximă oferită atunci când conexiunea a fost stabilită și jumătate din dimensiunea tamponului său, dacă este liberă.

Mai mult decât atât, emițătorul poate îmbunătăți situația netrimițând segmente de dimensiune mică. În schimb, el ar trebui să încerce să aștepte până când acumulează suficient spațiu în fereastră pentru a trimite un segment întreg sau măcar unul conținând cel puțin jumătate din dimensiunea tamponului receptorului (pe care trebuie să o estimeze din secvența actualizărilor de fereastră recepționate până acum).

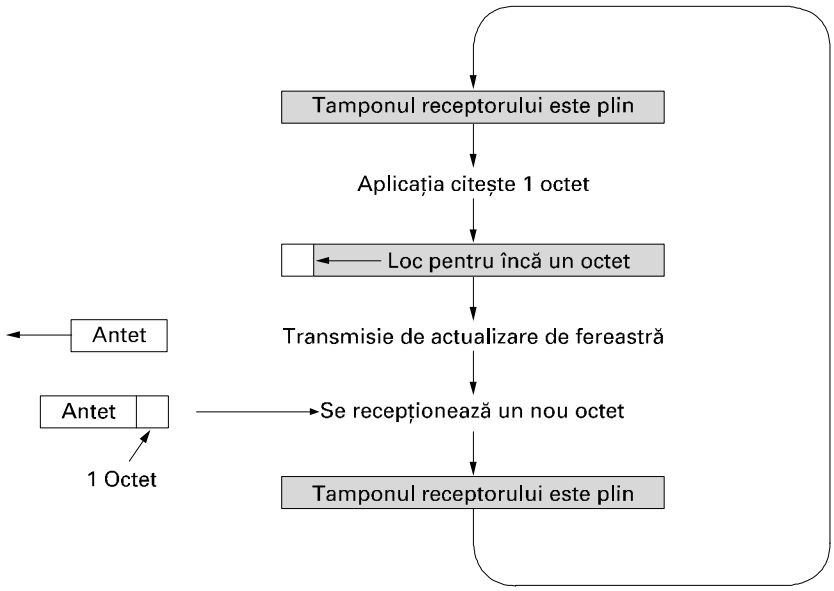


Fig. 6-35. Sindromul ferestrei stupide.

Algoritmul lui Nagle și soluția lui Clark pentru sindromul ferestrei stupide sunt complementare. Nagle a încercat să rezolve problema furnizării datelor către TCP octet cu octet, cauzată de aplicația emițătoare. Clark a încercat să rezolve problema extragerii datelor de la TCP octet cu octet, cauzată de către aplicația receptoare. Ambele soluții sunt valide și pot funcționa împreună. Scopul este ca emițătorul să nu trimită segmente mici, iar receptorul să nu ceară astfel de segmente.

Receptorul TCP poate face, pentru îmbunătățirea performanțelor, mai mult decât simpla actualizare a ferestrei în unități mari. Ca și emițătorul TCP, el are posibilitatea să memoreze date, astfel încât să poată bloca o cerere de READ a aplicației până când îi poate furniza o cantitate semnificativă de informație. Astfel se reduce numărul de apeluri TCP, deci și supraîncărcarea. Bineînțeles, în acest mod va crește și timpul de răspuns, dar pentru aplicații care nu sunt interactive, așa cum este transferul de fișiere, eficiența poate fi mai importantă decât timpul de răspuns la cereri individuale.

O altă problemă de discutat despre receptor se referă la ce trebuie să facă acesta cu segmentele care nu sosesc în ordine. Ele pot fi reținute sau eliminate, după cum dorește receptorul. Bineînțeles, confirmările pot fi trimise numai atunci când toată informația până la octetul confirmat a fost recepționată. Dacă receptorul primește segmentele 0, 1, 2, 4, 5, 6 și 7, el poate confirma totul până la ultimul octet din segmentul 2 inclusiv. Atunci când emițătorul constată o depășire de timp, el va retransmite segmentul 3. Dacă receptorul a memorat în tampon segmentele 4 până la 7, odată cu recepția segmentului 3 el poate confirma toți octeții până la sfârșitul segmentului 7.

6.5.9 Controlul congestiei în TCP

Atunci când încărcarea la care este supusă o rețea este mai mare decât poate aceasta să suporte, apare congestia. Internet-ul nu face excepție. În această secțiune, vom discuta algoritmi care se ocupă cu astfel de congestii și care au fost dezvoltați pe parcursul ultimului sfert de secol. Deși nivelul rețea

încearcă de asemenea să controleze congestia, cea mai mare parte a muncii este făcută de TCP, și aceasta deoarece adevărata soluție a congestiei constă în micșorarea ratei de transfer a informației.

Teoretic, congestia poate fi controlată pe baza unui principiu împrumutat din fizică: legea conservării pachetelor. Ideea de bază este de a nu injecta un nou pachet în rețea până când un pachet mai vechi nu o părăsește (de exemplu este furnizat receptorului). TCP încearcă să atingă acest scop prin manipularea dinamică a dimensiunii ferestrei.

Primul pas în controlul congestiei este detecția ei. Mai demult, detecția congestiei era dificilă. O depășire de timp datorată pierderii unui pachet putea fi cauzată fie de (1) zgomotul de pe linia de transmisie, fie de (2) eliminarea pachetului de către un ruter congestionat. Diferențierea celor două cazuri era dificilă.

În zilele noastre, pierderea pachetului din pricina erorilor de transmisie este destul de rară, deoarece cele mai multe din trunchiurile principale de comunicație sunt din fibră (deși rețelele fără fir sunt un subiect separat). În consecință, cele mai multe depășiri ale timpilor de transmisie pe Internet se datorează congestiilor. Toți algoritmii TCP din Internet presupun că depășirile de timp sunt cauzate de congestii și monitorizează aceste depășiri pentru a detecta problemele.

Înainte de a discuta despre modalitatea în care TCP reacționează la congestii, să descriem în primul rând modul în care se încearcă prevenirea apariției lor. Atunci când se stabilește o conexiune, trebuie să se aleagă o fereastră de o dimensiune potrivită. Receptorul poate specifica o fereastră bazându-se pe dimensiunea tamponului propriu. Dacă emițătorul acceptă această dimensiune a ferestrei, nu mai pot apărea probleme datorită depășirii tamponului la recepție, dar pot apărea în schimb datorită congestiei interne în rețea.

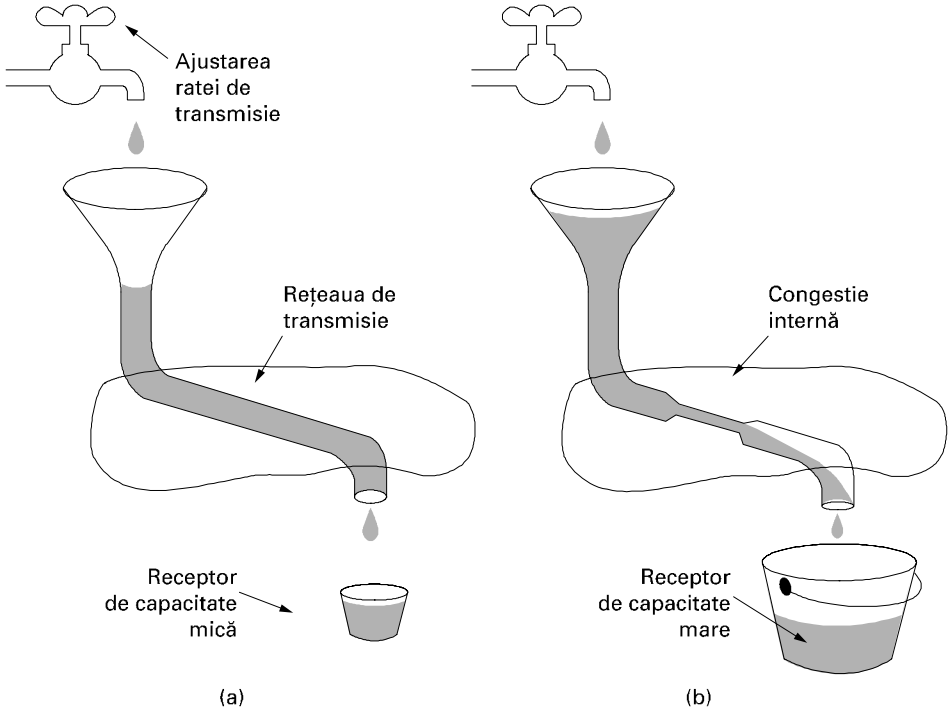


Fig. 6-36. (a) O rețea rapidă alimentând un rezervor de capacitate mică. (b) O rețea lentă alimentând un receptor de mare capacitate.

În fig. 6-36, putem vedea interpretarea hidraulică a acestei probleme. În fig. 6-36(a), observăm o conductă groasă care duce la un receptor de dimensiune mică. Atâta timp cât emițătorul nu trimite mai multă apă decât poate conține găleata, apa nu se va pierde. În fig. 6-36(b), factorul de limitare nu mai este capacitatea găleții, ci capacitatea de transport a rețelei. Dacă vine prea repede prea multă apă, ea se va revărsa și o anumită cantitate se va pierde (în acest caz prin umplerea pâlniei).

Soluția din Internet este de a realiza posibilitatea existenței a două probleme - capacitatea rețelei și capacitatea receptorului - și de a le trata pe fiecare separat. Pentru a face acest lucru, fiecare emițător menține două ferestre: fereastra acceptată de către receptor și o a doua fereastră, **fereastra de congestie**. Fiecare reflectă numărul de octeți care pot fi trimiși de către emițător. Numărul octeților care pot fi trimiși este dat de minimul dintre cele două ferestre. Astfel, fereastra efectivă este minimul dintre ceea ce emițătorul crede că este „în regulă" și ceea ce receptorul crede că este „în regulă". Dacă receptorul spune: „Trimite 8K octeți", dar emițătorul știe că o rafală de mai mult de 4K octeți poate aglomera excesiv rețeaua, el va trimite 4K octeți. Din alt punct de vedere, dacă receptorul spune: „Trimite 8K octeți" și emițătorul știe că o rafală de 32K octeți poate străbate fără efort rețeaua, el va trimite toți cei 8K octeți ceruți.

La stabilirea conexiunii, emițătorul inițializează fereastra de congestie la dimensiunea celui mai mare segment utilizat de acea conexiune. El trimite apoi un segment de dimensiune maximă. Dacă acest segment este confirmat înaintea expirării timpului, mai adaugă un segment la fereastra de congestie, făcând-o astfel de dimensiunea a două segmente de dimensiune maximă, și trimite două segmente. O dată cu confirmarea fiecăruia din aceste segmente, fereastra de congestie este redimensionată cu încă un segment de dimensiune maximă. Atunci când fereastra de congestie este de n segmente, dacă toate cele n segmente sunt confirmate în timp util, ea este crescută cu numărul de octeți corespunzător celor n segmente. De fapt, fiecare rafală confirmată cu succes dublează fereastra de congestie.

Fereastra de congestie crește în continuare exponențial până când sau se produce o depășire de timp, sau se atinge dimensiunea ferestrei receptorului. Ideea este ca dacă rafale de dimensiune, să spunem, 1024, 2048 și 4096 de octeți funcționează fără probleme, dar o rafală de 8192 octeți duce la o depășire de timp, fereastra de congestie va fi stabilită la 4096 de octeți pentru a evita congestia. Atâta timp cât fereastra de congestie rămâne la 4096, nu va fi transmisă nici o rafală mai mare de această valoare, indiferent cât de mult spațiu de fereastră este oferit de către receptor. Acest algoritm este numit **algoritmul startului lent**, fără a fi însă câtuși de puțin lent (Jacobson, 1988). Este exponențial. Toate implementările TCP trebuie să îl suporte.

Să privim acum algoritmul de control al congestiei în cazul Internetului. El utilizează în plus față de ferestrele de recepție și de congestie un al treilea parametru, numit **prag**, inițial de 64K. Atunci când apare o depășire de timp, pragul este poziționat la jumătate din fereastra curentă de congestie și fereastra de congestie este repoziționată la dimensiunea unui segment maxim. Startul lent este utilizat apoi pentru a determina cât poate rețeaua să ducă, atâta doar că acea creștere exponențială se oprește odată cu atingerea pragului. De aici înainte transmisiile reușite măresc în mod liniar dimensiunea ferestrei de congestie (cu câte un segment maxim pentru fiecare rafală), în locul unei creșteri pentru fiecare segment. De fapt, algoritmul presupune că este acceptabilă înjumătățirea ferestrei de congestie, din acel punct continuându-și gradual calea spre dimensiuni mai mari.

Funcționarea algoritmului de congestie se poate vedea în fig. 6-37. Dimensiunea unui segment maxim este, în acest caz, de 1024 de octeți. Inițial fereastra de congestie are 64K octeți, dar apare o depășire de timp și deci pragul este stabilit la 32K octeți iar fereastra de congestie la 1K octeți acesta fiind punctul 0 al transmisiei din figură. Fereastra de congestie crește apoi exponențial până atinge pragul (32K octeți). Începând de aici, creșterea este liniară.

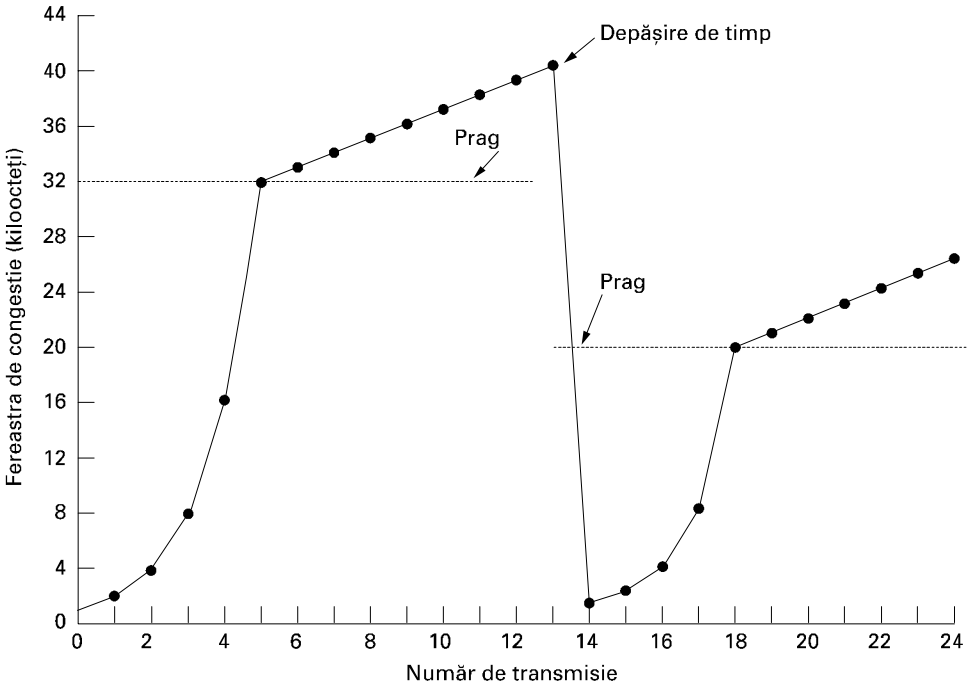


Fig. 6-37. Un exemplu al algoritmului de congestie din Internet.

Transmisia 13 nu are noroc (este de la sine înțeles) și apare o depășire de timp. Pragul este stabilit la jumătate din fereastra curentă (acum 40K octeți, deci jumătate este 20K octeți) și startul lent este inițiat din nou. Atunci când confirmările pentru transmisia 14 încep să sosească, primele patru dublează fiecare fereastra de congestie, dar după aceea creșterea redevine liniară.

Dacă nu mai apar depășiri de timp, fereastra de congestie va continua să crească până la dimensiunea ferestrei receptorului. În acest punct, creșterea ei va fi oprită și va rămâne constantă atâta timp cât nu mai apar depășiri și fereastra receptorului nu își modifică dimensiunea. Ca un alt aspect, dacă un pachet ICMP SOURCE QUENCH sosește și este furnizat TCP-ului, acest eveniment este tratat la fel ca o depășire de timp. O alternativă (și o abordare mai recentă) este descrisă în RFC 3168.

6.5.10 Administrarea contorului de timp în TCP

TCP utilizează (cel puțin conceptual) mai multe contoare pentru a face ceea ce are de făcut. Cel mai important dintre acestea este **contorul de retransmisie**. Atunci când este trimis un segment, se pornește un contor de retransmisie. Dacă segmentul este confirmat înainte de expirarea timpului, contorul este oprit. Pe de altă parte, dacă timpul expiră înaintea primirii confirmării, segmentul este retransmis (și contorul este pornit din nou). Întrebarea care se pune este următoarea: Cât de mare trebuie să fie intervalul de timp până la expirare?

Această problemă este mult mai dificilă la nivelul transport din Internet decât la nivelul protocoalelor generice de legătură de date prezentate în Cap. 3. În cazul din urmă, întârzierea așteptată este ușor predictibilă (de exemplu, are o variantă scăzută), deci contorul poate fi setat să expire chiar

imediat după ce era așteptată confirmarea, așa cum se arată în fig. 6-38(a). Cum confirmările sunt rareori întârziate în nivelul legătură de date, absența unei confirmări în momentul în care aceasta era așteptată înseamnă de obicei că s-a pierdut fie cadrul, fie confirmarea.

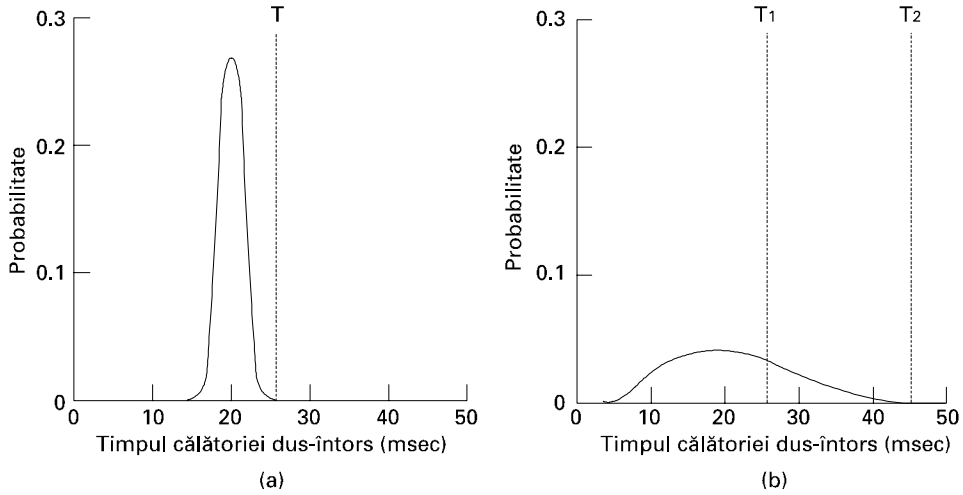


Fig. 6-38. (a) Densitatea de probabilitate a sosirilor în timp a confirmărilor în nivelul legătură de date. (b) Densitatea de probabilitate a sosiri confirmărilor pentru TCP.

TCP trebuie să facă față unui mediu radical diferit. Funcția de densitate a probabilității pentru timpul necesar întoarcerii unei confirmări TCP arată mai degrabă ca în fig. 6-38(b) decât ca în fig. 6-38(a). Este dificilă determinarea timpului în care se realizează circuitul dus-întors până la destinație. Chiar și când acesta este cunoscut, stabilirea intervalului de depășire este de asemenea dificilă. Dacă intervalul este prea scurt, să spunem T_1 în fig. 6-38(b), vor apărea retransmisii inutile, aglomerând Internet-ul cu pachete fără rost. Dacă este prea lung, (T_2), performanțele vor avea de suferit datorită întârzierii retransmisiei de fiecare dată când se pierde un pachet. Mai mult decât atât, media și varianța distribuției sosirii confirmărilor se pot schimba cu rapiditate pe parcursul a câtorva secunde atunci când apare sau se rezolvă o congestie.

Soluția este să se utilizeze un algoritm profund dinamic, care ajustează în mod constant intervalul de depășire bazându-se pe măsurători continue ale performanței rețelei. Algoritmul utilizat în general de către TCP este datorat lui Jacobson (1988) și este descris mai jos. Pentru fiecare conexiune, TCP păstrează o variabilă, *RTT*, care este cea mai bună estimare a timpului în care se parcurge circuitul dus-întors către destinația în discuție. Atunci când este trimis un segment, se pornește un contor de timp, atât pentru a vedea cât de mult durează până la primirea confirmării cât și pentru a iniția o retransmisie în cazul scurgerii unui interval prea lung. Dacă se primește confirmarea înaintea expirării contorului, TCP măsoară cât de mult i-a trebuit confirmării să sosească, fie acest timp *M*. În continuare el actualizează *RTT*, după formula:

$$RTT = \alpha RTT + (1 - \alpha)M$$

unde α este un factor de netezire care determină ponderea dată vechii valori. Uzual, α=7/8.

Chiar presupunând o valoare bună a lui *RTT*, alegerea unui interval potrivit de retransmisie nu este o sarcină ușoară. În mod normal, TCP utilizează βRTT, dar problema constă în alegerea lui β.

În implementările inițiale, β era întotdeauna 2, dar experiența a arătat că o valoare constantă este inflexibilă deoarece nu corespunde în cazul creșterii variației.

În 1988, Jacobson a propus ca β să fie aproximativ proporțional cu deviația standard a funcției de densitate a probabilității timpului de primire a confirmărilor, astfel încât o variație mare implică un β mare și viceversa. În particular, el a sugerat utilizarea *deviației medii* ca o estimare puțin costisitoare a *deviației standard*. Algoritmul său presupune urmărirea unei alte variabile de netezire, D, deviația. La fiecare sosire a unei confirmări, este calculată diferența dintre valorile așteptate și observate $|RTT - M|$. O valoare netezită a acesteia este păstrată în D, prin formula

$$D = \alpha D + (1 - \alpha)|RTT - M|$$

unde α poate sau nu să aibă aceeași valoare ca cea utilizată pentru netezirea lui *RTT*. Deși *D* nu este chiar deviația standard, ea este suficient de bună și Jacobson a arătat cum poate fi calculată utilizând doar adunări de întregi, scăderi și deplasări, ceea ce este un mare punct câștigat. Cele mai multe implementări TCP utilizează acum acest algoritm și stabilesc valoarea intervalului de depășire la:

$$\text{Depășire} = RTT + 4 * D$$

Alegerea factorului 4 este într-un fel arbitrară, dar are două avantaje. În primul rând, multiplicarea prin 4 poate fi făcută printr-o singură deplasare. În al doilea rând, minimizează depășirile de timp și retransmisiile inutile, datorită faptului că mai puțin de un procent din totalul pachetelor sosesc cu întârzieri mai mari de patru ori deviația standard. (De fapt, Jacobson a propus inițial să se folosească 2, dar experiența ulterioară a demonstrat că 4 conduce la performanțe mai bune).

O problemă legată de estimarea dinamică a *RTT* se referă la ce trebuie făcut în situația în care un segment cauzează o depășire și trebuie retransmis. Atunci când confirmarea este primită, nu este clar dacă aceasta se referă la prima transmisie sau la o transmisie următoare. O decizie eronată poate contamina serios estimarea lui *RTT*. Phil Karn a descoperit această problemă cu multă greutate. El este un radio amator entuziast, interesat în transmiterea pachetelor TCP/IP prin operare radio, un mediu recunoscut pentru lipsa de fiabilitate (pe o zi senină, la destinație ajung jumătate din pachete). El a făcut o propunere simplă: să nu se actualizeze *RTT* pentru nici un segment care a fost retransmis. În loc de aceasta, timpul de expirare este dublat la fiecare eșec, până când segmentele ajung prima oară la destinație. Această corecție este numită **algoritmul lui Karn**. Cele mai multe din implementările TCP utilizează acest algoritm.

Contorul de retransmisie nu este singurul utilizat de către TCP. Un al doilea contor este **contorul de persistență**. El este proiectat să prevină următoarea situație de interblocare. Receptorul trimite o confirmare cu o dimensiune a ferestrei 0, spunându-i emițătorului să aștepte. Mai târziu, receptorul actualizează fereastra, dar pachetul cu această actualizare este pierdut. Acum, atât emițătorul cât și receptorul așteaptă o reacție din partea celuilalt. Atunci când contorul de persistență expiră, emițătorul transmite o sondă către receptor. Răspunsul la această investigare furnizează dimensiunea ferestrei. Dacă această dimensiune continuă să fie zero, contorul de persistență este repoziționat și ciclul se repetă. Dacă este nenul, informația poate fi acum transmisă.

Un al treilea contor utilizat de unele implementări este **contorul de menținere în viață**. Când o conexiune a fost inactivă o lungă perioadă de timp, contorul de menținere în viață poate expira pentru a forța una din părți să verifice dacă cealaltă parte există încă. Dacă aceasta nu răspunde, conexiunea este închisă. Această facilitate este controversată, deoarece supraîncarcă rețeaua și poate termina o conexiune altfel sănătoasă datorită unei partiționări tranzitorii a rețelei.

Ultimul contor folosit la fiecare conexiune TCP este acela utilizat în stările de *AȘTEPTARE CONTORIZATĂ* pe parcursul închiderii conexiunilor. El funcționează pe durata a două vieți maxime ale unui pachet, pentru a se asigura că, atunci când o conexiune este închisă, toate pachetele create de aceasta au murit.

6.5.11 TCP și UDP în conexiune fără fir

Teoretic, protocoalele de transport ar trebui să fie independente de tehnologia nivelului rețea. În particular, TCP-ului ar trebui să nu-i pese dacă IP rulează peste o rețea cablu sau radio. În practică, acest lucru contează, deoarece cele mai multe implementări TCP au fost atent optimizate pe baza unor presupuneri care sunt adevărate pentru rețele cu cabluri, dar care nu mai sunt valabile în cazul rețelelor fără fir. Ignorarea proprietăților de transmisie fără fir poate conduce la o implementare TCP corectă din punct de vedere logic, dar cu performanțe incredibil de proaste.

Principala problemă este algoritmul de control al congestiei. Aproape toate implementările TCP din zilele noastre pleacă de la premisa că depășirile de timp sunt cauzate de congestie și nu de pierderea pachetelor. În consecință, atunci când expiră un contor, TCP încetinește ritmul și trimite pachete cu mai puțină vigoare (ex. algoritmul startului lent al lui Jacobson). Ideea din spatele acestei abordări constă în reducerea încărcării rețelei, astfel eliminându-se neplăcerile cauzate de congestie.

Din nefericire, legăturile bazate pe transmisia fără fir nu sunt deloc fiabile. Ele pierd tot timpul pachete. Pentru a controla această pierdere a pachetelor, abordarea corectă este să se retrimită cât mai repede posibil. Încetinirea ritmului nu face decât să înrăutățească lucrurile. Dacă presupunem că, atunci când emițătorul transmite 100 de pachete pe secundă, 20% din totalul pachetelor se pierde, productivitatea este de 80 pachete/sec. Dacă emițătorul încetinește ritmul la 50 pachete/sec, productivitatea scade la 40 pachete/sec.

Atunci când se pierde un pachet pe o rețea cu cabluri, emițătorul ar trebui să încetinească ritmul. Atunci când se pierde un pachet pe o rețea fără fir, emițătorul ar trebui să îl mărească. Dacă emițătorul nu știe despre ce tip de rețea este vorba, luarea unei decizii este dificilă.

În mod frecvent, calea de la emițător la receptor este eterogenă. Primii 1000 km pot să fie într-o rețea cu cabluri, dar ultimul kilometru poate să fie fără fir. Acum, luarea unei decizii în situația unei depășiri de timp este și mai dificilă, dat fiind că intervine și locul în care apare problema. O soluție propusă de Bakne și Badrinath (1995), **TCP indirect**, constă în spargerea conexiunii TCP în două conexiuni separate, ca în fig. 6-39. Prima conexiune pleacă de la emițător la stația de bază. Cea de-a doua leagă stația de bază de receptor. Această stație de bază nu face decât să copieze pachetele din cele două conexiuni în ambele direcții.

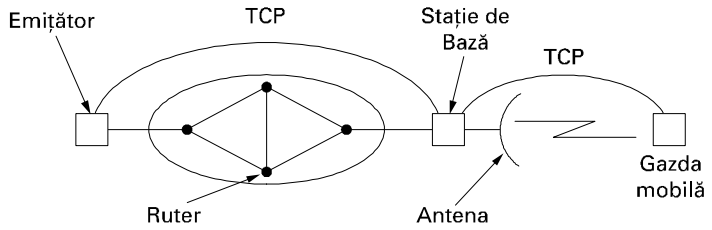


Fig. 6-39. Spargerea conexiunii TCP în două conexiuni.

Avantajul acestei scheme este acela că ambele conexiuni sunt acum omogene. Depășirile de timp din prima conexiune pot încetini emițătorul, în timp ce depășirile de timp din cea de-a doua îl pot accelera. Alți parametri pot fi de asemenea reglați separat în fiecare din cele două conexiuni. Dezavantajul este acela că este negată însăși semantica TCP. Atâta timp cât fiecare parte a conexiunii este o conexiune TCP în sine, stația de bază confirmă fiecare segment TCP în mod obișnuit. Doar că acum, recepția unei confirmări de către emițător nu mai înseamnă că receptorul a primit segmentul, ci doar că el a fost primit de către stația de bază.

O soluție diferită, datorată lui Balakrishnan et. al (1995), nu încalcă semantica TCP. Ea se bazează pe mici modificări făcute în codul nivelului rețea din stația de bază. Una din modificări constă în adăugarea unui agent de supraveghere care observă și interceptează pachetele TCP care pleacă spre gazda mobilă precum și confirmările care se întorc de la acesta. Atunci când observă un segment TCP care pleacă spre gazda mobilă, dar nu observă confirmarea recepționării acestuia într-un interval de timp dat (relativ scurt), agentul ascuns pur și simplu retransmite acel segment, fără a mai spune sursei acest lucru. De asemenea, el retransmite și atunci când observă confirmări duplicate din partea gazdei mobile, lucru care indică invariabil faptul că aceasta a pierdut ceva. Confirmările duplicate sunt eliminate pe loc, pentru a evita ca sursa să le interpreteze ca un semn de congestie.

Cu toate acestea, un dezavantaj al acestei transparențe este acela că, dacă legătura fără fir pierde multe pachete, sursa poate depăși limita de timp în așteptarea unei confirmări și poate invoca în consecință algoritmul de control al congestiei. În cazul TCP-ului indirect, algoritmul de control al congestiei nu va fi niciodată inițiat dacă nu apare într-adevăr o situație de congestie în partea „cablată" a rețelei.

Algoritmul Balakrishnan oferă de asemenea o soluție problemei pierderii segmentelor generate de către gazda mobilă. Atunci când stația de bază constată o pauză în interiorul domeniului numerelor de secvență, aceasta generează o cerere pentru o repetare selectivă a octetului lipsă, utilizând o opțiune TCP.

Utilizând aceste corecturi, legătura fără fir devine mai fiabilă în ambele direcții fără ca sursa să știe acest lucru și fără modificarea semanticii TCP.

Deși UDP-ul nu suferă de aceleași probleme ca și TCP-ul, comunicația fără fir induce și pentru el anumite dificultăți. Principala problemă este aceea că programele utilizează UDP se așteaptă ca acesta să fie foarte fiabil. Ele știu că nu este furnizată nici o garanție, dar cu toate acestea se așteaptă ca el să fie aproape perfect. Într-un mediu fără fir, el va fi însă departe de perfecțiune. Pentru programele care sunt capabile să se refacă după pierderea mesajelor UDP, dar numai cu un cost considerabil, trecerea bruscă de la un mediu în care mesajele puteau fi pierdute mai mult teoretic decât practic la un mediu în care ele sunt pierdute sistematic poate conduce la un dezastru în ceea ce privește performanțele.

Comunicația fără fir afectează și alte domenii decât cel al performanțelor. De exemplu, cum poate o gazdă mobilă să găsească o imprimantă locală la care să se conecteze, în loc să utilizeze propria imprimantă? Oarecum legată de aceasta este și problema obținerii paginii WWW pentru celula locală, chiar dacă numele ei nu este cunoscut. De asemenea, proiectanții paginilor WWW au tendința să presupună disponibilă o mare lărgime de bandă. Punerea unei embleme mari pe fiecare pagină poate să devină contraproductivă dacă transmisia paginii printr-o legătură fără fir lentă va dura 10 secunde, și acest lucru ajunge până la urmă să irite utilizatorii.

Cum rețelele cu comunicații fără fir devin tot mai comune, problema rulării TCP-ului pe ele a devenit tot mai acută. Documentații suplimentare în acest domeniu se găsesc în (Barakat ș.a., 2000; Ghani și Dixit, 1999; Huston, 2001; și Xylomenos ș.a., 2001).

6.5.12 TCP Tranzacțional

Mai devreme în acest capitol am analizat apelul de proceduri la distanță ca modalitate de a implementa sistemele client-server. Dacă atât cererea, cât și răspunsul sunt suficient de mici încât să se potrivească în pachete simple și operația este idempotentă, UDP-ul poate fi ușor utilizat. Totuși, dacă aceste condiții nu sunt îndeplinite, utilizarea UDP-ului este mai puțin atractivă. De exemplu, dacă răspunsul este unul lung, atunci datagramele trebuie sa fie secvențiate și trebuie inițiat un mecanism pentru a retransmite datagramele pierdute. De fapt, aplicației îi este cerut să reinventeze TCP-ul.

În mod cert, acest lucru nu este atractiv, dar nici utilizarea TCP-ului în sine nu este atractivă. Problema este eficiența. Secvența normală a pachetelor pentru a face un RPC peste TCP este prezentată în fig. 6-40(a). În cel mai bun caz sunt necesare nouă pachete.

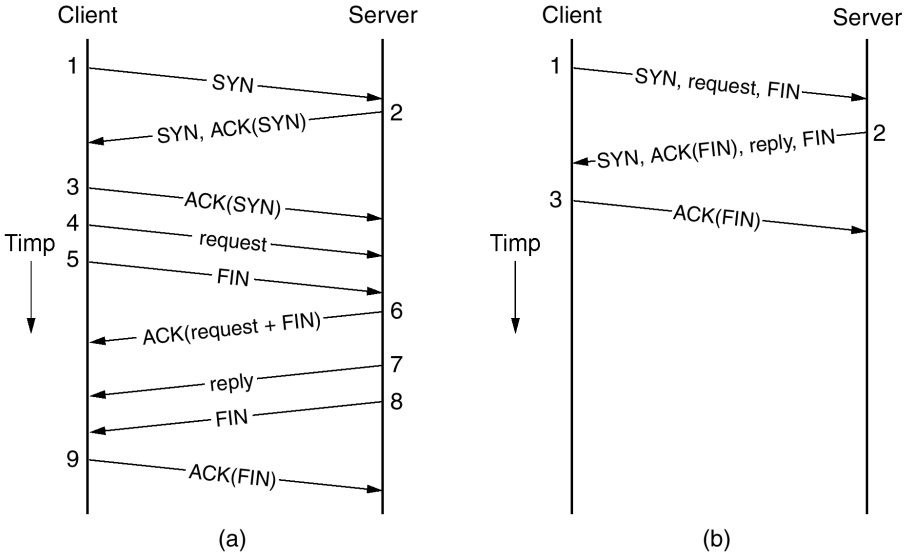


Fig. 6-40. (a) RPC folosind TCP clasic ; (b) RPC folosind T/TCP

Cele nouă pachete sunt după cum urmează:

1. Clientul trimite un pachet SYN pentru a stabili o conexiune.
2. Serverul trimite un pachet ACK pentru a recunoaște pachetul SYN.
3. Clientul finalizează înțelegerea în trei pași.
4. Clientul trimite cererea reală.
5. Clientul trimite un pachet FIN pentru a indica dacă s-a terminat trimiterea.
6. Serverul confirmă cererea și FIN-ul.
7. Serverul trimite răspunsul înapoi clientului.
8. Serverul trimite un pachet FIN pentru a indica că și acest lucru s-a încheiat.
9. Clientul confirmă FIN-ul server-ului.

A se reține că acesta este cazul ideal. În cazul cel mai rău, cererea clientului și FIN-ul sunt confirmate separat, precum sunt răspunsul server-ului și FIN-ul.

Întrebarea care apare imediat este dacă există vreo posibilitate de a combina eficiența RPC-ului folosind UDP (doar 2 mesaje) cu fiabilitatea TCP-ului. Răspunsul este: aproape că da. Se poate

realiza cu o variantă experimentală de TCP numită **T/TCP** (**Transactional TCP**, rom: TCP tranzacțional), care este descrisă în RFC 1379 și 1644.

Ideea principală este aceea de a modifica secvența standard de inițializare a conexiunii astfel încât să permită, la nivel înalt, transferul de date în timpul inițializării. Protocolul T/TCP este prezentat în fig. 6-40(b). Primul pachet al clientului conține bitul SYN, cererea în sine și FIN-ul. De fapt acesta spune: vreau să stabilesc o conexiune, aici sunt datele și am terminat

Când serverul primește cererea, caută sau calculează răspunsul și alege modul în care să răspundă. Dacă răspunsul încape într-un pachet, dă răspunsul din fig. 6-40(b), care spune: confirm FIN-ul tău, iată răspunsul, iar eu am terminat. Clientul confirmă apoi FIN-ul server-ului și protocolul ia sfârșit în (după) trei mesaje.

În orice caz, dacă rezultatul este mai mare de 1 pachet, serverul are de asemenea și opțiunea de a nu seta bitul FIN, caz în care poate trimite pachete multiple înainte de a-i închide direcția.

Merită probabil menționat faptul că T/TCP nu este singura îmbunătățire propusă pentru TCP. O altă propunere este **SCTP** (**Stream Control Transmission Protocol**, rom: Protocolul de control al transmisiei fluxului). Caracteristicile sale includ păstrarea legăturilor dintre mesaje, modalități multiple de livrare (de ex: livrarea neordonată), găzduirea multiplă (destinații de rezervă), și confirmări selective (Stewart and Metz, 2001). În orice caz, oricând cineva propune să schimbe ceva care a funcționat atât de bine de atâta timp, se duce o luptă aprigă între tabăra "utilizatorii doresc mai multe facilități" și cea "dacă nu e stricat, nu-l repara!".

6.6 ELEMENTE DE PERFORMANȚĂ

În rețelele de calculatoare sunt foarte importante elementele de performanță. Atunci când sunt interconectate sute sau mii de calculatoare, au loc adesea interacțiuni complexe cu consecințe nebănuite. Această complexitate conduce în mod frecvent la performanțe slabe fără ca cineva să știe de ce. În secțiunile următoare vom examina mai multe elemente legate de performanța rețelei pentru a identifica tipurile de probleme și ce poate fi făcut pentru rezolvarea lor.

Din nefericire, înțelegerea performanței rețelei este mai degrabă o artă decât o știință. Este prea puțină teorie care stă la bază și de fapt aceasta nu folosește în situații practice. Cel mai bun lucru pe care îl putem face este să indicăm reguli rezultate dintr-o experiență îndelungată și să prezentăm exemple luate din lumea reală. Am amânat în mod intenționat această discuție după studiul nivelului transport din rețelele TCP, pentru a fi capabili să folosim TCP ca exemplu în diverse locuri.

Nivelul transport nu este singurul loc unde apar elemente legate de performanță. Am văzut unele elemente la nivelul rețea, în capitolul precedent. Cu toate acestea, nivelul rețea tinde să fie preocupat în mare măsură de rutare și controlul congestiei. Problemele mai generale, orientate spre sistem ,tind să fie legate de nivelul transport, așa că acest capitol este locul potrivit pentru a le examina.

În următoarele cinci secțiuni vom examina cinci aspecte de performanță ale rețelei:

1. Probleme de performanță.
2. Măsurarea performanței rețelei.
3. Proiectarea de sistem pentru performanțe mai bune.
4. Prelucrarea rapidă TPDU.
5. Protocoale pentru rețele viitoare de mare performanță.

Ca o paranteză, avem nevoie de un nume pentru unitățile schimbate de către entitățile de transport. Termenul TCP de segment este cel mai confuz și în acest context nu este niciodată utilizat în afara lumii TCP. Termenii uzuali ATM (CS-PDU, SAR-PDU și CPCS-PDU) sunt specifici doar pentru ATM. Pachetele se referă în mod clar la nivelul rețea și mesajele aparțin nivelului aplicație. Din lipsa unui termen standard, vom reveni la numirea unităților schimbate de entitățile de transport TPDU-uri. Când ne referim atât la TPDU cât și la pachet, vom utiliza pachet ca un termen comun, ca de exemplu în „Procesorul trebuie să fie suficient de rapid pentru a prelucra pachetele în timp real." Prin aceasta subînțelegem atât pachetul nivelului rețea cât și TPDU-ul încapsulat în el.

6.6.1 Probleme de performanță în rețelele de calculatoare

Unele probleme de performanță, cum este congestia, sunt cauzate de supraîncărcarea temporară a resurselor. Dacă apare subit o creștere de trafic la nivelul unui ruter peste nivelul care poate fi controlat de acesta, se va produce o congestie și performanțele vor avea de suferit. Congestia a fost studiată în detaliu în capitolul precedent.

Performanțele se degradează de asemenea în cazul unei dezechilibrări structurale a balanței resurselor. De exemplu, dacă o linie de comunicație de un gigabit este atașată la un terminal PC de performanță scăzută, procesorul slab nu va putea prelucra suficient de repede pachetele care sosesc, unele din acestea pierzându-se. Aceste pachete vor fi retransmise în cele din urmă, adăugând întârzieri, consumând din lărgimea de bandă și în general reducând performanțele.

Supraîncărcarea poate fi de asemenea inițiată în mod sincron. De exemplu, dacă un TPDU conține un parametru eronat (de exemplu portul pentru care este destinat), în multe cazuri receptorul va înapoia cu multă grijă un anunț de eroare. Să vedem acum ce s-ar putea întâmpla dacă un TPDU eronat ar fi răspândit către 10000 de mașini: fiecare din ele ar putea întoarce un mesaj de eroare. **Furtuna de difuzare** rezultată ar putea să scoată din funcțiune rețeaua. UDP a suferit de această problemă până când protocolul a fost modificat pentru a împiedica mașinile să răspundă erorilor cauzate de TPDU-urile UDP-ului trimise către adrese de difuzare.

Un al doilea exemplu de supraîncărcare sincronă este dat de efectele unei căderi a energiei electrice. Odată cu revenirea curentului, toate mașinile execută programul ROM de reinițializare. O secvență tipică de reinițializare poate cere în primul rând unui anume server (DHCP) identitatea precisă a mașinii, apoi poate cere unui server de fișiere o copie a sistemului de operare. Dacă sute de mașini execută acest lucru simultan, serverul va ceda probabil la această încărcare.

Chiar în absența unei supraîncărcări sincrone și chiar atunci când sunt suficiente resurse disponibile, performanțele pot fi slabe datorită unei proaste reglări a sistemului. De exemplu, dacă o mașină are suficientă memorie și putere de prelucrare, dar nu a fost alocat suficient spațiu pentru tampoane, vor apărea aglomerări și se vor pierde TPDU-uri. Similar, dacă algoritmul de planificare nu acordă o prioritate suficient de mare prelucrării TPDU-urilor care sunt recepționate, unele din ele se vor pierde.

Un alt element de reglare este potrivirea corectă a intervalelor de limită de timp. Atunci când este trimis un TPDU, în mod normal se poziționează un contor pentru a evita pierderea TPDU-lui. Dacă limita de timp este prea scurtă, se vor produce retransmisii inutile, obturând cablurile. Dacă limita de timp este prea lungă, se vor introduce întârzieri inutile după pierderea unui TPDU. Alți parametri ce pot fi reglați sunt lungimea intervalului de timp după care datele acumulate sunt confirmate, și numărul retransmisiilor făcute înainte de a se renunța.

Rețelele gigabit aduc cu ele noi probleme de performanță. Să considerăm, de exemplu, transmisia unei rafale de date de 64 K octeți de la San Diego la Boston, pentru a umple tamponul de 64K

octeți al receptorului. Să presupunem că legătura este de 1 Gbps și că întârzierea într-un singur sens prin fibra de sticlă este de 20 ms. Inițial, la momentul $t = 0$, conducta este goală, ca în fig. 6-41(a). Doar cu 500 μs mai târziu, în fig. 6-41(b), toate TPDU-urile sunt deja plasate pe fibră. Primul TPDU transmis va fi acum undeva în vecinătatea Brawley-ului, încă departe, în California de Sud. Cu toate acestea, transmisia trebuie să se oprească până se primește o actualizare de fereastră.

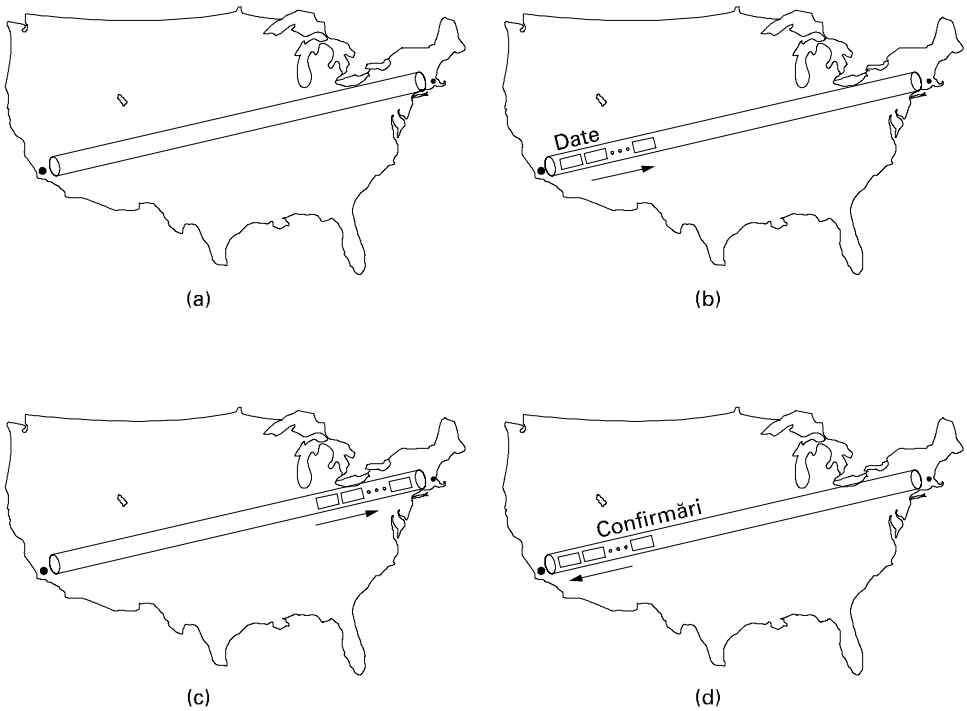


Fig. 6-41. Evoluția transmisiei unui megabit de la San Diego la Boston.
(a) La momentul t=0. (b) După 500 μsec. (c) După 20 ms. (d) După 40 ms.

După 20 ms, primul TPDU atinge Boston-ul, ca în fig. 6-41(c), și este confirmat. În final, la 40 ms după momentul inițial, prima confirmare sosește înapoi la emițător și a doua rafală poate să fie transmisă. Cum linia de transmisie a fost utilizată pentru 0.5 ms din cele 40 ms, eficiența este undeva în jurul a 1.25 procente. Această situație este tipică pentru cazul folosirii protocoalelor vechi în linii gigabit.

O mărime utilă de reamintit când se analizează performanțele rețelei este produsul **lărgime de bandă-întârziere**. El este obținut prin înmulțirea lărgimii de bandă (în biți pe secundă) cu întârzierea traseului dus-întors (în secunde). Produsul reflectă capacitatea conductei de la emițător la receptor și înapoi (în biți).

De exemplu, în fig. 6-41 produsul lărgime de bandă-întârziere este de 40 milioane de biți. Cu alte cuvinte, pentru a funcționa la viteză maximă, emițătorul poate transmite rafale de 40 milioane de biți până la recepționarea primei confirmări. Umplerea conductei (în ambele sensuri) presupune o cantitate însemnată de biți. Acesta este motivul pentru care o rafală de jumătate de milion de biți atinge o eficiență de 1.25 procente: ea reprezintă doar 1.25 procente din capacitatea conductei.

Concluzia care poate fi trasă aici este că pentru atingerea unor performanțe bune, fereastra receptorului trebuie să fie cel puțin la fel de mare ca și produsul lărgime de bandă-întârziere, prefera-

bil chiar puțin mai mare, deoarece receptorul poate să nu răspundă instantaneu. Pentru o linie gigabit transcontinentală sunt necesari cel puțin 5 megabiți.

Dacă eficiența este dezastruoasă pentru un megabit, imaginați-vă cum ar arăta ea pentru o cerere scurtă de câteva sute de octeți. În cazul în care nu se găsește o altă utilitate pentru linie în timpul în care primul client este în așteptarea răspunsului, o linie gigabit nu este cu nimic mai bună ca o linie megabit, numai că este mult mai scumpă.

O altă problemă de performanță care poate apărea în aplicațiile critice din punctul de vedere al timpului, precum audio/video, este zgomotul. Un timp mediu de transmisie de valoare mică nu este suficient. Este necesară și o deviație standard mică. Obținerea atât a unui timp mediu de transmisie mic, cât și a unei deviații standard mici, cere un efort serios de inginerie.

6.6.2 Măsurarea performanțelor rețelei

Atunci când performanțele unei rețele sunt slabe, utilizatorii ei se plâng persoanelor care o administrează, cerând îmbunătățirea situației. Pentru a crește performanțele, operatorii trebuie mai întâi să determine cu exactitate ce se întâmplă. Pentru a afla ce se întâmplă în realitate, operatorii trebuie să efectueze măsurători. În această secțiune ne vom concentra asupra măsurării performanțelor rețelei. Discuția care urmează se bazează pe lucrarea lui Mogul (1993).

Ciclul de bază utilizat pentru îmbunătățirea performanțelor rețelei conține următorii pași:

1. Măsurarea performanțelor și a parametrilor relevanți ai rețelei.
2. Încercarea de a înțelege ceea ce se petrece.
3. Modificarea unuia din parametri.

Acești pași se repetă până la atingerea unor performanțe suficient de bune sau până când este clar că și ultima îmbunătățire posibilă a fost pusă în aplicare.

Măsurarea poate fi făcută în multe moduri și în multe locuri (atât fizic cât și în stiva de protocoale). Cel mai simplu mod de măsurare este inițializarea unui contor la începutul unei activități și observarea timpului necesar pentru îndeplinirea acelei sarcini. De exemplu, un element cheie în măsurare este aflarea timpului necesar unui TPDU pentru a fi confirmat. Alte măsurători sunt făcute cu contoare care înregistrează frecvența de apariție a unor evenimente (de exemplu, numărul de TPDU-uri pierdute). În final, unii pot fi interesați să afle valorile unor mărimi, ca de exemplu numărul de octeți prelucrați într-un anume interval de timp.

Măsurarea performanțelor și a parametrilor rețelei ascunde multe capcane potențiale. În cele ce urmează, enunțăm doar câteva din acestea. Orice încercare sistematică de a măsura performanțele rețelei trebuie să le evite.

Dimensiunea testului trebuie să fie suficient de mare

Timpul necesar pentru a trimite un TPDU nu trebuie măsurat o singură dată, ci în mod repetat, să zicem, de un milion de ori, luându-se în considerare media valorilor rezultate. Un test de dimensiune mare va reduce gradul de incertitudine în media și deviația standard a măsurătorii. Această incertitudine poate fi calculată pe baza formulelor statistice obișnuite.

Testele trebuie să fie reprezentative

Ideal ar fi ca întreaga secvență a celor un milion de măsurători să fie repetată în diferite momente ale zilei și ale săptămânii pentru a pune în evidență efectul diferențelor de încărcare a sistemului asupra mărimii măsurate. Măsurătorile de congestie, de exemplu, nu sunt prea utile dacă sunt făcute

într-un moment în care nu există nici o congestie. Uneori, rezultatele pot fi inițial contrare intuiției, de exemplu congestii importante la orele 10, 11, sau 1, 2 după amiază, dar nici un fel de congestie la amiază (când toți utilizatorii au pauză de prânz).

Utilizarea ceasurilor cu granularitate mare trebuie făcută cu atenție

Ceasurile calculatoarelor funcționează prin incrementarea la intervale regulate a unui anumit contor. De exemplu, un contor de milisecunde adaugă unu la contor la fiecare 1 ms. Utilizarea unui astfel de contor pentru a măsura un eveniment care durează mai puțin de 1 ms nu este imposibilă, dar necesită o oarecare atenție. (Desigur, unele calculatoare au ceasuri cu precizie mai bună).

Pentru a măsura intervalul necesar trimiterii unui TPDU, de exemplu, ceasul sistem (să spunem în milisecunde) ar trebui să fie citit în momentul în care se intră în codul de transport și din nou când se părăsește acest cod. Dacă timpul real de transmisie TPDU este de 300 μsec, diferența dintre cele două citiri va fi sau 0, sau 1, ambele valori greșite. Cu toate acestea, dacă măsurătoarea este repetată de un milion de ori și suma tuturor măsurătorilor este împărțită la un milion, timpul mediu va avea o precizie mai bună de 1 μsec.

Nu trebuie să se petreacă ceva neașteptat în timpul măsurătorilor

Efectuarea măsurătorilor pe un sistem universitar în ziua în care urmează să fie predat vreun proiect important poate conduce la rezultate diferite față de cele ce s-ar obține în ziua imediat următoare. Similar, dacă vreun cercetător se decide să organizeze o videoconferință în rețea, în timpul testelor, poate fi obținut un rezultat alterat. Cel mai bine este ca testele să fie rulate pe un sistem complet inactiv, întreaga sarcină fiind construită în vederea testării. Chiar și această abordare are propriile capcane. Deși ne-am aștepta ca nimeni să nu utilizeze rețeaua la ora 3 dimineața, acesta poate să fie chiar momentul în care un program de salvare automată începe să copieze conținutul tuturor discurilor pe bandă. Mai mult decât atât, s-ar putea să existe un trafic important pentru minunatele pagini de Web de pe rețeaua testată, trafic provenit din zone aflate pe alte meridiane orare.

Lucrul cu memoria ascunsă poate distruge măsurătorile

Modalitatea evidentă de a măsura timpul de transfer al fișierelor este de a deschide un fișier de dimensiune mare, de a-l citi în întregime și de a-l închide, urmând a vedea cât de mult a durat toată operația. Se repetă apoi operația de mult mai multe ori, pentru a obține o medie corectă. Problema este că sistemul poate memora fișierul în memoria ascunsă, astfel încât doar prima măsurătoare să implice traficul în rețea. Restul nu sunt decât accese la memoria ascunsă locală. Rezultatele unei astfel de măsurători sunt, în esență, fără nici o valoare (doar dacă nu cumva se dorește măsurarea performanțelor memoriei ascunse).

De obicei, se poate ocoli memoria ascunsă prin simpla ei supraîncărcare. De exemplu, dacă memoria ascunsă este de 10 megaocteți, ciclul de test ar putea deschide, citi și închide două fișiere de 10 megaocteți la fiecare buclă, în tentativa de a forța rata de succes în accesul la memoria ascunsă la 0. Cu toate acestea, dacă nu se înțelege cu absolută precizie algoritmul de manipulare a memoriei ascunse, trebuie procedat cu grijă.

Memorarea datelor în tampoane poate avea același efect. Se cunoaște un program utilitar popular pentru măsurarea performanțelor TCP/IP care raportează performanțe ale UDP-ului substanțial mai mari decât o permit liniile fizice. Cum se întâmplă acest lucru? Un apel către UDP întoarce în mod normal controlul odată ce mesajul a fost acceptat de către nucleu și adăugat la coada de transmisie. Dacă este suficient spațiu în tampon, măsurarea a 1000 de apeluri UDP nu înseamnă neapă-

rat că informațiile au fost transmise. Cea mai mare a informațiilor poate să se afle încă în nucleu, dar instrumentul de măsurare interpretează că ele au fost toate deja transmise.

Trebuie înțeles ceea ce se măsoară

Atunci când se măsoară timpul necesar pentru a citi un fișier de la distanță, măsurătorile depind de rețea, de sistemele de operare de la ambele capete - client și server, de tipul de echipament al plăcii de interfață utilizat, de programele care le controlează și de alți factori. Procedând cu atenție, putem determina în ultimă instanță timpul de transfer al fișierului pentru configurația utilizată. Dacă scopul îl reprezintă reglarea acestei configurații particulare, atunci măsurătorile sunt în regulă.

Cu toate acestea, dacă, în scopul alegerii unei interfețe de rețea pentru a fi cumpărată, sunt făcute măsurători similare pe trei sisteme diferite, rezultatele pot fi complet bulversate în cazul în care unul din programele care controlează echipamentul este de-a dreptul îngrozitor și utilizează doar 10% din performanțele plăcii.

Atenție la extrapolarea rezultatelor

Să presupunem că s-au făcut măsurători ale unei anumite mărimi prin simularea încărcării rețelei între 0 (sistem complet descărcat) și 0.4 (sistem încărcat în proporție de 40%), conform punctelor de date și liniilor continue dintre ele din fig. 6-42. Ar putea fi tentant să se extrapoleze liniar, așa cum o sugerează linia punctată. Cu toate acestea, multe din rezultatele anterioare indică un factor de $1/(1 - \rho)$, unde ρ este încărcarea, astfel încât valorile adevărate pot arăta mai degrabă ca linia întreruptă, care crește mult mai repede decât liniar.

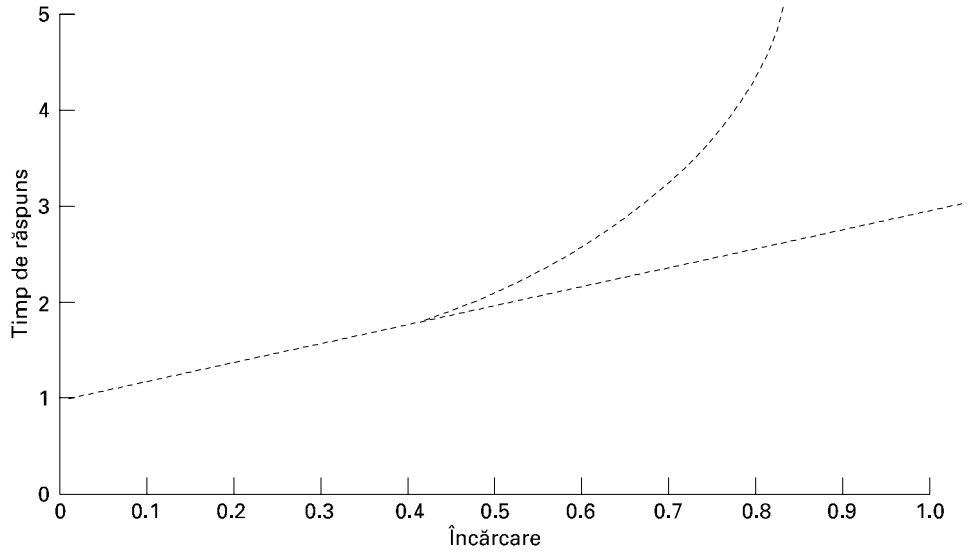


Fig. 6-42. Răspunsul, ca o funcție de încărcare.

6.6.3 Proiectarea de sistem pentru performanțe superioare

Măsurarea și ajustarea pot îmbunătăți considerabil performanțele, dar ele nu pot substitui o proiectare făcută bine de la început. O rețea proiectată superficial poate fi îmbunătățită tot în aceeași măsură. O soluție mai bună este să se refacă totul de la temelie.

În această secțiune, vom prezenta câteva reguli rezultate dintr-o experiență acumulată pe un număr mare de rețele. Aceste reguli se referă nu doar la proiectarea de rețea, ci și la proiectarea de sistem, și aceasta pentru că programul și sistemul de operare sunt deseori mai importante decât ruterul și echipamentul de interfață. Cele mai multe din aceste idei constituiau cunoștințe de bază ale proiectanților de rețele, care erau propagate din generație în generație pe cale orală. Ele au fost prima oară enunțate explicit de Mogul (1993); expunerea noastră merge în mare parte în paralel cu a sa. O altă sursă relevantă este (Metcalfe, 1993).

Regula #1: Viteza procesorului este mai importantă decât viteza rețelei.

O experiență îndelungată a arătat că în aproape toate rețelele, supraîncărcarea indusă de sistemul de operare și de protocol domină de fapt timpul de transmisie pe cablu. De exemplu, în teorie, timpul minim pentru un RPC pe o rețea Ethernet este de 102 μs, corespunzând unei cereri minime (64 de octeți) urmate de un răspuns minim (64 de octeți). În realitate, evitarea întârzierii suplimentare introduse de software și obținerea timpului RPC cât de cât apropiat de cel teoretic este o realizare considerabilă.

Similar, cea mai mare problemă în funcționarea la 1 Gbps constă în plasarea biților din tamponul utilizatorului pe fibră suficient de repede și în recepționarea acestor biți de către procesorul receptor la fel de repede cum sosesc. Pe scurt, dacă se dublează viteza procesorului, de regulă se poate ajunge până aproape de dublarea productivității. Dublarea capacității rețelei nu are de regulă nici un efect, deoarece gâtuirea se produce în general la calculatoarele gazdă.

Regula #2: Reducerea numărului de pachete pentru a reduce supraîncărcarea datorată programelor.

Prelucrarea unui TPDU adaugă o anumită supraîncărcare per TPDU (de exemplu prelucrarea antetului) și o anumită cantitate de prelucrare suplimentară per octet (de exemplu calculul sumei de control). Atunci când se trimit un milion de octeți, supraîncărcarea per octet este aceeași, indiferent dacă dimensiunea TPDU variază. Cu toate acestea, utilizarea de TPDU-uri de 128 octeți presupune de 32 de ori mai multă supraîncărcare per TPDU față de cazul a 4K octeți per TPDU. Această supraîncărcare crește cu rapiditate.

În plus față de supraîncărcarea TPDU-ului, trebuie considerată și supraîncărcarea datorată nivelurilor inferioare. Fiecare sosire a unui pachet generează o întrerupere. Pe un procesor cu bandă de asamblare modern, fiecare întrerupere fragmentează banda de asamblare a procesorului, interferă cu memoria tampon, presupune o schimbare de context în controlul memoriei și impune salvarea unui număr important din registrele procesorului. O divizare prin n a numărului de TPDU-uri trimise reduce în consecință numărul de întreruperi și supraîncărcarea pachetelor cu un factor de n.

Această observație justifică necesitatea colectării unei cantități importante de date înaintea transmisiei, în scopul reducerii numărului de întreruperi la celălalt capăt. Algoritmul Nagle și soluția Clark pentru sindromul ferestrei stupide reprezintă încercări de a face exact acest lucru.

Regula #3: Minimizarea numărului de comutări de context.

Comutările de context (de exemplu, din mod nucleu în mod utilizator) sunt catastrofale. Ele au aceleași proprietăți incomode ca și întreruperile, cea mai rea fiind o lungă serie de eșecuri la accesul inițial la memoria ascunsă. Comutările de context pot fi reduse dacă funcția de bibliotecă ce trimite date le stochează intern până la acumularea unei cantități semnificative. Similar, de partea receptorului, TPDU-urile de dimensiune mică recepționate ar trebui colectate și trimise utilizatorului dintr-un singur foc și nu individual, în scopul minimizării comutărilor de context.

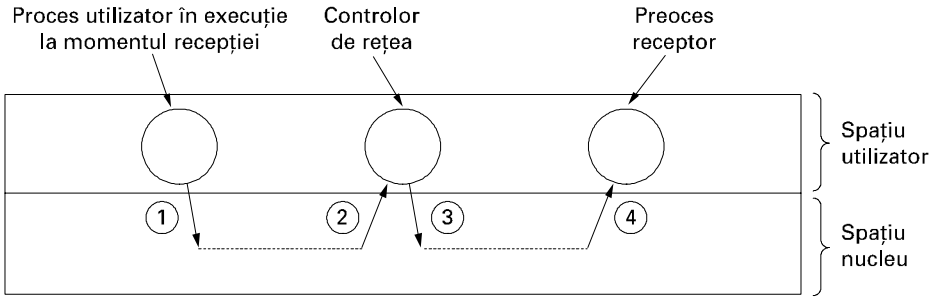


Fig. 6-43. Patru comutări de context pentru fiecare pachet, cu un controlor de rețea în spațiul utilizator.

În cel mai bun caz, sosirea unui pachet produce o comutare de context din modul utilizator curent în modul nucleu și apoi o comutare către procesul receptor, astfel încât acesta din urmă să preia informația nou sosită. Din nefericire, în multe sisteme de operare se petrec comutări de contexte suplimentare. De exemplu, dacă procesul controlor de rețea rulează ca un proces special în spațiul utilizator, sosirea unui pachet provoacă, probabil, o comutare de context de la utilizatorul curent către nucleu, apoi încă o comutare de la nucleu către controlorul de rețea, urmată de încă una înapoi către nucleu și în final din nucleu către procesul receptor. Această secvență este prezentată de fig. 6-43. Toate aceste comutări de contexte pentru fiecare pachet consumă mult din timpul procesorului și au un efect distrugător asupra performanțelor rețelei.

Regula #4: Minimizarea numărului de copieri.

Efectuarea unor copii multiple este și mai dăunătoare decât comutările multiple de contexte. Nu este nimic deosebit în faptul că un pachet proaspăt recepționat este copiat de trei sau patru ori înainte ca TPDU-ul conținut în el să fie livrat. După ce un pachet este recepționat de către echipamentul de rețea într-un tampon special cablat pe placă, el este copiat de obicei într-un tampon al nucleului. De aici el este copiat într-un tampon al nivelului rețea, apoi într-unul al nivelului transport și în final de către procesul aplicației receptoare.

Un sistem de operare inteligent va copia câte un cuvânt odată, dar nu este deloc neobișnuit să fie necesare cinci instrucțiuni per cuvânt (încărcare, memorare, incrementarea unui registru index, un test pentru marcajul de sfârșit al datelor și un salt condiționat). A face trei copii ale fiecărui pachet la cinci instrucțiuni copiate pe fiecare cuvânt de 32 de biți, necesită 15/4 sau aproximativ patru instrucțiuni per octet copiat. Pe un procesor de 500 MIPS, o instrucțiune durează 2 ns deci fiecare octet necesită 8 ns sau aproximativ 1 ns per bit, dând o rată de transfer de aproximativ 1 Gbps. Ținând cont și de prelucrarea antetului, tratarea întreruperilor și a comutărilor de contexte, se pot atinge 500 Mbps, aceasta fără a pune la socoteală prelucrarea efectivă a datelor. Evident, controlul unei linii Ethernet de 10 Gbps care funcționează la viteză maximă, nici nu intră în discuție.

De fapt, probabil că nici o linie de 500 Mbps nu poate fi manipulată la viteză maximă. În calculul anterior am presupus că o mașină cu 500 MIPS poate executa oricare 500 milioane de instrucțiuni pe secundă. De fapt, mașinile pot opera la această viteză doar dacă nu execută referiri la memorie. Operațiile cu memoria sunt, de cele mai multe ori, de zece ori mai lente decât instrucțiunile registru-registru (adică 20 ns / instrucțiune). Dacă 20 de procente din instrucțiuni chiar referă memoria (adică datele necesare nu se află în memoria tampon), ceea ce este de așteptat când este vorba despre pachetele care vin, timpul mediu de execuție este 5.6 ns (0.8x2+0.2x20). Cu patru instrucțiuni/octet,

avem nevoie de 22.4 ns / octet, sau 2.8 ns / bit, care înseamnă aproximativ 357 Mbps. Considerând un procent de 50 supraîncărcare obținem 178 Mbps. De notat că echipamentul suport nu ajută în acest caz. Problema este că sistemul de operare execută prea multe operații de copiere.

Regula #5: Oricând se poate cumpăra mai multă lărgime de bandă, dar niciodată o întârziere mai mică.

Următoarele trei reguli se ocupă de comunicație mai mult decât de prelucrarea protocolului. Prima regulă stabilește că, dacă se dorește o lățime de bandă mai mare, este suficient să o cumperi. Dacă se pune o a doua fibră alături de prima, se dublează lărgimea de bandă, dar nu se micșorează deloc întârzierile. Micșorarea întârzierilor presupune îmbunătățirea programului de protocol, a sistemului de operare sau a interfeței cu rețeaua. Chiar dacă toate acestea sunt îndeplinite, întârzierea nu se va reduce dacă gâtuirea constă în timpul de transmisie.

Regula #6: Evitarea congestiei este preferabilă eliminării congestiei.

Vechea maximă conform căreia o uncie de prevenire este mai bună decât o livră de însănătoșire este cu certitudine valabilă și în cazul congestiei rețelei. Atunci când o rețea este congestionată, se pierd pachete, lărgimea de bandă este irosită, apar întârzieri inutile și multe altele. Recuperarea din congestie ia timp și răbdare. Este de preferat să se procedeze de așa natură, încât congestia să nu apară. Evitarea congestiei este ca și vaccinarea împotriva tetanosului: doare puțin în momentul în care se face vaccinul, dar aceasta împiedică o durere mult mai mare mai târziu.

Regula #7: Evitarea întârzierilor.

În rețele sunt necesare contoare, dar ele ar trebui utilizate cu măsură și întârzierile ar trebui minimizate. Atunci când expiră un contor, se repetă de regulă o acțiune. Dacă este într-adevăr necesară repetarea acțiunii respective, atunci asta este, dar repetarea ei fără rost reprezintă o risipă.

Modalitatea de a evita un efort suplimentar constă în înțelegerea faptului că aceste contoare se cam situează de partea conservatoare a lucrurilor. Un contor căruia îi ia mult timp ca să expire adaugă o mică întârziere suplimentară în cazul (puțin probabil) în care un TPDU se pierde. Un contor care expiră atunci când nu ar trebui, consumă în mod nepermis din timpul procesor, irosește din lărgimea de bandă și adaugă o încărcare suplimentară în zeci de rutere, probabil fără nici un motiv serios.

6.6.4 Prelucrarea rapidă a TPDU-urilor

Morala poveștii anterioare este aceea că principalul obstacol către rețelele rapide îl constituie programul de protocol. În această secțiune, vom studia câteva modalități pentru a crește viteza acestui program. Pentru mai multe informații, pot fi citite (Clark și alții 1989; și Chase și alții, 2001).

Supraîncărcarea indusă de prelucrarea TPDU-urilor are două componente: supraîncărcare per TPDU și supraîncărcare per octet. Ambele trebuie să fie abordate. Cheia accelerării prelucrării TPDU-urilor constă în separarea cazului normal (transferul datelor într-un singur sens) și tratarea sa specială. Cu toate că este necesară o secvență de TPDU-uri speciale pentru a se intra într-o stare *STABILIT*, odată intrat în starea respectivă, prelucrarea TPDU-urilor decurge lin, până în clipa în care una din părți inițiază închiderea conexiunii.

Să începem examinarea părții emițătoare din starea *STABILIT*, atunci când există date de transmis. Pentru claritate, presupunem că entitatea transport este în nucleu, aceleași idei aplicându-se și în cazul în care este vorba de un proces în spațiul utilizator sau o bibliotecă în interiorul proce-

sului emițător. În fig. 6-44, pentru a executa SEND-ul, procesul emițător intră în mod nucleu prin acționarea unei capcane. Primul lucru pe care îl face entitatea transport este de a testa dacă nu cumva este vorba de cazul normal: starea este *STABILIT*, nici o parte nu încearcă să închidă conexiunea, un TPDU normal (adică unul care nu e în afara limitelor) este în curs de a fi transmis și la receptor este disponibil un spațiu fereastră suficient. Dacă toate condițiile sunt îndeplinite, nici un test suplimentar nu mai este necesar și poate fi acaparată calea rapidă prin entitatea de transport emițătoare. De obicei, această cale este acaparată în majoritatea timpului.

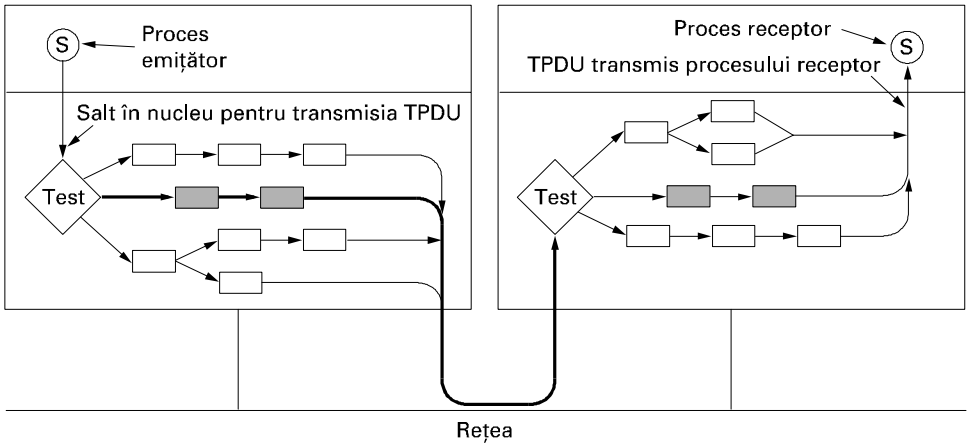


Fig. 6-44. Calea rapidă de la emițător la receptor este indicată printr-o linie groasă. Pașii de prelucrare ai acestei căi sunt reprezentați prin dreptunghiuri umbrite.

În cazul obișnuit, antetele mai multor date TPDU consecutive sunt în mare parte identice. Pentru a profita de acest lucru, în interiorul entității transport se memorează un antet prototip. La începutul căii rapide, el este copiat cât de repede posibil într-un tampon special, cuvânt cu cuvânt. Acele câmpuri care ulterior se modifică de la un TPDU la altul sunt suprascrise în tampon. În mod frecvent, aceste câmpuri sunt ușor de derivat din variabilele de stare, de exemplu următorul număr de secvență. Apoi este pasat nivelului rețea un indicator spre întregul antet TPDU, împreună cu un indicator spre informația utilizator. Și aici se poate urma aceeași strategie (caz neacoperit de fig. 6-44). În final, nivelul rețea furnizează pachetul rezultat nivelului legătură de date, în vederea transmisiei.

Pentru a exemplifica modul în care operează acest principiu în practică, să considerăm cazul TCP/IP-ului. Fig. 6-45(a) prezintă antetul TCP. Câmpurile hașurate sunt identice între două TPDU-uri consecutive, pe un flux într-un singur sens. Tot ce are de făcut entitatea transport este să copieze cele cinci cuvinte dintr-un antet prototip în tamponul care urmează să fie transmis, să completeze următorul număr de secvență (prin copierea lui dintr-un cuvânt din memorie), să calculeze suma de control și să incrementeze numărul de secvență din memorie. Entitatea poate înmâna apoi antetul, împreună cu datele aferente, unei proceduri IP speciale, în vederea transmisiei obișnuite a unui TPDU de dimensiune maximă. În continuare IP copiază cele cinci cuvinte ale antetului său prototip [vezi fig. 6-45(b)] în tampon, completează câmpul *Identificare* și calculează suma sa de control. Pachetul este acum gata pentru transmisie.

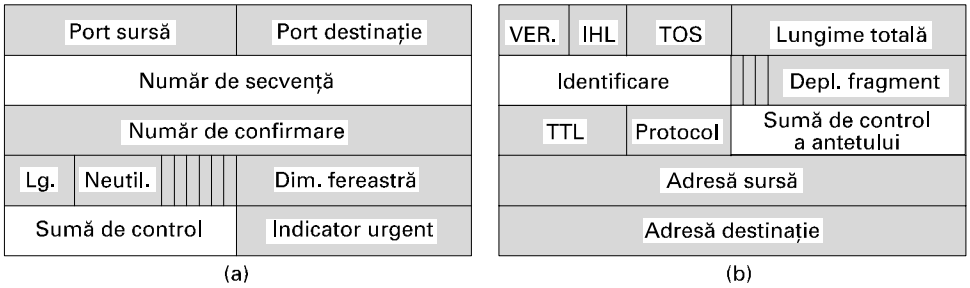


Fig. 6-45. (a) Antetul TCP. (b) Antetul IP. În ambele cazuri, câmpurile hașurate se obțin din prototip, fără nici o modificare.

Să aruncăm o privire asupra prelucrării pe calea rapidă în cazul receptorului din fig. 6-44. Pasul 1 constă în localizarea înregistrării de conexiune din TPDU-ul recepționat. În cazul TCP, înregistrarea de conexiune poate fi memorată într-o tabelă de dispersie pentru care cheia poate fi o funcție simplă aplicată celor două adrese IP și celor două porturi. Odată ce înregistrarea de conexiune a fost localizată, corectitudinea sa trebuie verificată prin compararea ambelor adrese și ambelor porturi.

O optimizare care accelerează și mai mult determinarea înregistrării de conexiune constă în menținerea unui indicator către ultima înregistrare utilizată, urmând ca aceasta să fie prima înregistrare testată. Clark ș.a. (1989) au aplicat această idee și au observat o rată de succes care depășește 90%. Alte euristici de căutare sunt descrise în (McKenney și Dove, 1992).

În continuare, TPDU-ul este verificat pentru a determina dacă este vorba de cazul normal: starea este *STABILIT*, niciuna din părți nu încearcă închiderea conexiunii, este un TPDU complet, nici un indicator special nu este poziționat și numărul de secvență este cel așteptat. Aceste teste înseamnă doar câteva instrucțiuni. Dacă toate condițiile sunt îndeplinite, este invocată o procedură TCP pentru cale rapidă.

Calea rapidă actualizează înregistrarea de conexiune și copiază informația către utilizator. Totodată, suma de control este calculată chiar pe parcursul copierii, eliminând astfel trecerile suplimentare pe secvența de date. Dacă suma de control este corectă, înregistrarea de conexiune este actualizată și se trimite o confirmare. Schema generală care constă într-un control rapid la început, pentru a vedea dacă antetul este cel așteptat, precum și în existența unei proceduri speciale care tratează cazul respectiv, se numește **predicția antetului**. Schema este utilizată în multe din implementările TCP. Atunci când această optimizare este utilizată împreună cu celelalte optimizări discutate în acest capitol, este posibil ca TCP-ul să atingă la execuție 90% din viteza de copiere locală memorie - memorie, presupunând că mediul de comunicație este suficient de rapid.

Alte două domenii unde se pot obține câștiguri importante în performanțe sunt controlul tampoanelor și al contoarelor de timp. În controlul tampoanelor, ideea constă în evitarea copierilor inutile, așa cum s-a menționat anterior. Controlul contoarelor de timp este important, deoarece aproape nici un contor nu expiră de fapt. Acestea sunt poziționate astfel, încât să ne păzească împotriva pierderii de TPDU-uri, dar majoritatea TPDU-urilor, ca și confirmările lor, de altfel, ajung corect la destinație. Este deci important să se optimizeze controlul contoarelor de timp pentru cazul în care acestea expiră rar.

O schemă uzuală constă în utilizarea unei liste îmbunătățite a evenimentelor generate de contoare, sortate în funcție de momentul expirării acestora. Intrarea din capătul acestei liste conține un contor care indică depărtarea în impulsuri de ceas față de momentul expirării. Fiecare din intrările

care urmează indică printr-un contor depărtarea în impulsuri de ceas față de intrarea precedentă. Astfel, pentru evenimente care expiră în 3, 10 și respectiv 12 impulsuri, cele trei contoare sunt 3, 7 și, respectiv, 2.

La fiecare impuls de ceas, contorul din capătul listei este decrementat. Atunci când contorul atinge valoarea 0, se prelucrează evenimentul asociat lui și următorul element din listă devine capătul listei. Contorul acestuia nu trebuie să fie modificat. Prin această schemă, inserarea și ștergerea evenimentelor sunt operații costisitoare, cu un timp de execuție proporțional cu lungimea listei.

Dacă intervalul maxim de expirare al contorului este limitat și cunoscut în avans, poate fi utilizată o abordare mai eficientă. În acest caz, poate fi utilizat un vector numit **roata timpului**, ca în fig. 6-46. Fiecare poziție corespunde unui impuls de ceas. Ceasul curent reprezentat este $T = 4$. Contoarele sunt planificate să expire la 3, 10 și 12 impulsuri față de acest moment. Dacă un contor nou este brusc poziționat să expire peste 7 impulsuri, se creează pur și simplu o intrare în poziția 11. Similar, în cazul în care contorul planificat să expire la $T + 10$ trebuie să fie anulat, trebuie parcursă lista care începe pe poziția 14 și intrarea cerută trebuie ștearsă. Să observăm că vectorul din fig. 6-46 nu poate suporta contoare care expiră după $T + 15$.

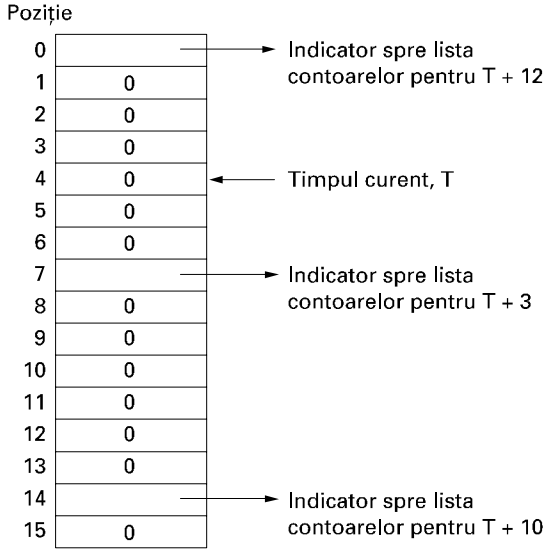


Fig. 6-46. O roată a timpului.

Cu fiecare impuls de ceas indicatorul de timp curent este avansat cu o poziție (circular). Dacă intrarea indicată este nenulă, sunt prelucrate toate contoarele asociate ei. Mai multe variațiuni pe această temă sunt discutate în (Varghese și Lauck, 1987).

6.6.5 Protocoale pentru rețele gigabit

La începutul anilor '90 au început să apară rețele gigabit. Prima reacția a oamenilor a fost să utilizeze pentru ele vechile protocoale, lucru care a pus în scurt timp diferite probleme. Vom discuta în această secțiune câteva din aceste probleme, precum și direcțiile urmate de noile protocoale pentru a le soluționa, pe măsură ce evoluăm către rețele tot mai rapide.

Prima problemă este că multe protocoale utilizează secvențe de numere de 32 de biți. Când a apărut Internetul, liniile între rutere erau mai ales linii închiriate de 56 Kbps, astfel că unui calculator gazdă care mergea la viteză maximă îi trebuia mai mult de o săptămână să treacă prin toate numerele de secvență. Pentru proiectanții de TCP, 2^{32} a fost o aproximație destul de decentă a infinitului, pentru că era mic pericolul ca pachete vechi să mai fie prin preajmă la o săptămână după ce au fost transmise. Cu o linie Ethernet de 10 Mbps, timpul de parcurgere a numerelor de secvență a devenit de 57 de minute, mult mai mic, dar încă administrabil. Cu o linie Ethernet de 1 Gbps transmițând date prin Internet, timpul de parcurgere este de aproximativ 34 secunde, mult sub timpul maxim de viață al unui pachet în Internet, de 120 de secunde. Dintr-o dată, 2^{32} nu mai este o aproximare atât de bună a infinitului, din moment ce un transmițător poate cicla prin spațiul de secvență, în timp ce pachetele vechi există încă. RFC 1323 oferă totuși o fereastră de ieșire.

Problema este că mulți proiectanți de protocoale au plecat de la presupunerea că timpul necesar consumării spațiului numerelor de secvență depășește cu mult timpul maxim de viață al unui pachet. În consecință, nu era nici măcar nevoie să își facă griji că duplicate vechi pot să existe încă undeva atunci când numerele de secvență au revenit la vechile valori. La viteze gigabit această presupunere cade.

O a doua problemă este aceea că viteza de comunicație a crescut mult mai repede decât viteza de prelucrare. (Notă pentru inginerii de calculatoare: Ieșiți în stradă și bateți-i pe inginerii de comunicații! Ne bazăm pe voi.) În anii '70, ARPANET-ul opera la 56 Kbps și avea calculatoare care funcționau la aproape 1 MIPS. Pachetele erau de 1008 biți și astfel ARPANET-ul putea livra aproximativ 56 pachete/sec. Având disponibile 18 ms pentru fiecare pachet, o mașină gazdă putea să-și permită să irosească 18000 instrucțiuni pentru prelucrarea unui pachet. Desigur, dacă ar fi făcut astfel, ar fi asfixiat complet procesorul, dar putea renunța la doar 9000 instrucțiuni per pachet și tot i-ar mai fi rămas jumătate din puterea procesorului pentru celelalte prelucrări.

Să comparăm aceste numere cu calculatoarele moderne de 1000 MIPS care interschimbă pachete de 1500 de octeți pe o linie gigabit. Pachetele pot curge cu o viteză de peste 80000 pe secundă, astfel încât, dacă vrem să rezervăm jumătate din puterea procesorului pentru aplicații, prelucrarea unui pachet trebuie să se încheie în 6,25 μsec. În 6,25 μsec, un calculator de 1000 MIPS poate executa doar 6250 instrucțiuni, doar 1/3 din ceea ce își putea permite un calculator gazdă ARPANET. Mai mult decât atât, instrucțiunile RISC moderne fac mai puține lucruri per instrucțiune decât o făceau vechile instrucțiuni CISC, deci problema este chiar mai gravă decât pare. În concluzie: există mult mai puțin timp pentru prelucrarea efectuată de protocol decât exista altădată, deci protocoalele trebuie să devină mai simple.

O a treia problemă este aceea că protocolul cu întoarcere în *n* pași are performanțe slabe pe linii cu un produs lărgime de bandă-întârziere de valoare mare. Să considerăm de exemplu o linie de 4000 km operând la 1 Gbps. Timpul de transmisie dus-întors este de 40 ms, timp în care un emițător poate transmite 5 megaocteți. Dacă se detectează o eroare, atunci vor fi necesare 40 ms înainte ca emițătorul să fie avertizat de acest lucru. Dacă este utilizat algoritmul cu întoarcere în *n* pași, emițătorul nu va avea de retransmis doar pachetul eronat, ci și toți cei 5 megaocteți care au urmat după el. În mod clar, are loc o risipă mare de resurse.

O a patra problemă este aceea că liniile gigabit sunt fundamental diferite de liniile megabit, liniile gigabit de lungime mare fiind limitate de întârziere mai degrabă decât de lărgimea de bandă. În fig. 6-47 arătăm timpul necesar transferului unui fișier de un megabit la 4000 km distanță pentru diferite viteze de transfer. La o viteză de până la 1 Mbps, timpul de transmisie este dominat de viteza la care pot fi transferați biții. La 1 Gbps, întârzierea de 40 ms a circuitului dus-întors domină acea

milisecundă necesară pentru a pune bitul pe fir. Creșteri suplimentare ale lărgimii de bandă aproape că rămân fără efect.

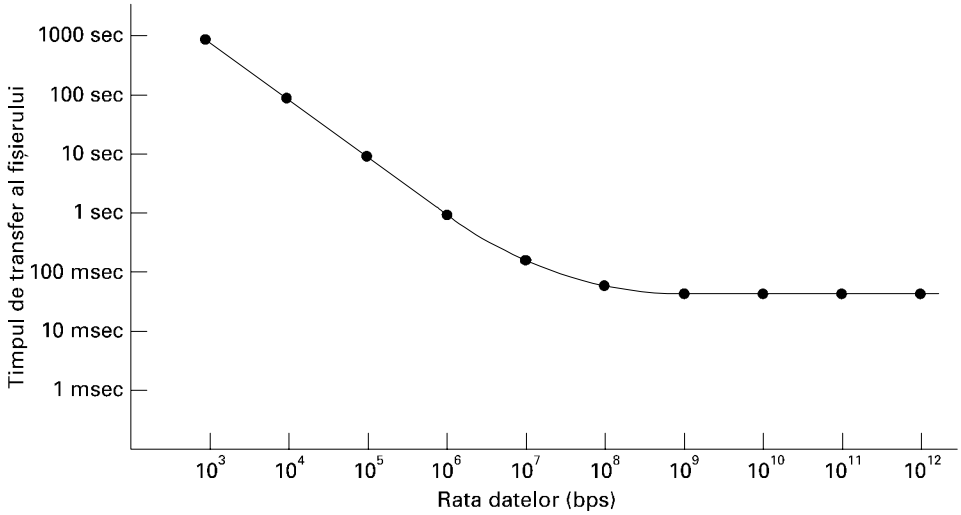


Fig. 6-47. Timpul necesar transferului și confirmării unui fișier de un megabit pe o linie de 4000 km.

Fig. 6-47 are implicații nedorite pentru protocoalele de rețea. Ea arată că protocoalele pas-cu-pas (stop-and-wait), precum RPC, au o limită superioară inerentă de performanță. Această limită este dictată de viteza luminii. Oricât de mare ar fi progresul tehnologic în optică, nu se poate obține nici o îmbunătățire (deși noi legi ale fizicii ar fi folositoare).

O a cincea problemă care merită menționată nu este o problemă de protocol sau o problemă tehnologică, ci un rezultat al noilor aplicații. Enunțată doar, ea spune că pentru multe aplicații gigabit, precum multimedia, varianța sosirii pachetelor în timp este la fel de importantă ca însăși media întârzierilor. De cele mai multe ori este de preferat o viteză de livrare redusă, dar uniformă uneia rapide, dar fluctuante.

Să revenim acum de la identificarea problemelor la modalitățile lor de rezolvare. Vom face, pentru început, câteva remarci generale, apoi vom analiza mecanismele de protocol, aspectul pachetelor și programele de protocol.

Principiul de bază pe care toți proiectanții de rețele gigabit ar trebui să-l învețe pe de rost este:

Proiectați astfel încât să optimizați viteza, nu lățimea de bandă.

Protocoalele vechi au fost de regulă proiectate pentru a minimiza numărul de biți de pe fir, prin utilizarea frecventă a câmpurilor de dimensiuni mici și prin împachetarea lor în octeți și cuvinte. În zilele noastre, există suficient de multă lățime de bandă. Problema o reprezintă prelucrările efectuate de protocoale, deci acestea trebuie reduse la minim. Clar că proiectanții procesorului IPv6 au înțeles acest principiu.

O modalitate tentantă pentru accelerarea lucrurilor este de a construi interfețe rapide cu rețeaua sub formă de echipamente fizice. Dificultatea acestei strategii constă în aceea că, în lipsa unui protocol extrem de simplu, un nou echipament înseamnă conectarea unei noi plăci, cu un al doilea

procesor și cu propriul program. Pentru a asigura faptul că un co-procesor de rețea este mai ieftin decât procesorul principal, el este de regulă o componentă mai lentă. Consecința acestei soluții este că mult timp, procesorul principal (rapid) este inactiv, așteptând ca al doilea procesor (cel lent) să facă toată munca critică. Este un mit acela că procesorul principal ar avea altceva de făcut în acest timp. Mai mult decât atât, atunci când comunică două procesoare independente, pot apărea condiții de cursă, fiind deci necesare protocoale complexe, pentru sincronizarea lor corectă. În general, cea mai bună abordare este de a concepe protocoalele cât mai simplu și de a lăsa procesorul principal să facă toată treaba.

Să studiem acum noțiunea de reacție în protocoalele de mare viteză. Datorită buclei de întârziere (relativ) lungă, reacția ar trebui evitată: receptorului îi ia prea mult timp pentru a anunța emițătorul. Un exemplu de reacție este controlul vitezei de transfer prin utilizarea unui protocol cu fereastră glisantă. Pentru a evita întârzierile (îndelungate) inerente atunci când receptorul trimite emițătorului actualizările de fereastră, cel mai bine este să se utilizeze o rată de transfer dictată de protocol. Într-un astfel de protocol, emițătorul poate trimite tot ceea ce dorește el să trimită, cu condiția să nu o facă mai repede decât s-a convenit inițial cu receptorul.

Un al doilea exemplu de reacție este algoritmul startului lent al lui Jacobson. Acest algoritm face interogări multiple pentru a determina cât de mult poate suporta rețeaua. În rețelele de mare viteză utilizarea a jumătate de duzină de interogări scurte pentru testarea răspunsul rețelei conduce la irosirea unei părți imense din lățimea de bandă. O schemă mai eficientă este ca emițătorul, receptorul și rețeaua să rezerve resursele necesare în momentul stabilirii conexiunii. Rezervarea în avans a resurselor are de asemenea avantajul diminuării fluctuațiilor. Pe scurt, migrarea spre viteze mari împinge inexorabil proiectarea spre operații orientate pe conexiuni sau spre ceva foarte apropiat de acestea. Desigur, dacă lărgimea de bandă nu va mai fi o problemă în viitor încât nimănui să nu-i mai pese de pierderi, regulile de proiectare vor deveni foarte diferite.

În rețelele gigabit, o importanță deosebită trebuie acordată organizării pachetelor. Antetul ar trebui să conțină cât mai puține câmpuri cu putință pentru a reduce timpul de prelucrare, iar aceste câmpuri ar trebui să fie suficient de mari pentru a-și îndeplini scopul și pentru a fi aliniate la cuvânt, fiind astfel mai ușor de prelucrat. În acest context, „suficient de mari” înseamnă eliminarea problemelor precum revenirea numerelor de secvență la valori vechi cât timp există încă pachete vechi, incapacitatea receptorilor de a oferi suficient spațiu de fereastră datorită unei dimensiuni prea mici a câmpului fereastră ș.a.m.d.

Antetul și informația ar trebui să fie prevăzute cu sume de control separate, din două motive. În primul rând, pentru a face posibilă calcularea sumei de control a antetului, dar nu și a datelor. În al doilea rând, pentru a determina corectitudinea antetului înaintea începerii copierii datelor în spațiul utilizator. Este de dorit să se calculeze suma de control a datelor la momentul copierii lor în spațiul utilizator, dar dacă antetul este incorect, copierea poate să se facă în spațiul unui alt proces. Pentru a evita o copiere incorectă, dar pentru a permite și calculul sumei de control a datelor în timpul copierii, este esențial ca cele două sume de control să fie separate.

Dimensiunea maximă a datelor ar trebui să fie mare pentru a permite operații eficiente chiar și în situația transferului la distanțe mari. De asemenea, cu cât este mai mare blocul de informație, cu atât este mai mică fracțiunea din totalitatea lățimii de bandă alocată antetului. 1500 de octeți este prea puțin.

O altă caracteristică importantă este posibilitatea de a trimite o cantitate rezonabilă de informație o dată cu cererea de conexiune. În acest fel se poate salva un timp de comunicație dus-întors.

În sfârșit, sunt utile câteva cuvinte despre programul de protocol. Cea mai mare atenție trebuie acordată cazului de succes. Multe din protocoalele vechi aveau tendința de a evidenția ce este de făcut atunci când ceva nu mergea cum trebuie (de exemplu pierderea unui pachet). Pentru a face protocoalele să meargă mai repede, proiectanții ar trebui să aibă ca scop minimizarea timpului de prelucrare atunci când totul funcționează corect. Minimizarea timpului de prelucrare în caz de eroare trebuie să treacă pe planul doi.

Un al doilea aspect legat de programe este minimizarea timpului de copiere. Așa cum am văzut mai devreme, copierea datelor introduce în general un cost suplimentar. Ideal ar fi ca echipamentul fizic să depoziteze în memorie fiecare pachet recepționat ca un bloc contiguu de date. Programul ar trebui apoi să copieze acest pachet în tamponul utilizator printr-o singură copiere de bloc. În funcție de modul de lucru al memoriei tampon, ar fi de dorit chiar să se evite copierea în buclă. Cu alte cuvinte, cel mai rapid mod de a copia 1024 de cuvinte este de a avea 1024 de instrucțiuni MOVE una după cealaltă (sau 1024 de perechi încărcare-memorare). Rutina de copiere este critică într-o asemenea măsură, încât, dacă nu există altă modalitate de a păcăli compilatorul ca să producă cu precizie codul optimal, ea ar trebui scrisă de mână, cu multă atenție, direct în cod de asamblare.

6.7 REZUMAT

Nivelul transport reprezintă cheia pentru înțelegerea protocoalelor organizate pe niveluri. El furnizează diferite servicii, cel mai important dintre acestea fiind fluxul de octeți capăt-la-capăt de la emițător la receptor, fiabil și orientat pe conexiuni. El este accesat prin primitive de serviciu care permit stabilirea, utilizarea și eliberarea conexiunilor. O interfață de nivel de transport obișnuită este cea oferită de soclurile Berkeley.

Protocoalele de transport trebuie să fie capabile să controleze conexiunea în rețele nefiabile. Stabilirea conexiunii este complicată de existența pachetelor duplicate întârziate, care pot apărea la momente inoportune. Pentru a le face față, stabilirea conexiunii trebuie făcută prin intermediul protocoalelor cu înțelegere în trei pași. Eliberarea unei conexiuni este mai simplă decât stabilirea sa, dar este încă departe de a fi banală datorită problemei celor două armate.

Chiar și în cazul unui nivel rețea complet fiabil, nivelul transport are suficient de mult de lucru. El trebuie să controleze toate primitivele de serviciu, toate conexiunile și contoarele de timp și trebuie să aloce și să utilizeze credite.

Internetul are două protocoale de transport principale: UDP și TCP. UDP este un protocol neorientat pe conexiune care este în principal un ambalaj pentru pachetele IP, cu caracteristicile suplimentare de multiplexare și demultiplexare a proceselor multiple folosind o singura adresa de IP. UDP poate fi folosit pentru interacțiunile client-server, de exemplu, utilizând RPC. De asemenea poate fi folosit pentru construirea protocoalelor în timp real cum ar fi RTP.

Principalul protocol de transport în Internet este TCP. El oferă un flux sigur, bidirecțional de octeți. El utilizează un antet de 20 de octeți pentru toate segmentele. Segmentele pot fi fragmentate de rutere în cadrul Internet-ului, deci calculatoarele gazdă trebuie să fie pregătite să le reasambleze. S-a depus un mare efort pentru optimizarea performanțelor TCP-ului, utilizând algoritmii propuși de Nagle, Clark, Jacobson, Karn și alții. Legăturile fără fir adaugă o varietate de complicații TCP-ului.

TCP Tranzacțional este o extensie a TCP-ului care se ocupă de interacțiunile client-server cu un număr redus de pachete.

Performanțele rețelei sunt dominate în mod tipic de protocol și de costul suplimentar datorat tratării TPDU-urilor, situație care se înrăutățește la viteze mari. Protocoalele ar trebui proiectate astfel, încât să minimizeze numărul de TPDU-uri, copierile lor repetate și comutările de context. Pentru rețelele gigabit sunt de dorit protocoale simple.

6.8 PROBLEME

1. În exemplele noastre de primitive de transport din fig. 6-2, LISTEN este un apel blocant. Este acest lucru strict necesar? Dacă nu, explicați cum ar putea fi utilizată o primitivă neblocantă. Ce avantaje ar avea aceasta pentru schema descrisă în text?

2. În modelul pe care se bazează fig. 6-4 se presupune că pachetele pot fi pierdute de către nivelul rețea și trebuie deci să fie confirmate individual. Să presupunem că nivelul rețea este 100% fiabil și nu pierde pachete niciodată. Ce modificări sunt necesare (dacă sunt necesare) în fig. 6-4?

3. În ambele părți ale fig. 6-6 este un comentariu conform căruia valoarea SERVER_PORT trebuie să fie aceeași și în client și în server. De ce este acest lucru atât de important?

4. Să presupunem că pentru generarea numerelor de secvență inițiale se utilizează o schemă dirijată de ceas cu un contor de timp de 15 biți. Ceasul generează un impuls la fiecare 100 ms și durata de viață maximă a unui pachet este de 60 sec. Cât de des este necesar să aibă loc o resincronizare
a) în cel mai rău caz?
b) atunci când se consumă 240 de numere de secvență pe secundă?

5. De ce este necesar ca timpul maxim de viață al unui pachet, *T*, să fie suficient de mare pentru a acoperi nu numai dispariția pachetului, dar și a confirmării?

6. Să ne imaginăm că pentru stabilirea unei conexiuni se utilizează un protocol cu înțelegere în doi pași și nu unul cu înțelegere în trei pași. Cu alte cuvinte, al treilea mesaj nu mai este necesar. Sunt posibile interblocări în această situație? Dați un exemplu sau arătați că nu există nici o interblocare.

7. Imaginați o problemă generalizată a celor *n* armate, în care acordul dintre oricare două armate este suficient pentru victorie. Există un protocol care îi permite albastrului să câștige?

8. Să considerăm problema recuperării după defectarea unei mașini gazdă (adică fig. 6-18). Dacă intervalul dintre scrierea și trimiterea unei confirmări, sau vice-versa, poate fi făcut relativ scurt, care sunt cele mai bune strategii emițător-receptor pentru minimizarea șansei de defectare a protocolului?

9. Sunt posibile interblocările pentru entitățile transport descrise în text (fig. 6-20)?

10. Din pură curiozitate, programatorul entității transport din fig. 6-20 a decis să pună contoare în interiorul procedurii *sleep*, pentru a colecta astfel statistici despre vectorul *conn*. Între acestea se află și numerele de conexiuni din fiecare dintre cele șapte stări posibile Σn_i (i=1,…, 7). După scrierea unui program FORTRAN serios pentru a analiza datele, programatorul nostru descoperă că relația Σn_i = *MAX_CONN* pare să fie totdeauna adevărată. Există și alți invarianți care să implice doar aceste șapte variabile?

11. Ce se întâmplă dacă utilizatorul entității transport din fig. 6-20 trimite un mesaj de lungime 0? Discutați semnificația răspunsului.

12. Pentru fiecare eveniment care poate apărea în entitatea transport din fig. 6-20, spuneți dacă este sau nu ca utilizatorul să doarmă (eng.: sleep) în starea *transmisie*.

13. Discutați avantajele și dezavantajele creditelor față de protocoalele cu fereastră glisantă.

14. De ce există UDP? Nu ar fi fost suficient ca procesul utilizator să fie lăsat să trimită pachete bloc IP?

15. Se consideră un protocol simplu la nivelul aplicației, construit pe UDP, care permite unui client să recupereze un fișier de la un server la distanță aflat la o adresă bine cunoscută. Clientul trimite mai întâi o cerere cu numele fișierului, iar serverul răspunde cu o secvență de pachete de date conținând diferite părți din fișierul cerut. Pentru a asigura fiabilitate și livrare secvențială, clientul și serverul folosesc un protocol pas-cu-pas. Ignorând problema evidentă a performanței, vedeți vreo problemă la acest protocol? Gândiți-vă atent la posibilitatea terminării bruște a proceselor.

16. Un client trimite cereri de 128 de octeți către un server localizat la 100 km depărtare, printr-un cablu optic de 1 gigabit. Care este eficiența liniei în timpul apelului de procedură la distanță?

17. Să considerăm din nou situația din problema precedentă. Calculați timpul minim posibil de răspuns atât pentru linia de 1 Gbps anterioară cât și pentru o linie de 1 Mbps. Ce concluzie puteți trage?

18. Atât UDP, cât și TCP, folosesc numere de port pentru a identifica entitatea destinație când livrează mesaje. Dați două motive pentru care aceste protocoale au inventat un nou ID abstract (numere de port), în loc să folosească ID-uri de procese, care existau deja când aceste protocoale au fost proiectate.

19. Care este dimensiunea totală a MTU TCP, incluzând supraîncărcarea TCP-ului și IP-ului dar neincluzând supraîncărcarea nivelului legătură de date?

20. Fragmentarea și reasamblarea datagramelor sunt controlate de IP și sunt invizibile TCP-ului. Înseamnă acest lucru că TCP-ul nu trebuie să-și facă griji pentru datele care sosesc în ordine eronată?

21. RTP este utilizat pentru a transmite date audio de calitatea CD-ului, ceea ce înseamnă o pereche de eșantioane pe 16 biți de 44.100 de ori pe secundă, un eșantion pentru fiecare dintre canalele stereo. Câte pachete pe secundă trebuie să transmită RTP?

22. Ar fi posibil să fie plasat cod RTP în nucleul sistemului de operare, alături de cod UDP? Explicați răspunsul.

23. Un proces de pe mașina 1 a fost asociat portului *p* și un proces de pe mașina 2 a fost asociat portului *q*. Este posibil ca între cele două porturi să fie deschise mai multe conexiuni TCP în același timp?

24. În fig. 6-29 am văzut că alături de câmpul *Acknowledgement* de 32 de biți, este un bit *ACK* în al patrulea cuvânt. Acesta adaugă cu adevărat ceva? De ce da, sau de ce nu?

25. Informația utilă maximă dintr-un segment TCP este de 65495 octeți. De ce a fost ales un număr atât de straniu?

26. Descrieți două moduri de a ajunge în starea *SYN RCVD* din fig. 6-28.

27. Prezentați un dezavantaj potențial al algoritmului Nagle atunci când este utilizat într-o rețea puternic congestionată.

28. Să considerăm efectul utilizării startului lent pe o linie cu timpul circuitului dus-întors de 10 ms și fără congestie. Fereastra receptorului este de 24 Kocteți și dimensiunea maximă a segmentului este de 2 Kocteți. Cât timp trebuie să treacă înainte ca prima fereastră completă să poată fi trimisă?

29. Să presupunem că fereastra de congestie TCP este de 18 Kocteți și apare o depășire de timp. Cât de mare va fi fereastra dacă următoarele patru rafale de transmisie reușesc? Se presupune că dimensiunea maximă a segmentului este de 1 Koctet.

30. Dacă timpul circuitului TCP dus-întors, RTT, este la un moment dat 30 ms și următoarele confirmări sosesc după 26, 32 și, respectiv, 24 ms, care este noul RTT estimat folosind algoritmul Jacobson? Utilizați α=0.9.

31. O mașină TCP trimite cadre de 65535 octeți pe un canal de 1 Gbps pentru care întârzierea pe un singur sens este de 10 ms. Care este productivitatea maximă care poate fi atinsă? Care este eficiența liniei?

32. Care este cea mai mare viteză a liniei la care o mașină gazdă poate transmite pachete TCP de 1500 octeți cu un timp de viață care poate fi de maxim 120 sec fără ca numărul de secvență să se repete? Luați în considerare supraîncărcarea de la TCP, IP și Ethernet. Presupuneți că se pot transmite continuu cadrele Ethernet.

33. Într-o rețea care are dimensiunea maximă a TPDU-urilor de 128 octeți, timpul maxim de viață al unui TPDU de 30 sec și numărul de secvență de 8 biți, care este rata maximă de date per conexiune?

34. Să presupunem că măsurați timpul necesar recepționării unui TPDU. Atunci când apare o întrerupere, se citește timpul sistem în milisecunde. Când TPDU-ul este complet prelucrat, se citește din nou timpul sistem. S-au înregistrat 0 ms de 270000 de ori și 1 de 730000 de ori. Care este timpul de recepție al unui TPDU?

35. Un procesor execută instrucțiuni la o viteză de 1000 MIPS. Informația poate fi copiată câte 64 de biți odată, fiecare copiere a unui cuvânt costând zece instrucțiuni. Dacă un pachet recepționat trebuie să fie copiat de patru ori, poate sistemul să facă față unei linii de 1 Gbps? Pentru a simplifica, presupunem că toate instrucțiunile, inclusiv acelea de citire/scriere din memorie, rulează la viteza maximă de 1000 MIPS.

36. Pentru a evita problema revenirii numerelor de secvență la valori inițiale în timp ce există încă pachete vechi, s-ar putea utiliza numere de secvență pe 64 de biți. Cu toate acestea, teoretic, un cablu optic poate opera la 75 Tbps. Care este durata maximă de viață pe care trebuie să o aibă un pachet pentru ca viitoarele rețele de 75 Tbps să nu se lovească de aceeași problemă a revenirii numerelor de secvență chiar și în cazul reprezentării lor pe 64 biți? Presupuneți, ca și TCP-ul, că fiecare octet are propriul său număr de secvență.

37. Dați un avantaj al RPC pe UDP față de TCP tranzacțional. Dați un avantaj al T/TCP față de RPC.

38. În fig. 6-40(a), vedem că sunt necesare 9 pachete pentru a realiza RPC-ul. Există situații în care sunt necesare exact 10 pachete?

39. În secțiunea 6.6.5 am calculat că o linie gigabit livrează unei mașini gazdă 80000 de pachete pe secundă, permițându-i doar 6250 de instrucțiuni pentru a prelucra un pachet și lăsând doar jumătate din capacitatea procesorului pentru aplicații. Acest calcul presupune un pachet de 1500 octeți. Refaceți calculul pentru pachetele ARPANET de dimensiune de 128 octeți. În ambele cazuri, presupuneți că dimensiunile pachetelor date includ toate supraîncărcările.

40. Pentru o rețea care operează la 1 Gbps pe o distanță de 4000 km, factorul limitator nu este dat de lărgimea de bandă, ci de întârziere. Considerăm un MAN cu sursa și destinația situate în medie la 20 km una de cealaltă. La ce viteză de date întârzierea circuitului dus-întors datorată vitezei luminii egalează întârzierea de transmisie pentru un pachet de 1 Koctet?

41. Calculați produsul dintre întârziere și lățimea de bandă pentru următoarele rețele: (1) T1 (1,5 Mbps), (2) Ethernet (10 Mbps), (3) T3 (45 Mbps) și (4) STS-3 (155 Mbps). Presupuneți RTT de 100 ms. Amintiți-vă ca antetul TCP-lui are 16 biți rezervați pentru Dimensiunea Ferestrei. Care sunt implicațiile din punctul de vedere al calculelor voastre?

42. Care este produsul dintre întârziere și lățimea de bandă pentru un canal de 50 Mbps pe un satelit geostaționar? Dacă pachetele sunt toate de 1500 de octeți (incluzând supraîncărcarea), cât de mare ar trebui să fie fereastra în pachete?

43. Serverul de fișiere din fig. 6-6 este departe de a fi perfect și i-ar fi folositoare câteva îmbunătățiri. Faceți următoarele modificări:
a) Dați clientului un al treilea argument care specifică un interval de octeți.
b) Adăugați un flag –w de client care permite fișierului să fie scris pe server.

44. Modificați programul din fig. 6-20 pentru a asigura revenirea din erori. Adăugați un nou tip de pachet, *reset*, care poate ajunge doar după ce conexiunea a fost deschisă de ambele părți și n-a fost închisă de niciuna. Acest eveniment, care are loc simultan la ambele capete ale conexiunii, indică faptul că orice pachet care era în tranzit a fost sau distrus, sau livrat, în orice caz el nemaiaflându-se în subrețea.

45. Scrieți un program care simulează controlul tampoanelor într-o entitate transport, utilizând un flux de control cu fereastră glisantă și nu un control al fluxului cu credite, ca în fig. 6-20. Lăsați procesele de pe nivelul superior să deschidă conexiuni, să trimită date și să închidă conexiuni în mod aleatoriu. Pentru a păstra programul cât mai simplu, faceți ca toată informația să călătorească doar de la mașina A la mașina B și deloc în sens invers. Experimentați cu diferite strategii de alocare a tampoanelor la nivelul mașinii B, ca, de exemplu, tampoane dedicate unei anume conexiuni față de tampoane preluate dintr-un depozit comun și măsurați productivitatea totală atinsă în ambele cazuri.

46. Proiectați și implementați un sistem de discuții care permite mai multor grupuri de utilizatori să discute. Coordonatorul discuțiilor se află la o adresă de rețea bine cunoscută, folosește UDP pentru comunicarea cu clienții de discuții, setează serverele de discuții pentru fiecare sesiune de discuții, și menține un director de sesiuni de discuții. Este un server de discuții pentru fiecare sesiune de discuții. Un server de discuții folosește TCP pentru comunicație cu clienți. Un client de discuții permite utilizatorilor să pornească, să intre sau să iasă dintr-o sesiune de discuții. Proiectați și implementați codul pentru coordonator, server și client.