sítését. Ha az ügyfél egy cellát túl korán küld el, érvényteleníti-e ez a szerződést? Ha a szolgáltató nem tudja teljesíteni az egyik minőségi paraméterét egy 1 ms-ig, perelhet-e ezért az ügyfél? A szerződésnek ezt a részét a felek megtárgyalják, és azt mondja meg, milyen szigorúan kényszerítő erejű az első két rész.

Az ATM-ben és az Internetben a szolgálat minőségére vonatkozó modellek némileg eltérnek, s ez a megvalósításaikban is megmutatkozik. Az ATM modell szigorúan összeköttetésekre alapoz, míg az Internet modell datagramokat és folyamokat használ (mint az RSVP). Ezen két modell összehasonlítását adják meg (Crowcroft és mások, 1995).

5.6.6. Forgalomformálás és forgalmi politika

A szolgálat minősége paramétereinek használatára és kikényszerítésére irányuló mechanizmus (részben) egy sajátságos algoritmuson, az **eredeti cellasebesség algoritmusán (Generic Cell Rate Algorithm, GCRA)** alapul. Ez úgy működik, hogy minden egyes cellát leellenőriz, hogy az megfelel-e a virtuális áramköre paramétereinek.

A GCRA-nak két paramétere van. Ezek a maximálisan megengedett érkezési sebességet (a *PCR*-t) és az itt eltűrhető szórás mértékét (a *CDVT*-t) specifikálják. A *PCR* reciproka, T = 1/*PCR*, a cellák érkezése közti minimális idő, ahogy az 5.73.(a) ábra mutatja. Ha az ügyfél beleegyezik, hogy ne küldjön 100 000 cellánál többet másodpercenként, akkor T = 10 μs. A maximális esetben pontosan 10 μs-ként érkezik egy cella. Hogy elkerüljük a nagyon kis számokat, mikroszekundomokkal fogunk dolgozni, de mivel mindegyik paraméter valós szám, az időegység nem számít.

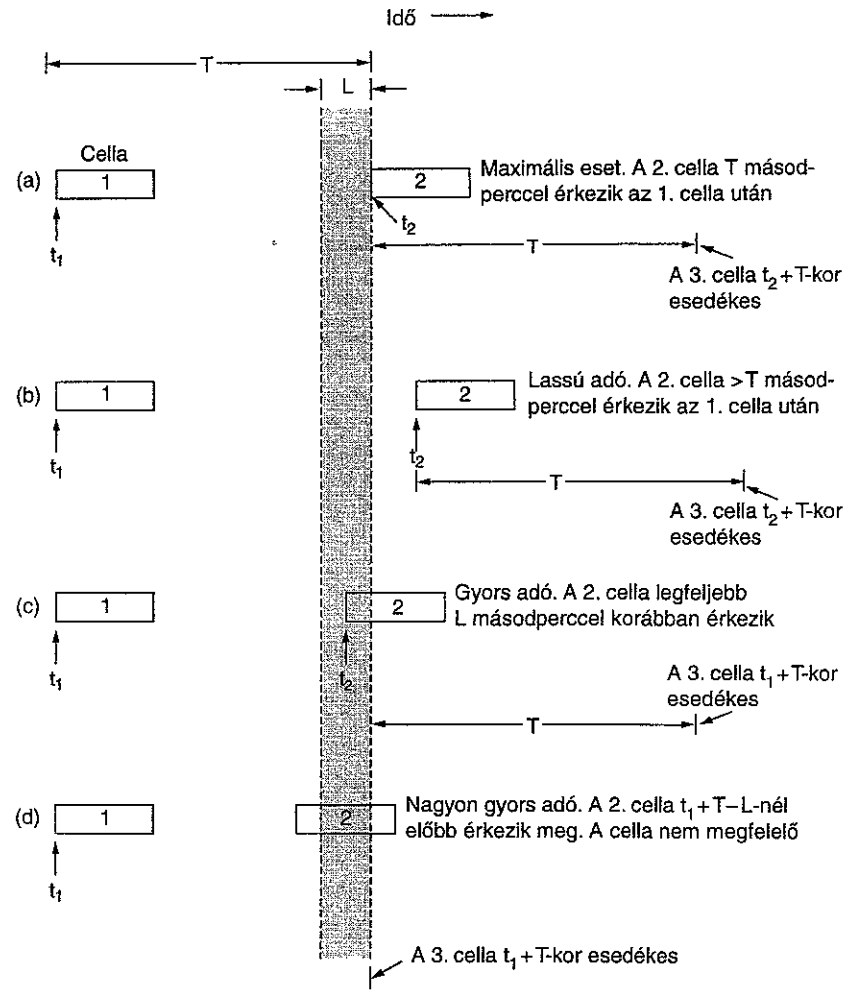
Az adó az egymás után következő cellák között mindig kihagyhat T-nél többet, ahogy az 5.73.(b) ábra mutatja. Minden olyan cella megfelelő, amely több mint T μs-mal az előző után érkezik.

A probléma azoknál az adóknál merül fel, amelyek nem szokták megvárni a startjelet, mint ahogy az 5.73.(c) és (d) ábrán látszik. Ha egy cella kicsit korábban érkezik ($t_1 + T - L$-nél korábban vagy pont akkor), akkor az megfelelő, de a következő cellát továbbra is $t_1 + 2T$-nél várják (nem pedig $t_2 + T$-nél), hogy az adót meggátolják abban, hogy minden cellát L μs-mal korábban adjon, és ezáltal megnövelje a csúcs cellasebességet.

Ha egy cella több mint L μs-mal korábban érkezik, nem megfelelőnek nyilvánítják. A nem megfelelő cellák kezelése a szolgáltatóra van bízva. Néhány szolgáltató egyszerűen eldobja azokat; mások megtartják, de a *CLP* bitet beállítják, és ezáltal alacsony prioritásúnak tüntetik fel azokat, hogy a kapcsolóknak lehetőségük legyen torlódás esetén először a nem igazodó cellákat eldobni. A *CLP* bit használata az 5.69. ábra különböző szolgálati osztályainál is eltérhet.

Most gondoljuk meg, mi történik, ha egy adó egy kicsit csalni próbál, ahogy az 5.74.(a) ábrán látszik. Ahelyett, hogy várna T ideig a 2. cella elküldésével, az adó egy egészen kicsivel korábban viszi át, T–ε-kor, ahol mondjuk ε = 0,3L. Ezt a cellát gond nélkül elfogadják.

Most az adó a 3. cellát viszi át, megint az előző cella után T–ε-nal, vagyis, T–2ε-kor. Ezt megint elfogadják. Viszont minden következő cella közelebb és közelebb araszol a végzetes $T - L$ korláthoz. Ebben az esetben, az 5. cella $T - 4\varepsilon$-kor (T – 1,2L-kor) érkezik meg, amely túl korai, így az 5. cellát nem megfelelőnek nyilvánítják és a hálózati interfész eldobja azt.



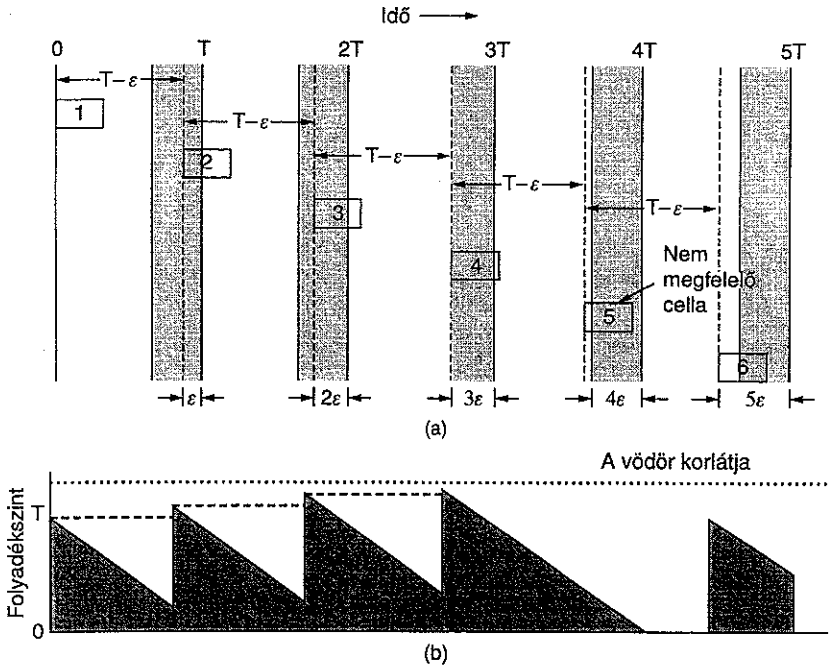
5.73. ábra. *Az eredeti cellasebesség algoritmusa*

Amikor ilyen szemszögből nézzük, akkor a GCRA algoritmust egy **virtuális ütemező algoritmusnak (virtual scheduling algorithm)** nevezzük. Viszont másfelől nézve megegyezik egy lyukas vödör algoritmussal, ahogy az 5.74.(b) ábrán látszik. Képzeljük el, hogy minden megfelelő cella T egység folyadékot önt egy lyukas vödörbe. A vödör 1 egység/s sebességgel csorgatja ki a folyadékot, így ezután T μs-

mal már mind el is tűnt. Ha a cellák pontosan T µs-ként érkeznek, mindegyik érkező cella (éppen) üresen találja a vödröt, és T egység folyadékkal újratölti. Ezáltal a folyadékszint T-re emelkedik, amikor egy cella érkezik, és lineárisan csökken, amíg el nem éri a nullát. Ezt a helyzetet az 5.74.(b) ábra 0 és T közti része mutatja.

Mivel a folyadék az idővel egyenes arányban szivárog ki, egy cella érkezése utáni t időpontban a megmaradt folyadék mennyisége $T - t$. Mikor a 2. cella $T - \varepsilon$-kor megérkezik, akkor még mindig van a vödörben ε egységnyi folyadék. Az új cella hozzáadása ezt az értéket $T + \varepsilon$-ra emeli. Hasonlóan akkor, amikor a 3. cella megérkezik, 2ε egység maradt még a vödörben, így az új cella a folyadékszintet $T + 2\varepsilon$ -ra növeli. Amikor a 4. cella megérkezik, ez $T + 3\varepsilon$ -ra emelkedik.

Ha ez végtelen ideig így megy, akkor valamelyik cella a szintet a vödör kapacitása fölé fogja emelni, és ezáltal azt vissza fogják utasítani. Hogy lássuk, melyik az, számoljuk most ki a vödör kapacitását. Szándékunk szerint a lyukas vödör algoritmus ugyanazt az eredményt adja, mint ami az 5.74.(a) ábrán látható, így akkor akarjuk, hogy túlcsordulás következzen be, amikor egy cella L µs-mal korábban érkezik. Ha a megmaradt folyadék L µs alatt szivárog ki, akkor a folyadék mennyisége L kell legyen, mivel a kiszivárgási sebesség 1 egység/µs. Ezért a vödör kapacitása szándékaink szerint $T + L$ lesz, hogy bármely L µs-nál többet siető cellát visszautasítsanak a vödör



5.74. ábra. *(a) Egy csalni próbáló adó. (b) Ugyanez a cellaérkezési minta, de most egy lyukas vödör szemszögéből nézve*

túlcsordulása miatt. Az 5.74.(b) ábrán, amikor az 5. cella megérkezik, a T egység hozzáadása a már jelenlevő 4ε egység folyadékhoz a vödör szintjét $T + 4\varepsilon$-ra növeli. Mivel ebben a példában $\varepsilon = 0{,}3$-mat használunk, a vödör szintje $T + 1{,}2L$-re emelkedne az 5. cella hozzáadásával, így a cellát visszautasítják, nem kerül be újabb folyadék, és a vödör valóban kiürül.

Egy adott T-nél, ha L-et nagyon kicsire vesszük, a vödör kapacitása alig lesz nagyobb, mint T, így minden cellát nagyon egyenletes időközönként kell elküldeni. Viszont, ha L-t egy T-nél sokkal nagyobb értékre emeljük, akkor a vödör több cellát is tartalmazhat, mivel $T + L \gg T$. Ez azt jelenti, hogy az adó közvetlenül egymás után vihet át egy több cellából álló löketet is, és még mindig el fogják azokat fogadni.

Könnyen kiszámíthatjuk a megfelelő cellák azon N számát, amelyeket $PCR = 1/T$ csúcs cellasebességnél közvetlenül egymás után átvihetünk. Egy N cellából álló löket alatt a vödörhöz hozzáadott teljes folyadékmennyiség NT, mivel minden cella T-t ad hozzá. Viszont a maximális löket alatt a folyadék 1 egység/időintervallum sebességgel szivárog ki a vödörből. Nevezzük a cellaátviteli időt δ időegységnek. Vegyük észre, hogy $\delta \leq T$, mivel teljesen lehetséges egy 155,52 Mb/s sebességű vonalon levő adó számára, hogy megegyezzen abban, hogy, nem küld több mint 100 000 cellát másodpercenként, amikor is $\delta = 2{,}73$ µs, és $T = 10$ µs lesz. Az N cellából álló löket alatt a szivárgás mennyisége $(N - 1)\delta$, mert a szivárgás nem kezdődik el addig, amíg az első cella teljesen át nem ment.

Ezekből a megfigyelésekből láthatjuk, hogy a folyadék felhalmozódó növekedése a vödörben a maximális löket alatt $NT - (N - 1)\delta$. A vödör kapacitása $T + L$. A két mennyiséget egyenlővé téve kapjuk, hogy:

$$NT - (N - 1)\delta = T + L$$

Ezt az egyenletet N-re megoldva kapjuk, hogy:

$$N = 1 + \frac{L}{T - \delta}.$$

Viszont ha ez a szám nem egész szám, akkor lefelé kell kerekíteni egy egész számra, hogy megelőzzük a vödör túlcsordulását. Például, ha $PCR = 100\,000$ cella/s, $\delta = 2{,}73$ µs, és $L = 50$ µs, akkor hét cellát lehet átvinni közvetlenül egymás után 155,52 Mb/s-os sebességgel anélkül, hogy a vödör megtelne. Egy nyolcadik cella már nem lenne megfelelő.

A GCRA-t rendszerint a T és L paraméterek megadásával határozzák meg. T egyszerűen PCR reciproka; az L pedig a $CDVT$. A GCRA-t arra is használják, hogy meggyőződjenek arról, hogy az átlagos cellasebesség nem lépi túl SCR-t hosszabb időtartam alatt sem.

Ebben a példában feltételeztük, hogy a cellák egyenletesen érkeznek. A valóságban ez nem igaz. Ettől függetlenül a lyukas vödör algoritmusa itt is használható. Minden cella érkezésekor egy ellenőrzés történik, hogy van-e a vödörben hely további T egység folyadék számára. Ha van, a cella megfelelő; egyébként nem az.

A GCRA azon kívül, hogy egy szabályt ad arra, hogy mely cellák megfelelőek és

melyek nem, a forgalmat is formálja, hogy a lökések egy részét eltávolítsa. Minél kisebb a *CDVT*, annál nagyobb ez a simító hatás, de annak az esélye is nagyobb, hogy nem megfelelő cellákat eldobnak. Néhány megvalósítás kombinálja a GCRA lyukas vödröt egy vezérjeles vödörrel, hogy további simítást biztosítson.

5.6.7. Torlódásvédelem

Az ATM hálózatok még forgalomformálással sem teljesítik automatikusan a forgalmi szerződésben előírt követelményeket. Például a közbenső kapcsolóknál torlódás mindig felléphet, különösen akkor, amikor több mint 350 000 cella ömlik be másodpercenként minden vonalon, és egy kapcsolónak akár 100 vonala is lehet. Következésképpen nagyon alaposan végiggondolták az ATM hálózatok a teljesítőképességével és torlódásvédelmével kapcsolatos kérdéseket. Ebben a szakaszban néhány ehhez használt megközelítést tárgyalunk. További információért lásd (Eckberg, 1992; Eckberg és mások, 1991; Hong és Suda, 1991; Jain, 1995; és Newman, 1994).

Az ATM hálózatoknak foglalkozniuk kell a hosszútávú torlódással, amelyet az okoz, hogy nagyobb forgalom jön be, mint amit a rendszer kezelni tud, és a rövid távú torlódással is, amelyet a forgalom lökései okoznak. Ennek eredményeképpen számos különböző stratégiát használnak egyszerre. Ezek közül a legfontosabbak három kategóriába esnek:

1. A belépés ellenőrzése.
2. Erőforrás-foglalás.
3. Sebesség alapú torlódásvédelem.

A továbbiakban mindegyik stratégiát sorban meg fogjuk tárgyalni.

A belépés ellenőrzése

Kis sebességű hálózatokban rendszerint megfelelő az, hogy megvárjuk a torlódás bekövetkezését, majd azután a csomagok forrását lassításra utasítjuk. Nagy sebességű hálózatokban azonban ez a megoldás sokszor gyatrán működik, mert a figyelmeztetés elküldése és a figyelmeztetés megérkezése közötti időben további több ezer csomag érkezhet meg.

Továbbá sok ATM hálózatnak vannak valós idejű forgalmat előállító forrásai, amelyek belső sebességgel állítják elő az adatokat. Nem biztos, hogy egy ilyen forrást arra lehet utasítani, hogy lassítson le. (Képzeljünk el egy új digitális telefont, rajta egy piros lámpával: amikor torlódást jeleznek, a piros lámpa kigyullad, és a beszélőnek 25 százalékkal lassabban kell beszélnie.)

Következésképpen az ATM hálózatok arra helyezik a hangsúlyt, hogy a torlódás

kialakulását eleve megakadályozzák. De mivel a CBR, VBR és UBR osztályú forgalmaknál egyáltalán nincs dinamikus torlódásvédelem, ezért itt egy gramm megelőzés megér egy kiló (vagy még inkább egy tonna) gyógyítást. A torlódás megelőzésének egyik fő eszköze a belépés ellenőrzése. Amikor egy hoszt egy új virtuális áramkört akar, le kell írnia a felkínálandó forgalmat és az elvárt szolgáltatást. Ezután a hálózat ellenőrizheti, hogy lehetséges-e ezt az összeköttetést kezelni anélkül, hogy az károsan befolyásolná a már meglevő összeköttetéseket. Esetleg több lehetséges útvonalat is meg kell vizsgálni, hogy egy olyat találjon, amely el tudja végezni a munkát. Ha nem lehet ilyen útvonalat találni, a hívást visszautasítja.

A belépés megtagadását igazságosan kell megtenni. Vajon igazságos az, hogy egy televízióműsorok tucatjai között kapcsolgató lajhár 100 szorgos hangyát is kiszoríthat, akik az e-levelüket próbálják elolvasni? Ha semmilyen szabályozást nem alkalmazunk, kevés nagy sávszélességű felhasználó komolyan akadályozhat sok kis sávszélességű felhasználót. Hogy ezt megelőzzük, a felhasználókat a használat alapján osztályokba kell sorolni. A szolgálatmegtagadás valószínűségének minden osztályra durván ugyanakkorának kell lennie (esetleg azáltal, hogy minden osztálynak saját erőforráshalmazt adunk).

Erőforrás-foglalás

A belépés ellenőrzéséhez szorosan kapcsolódik az erőforrások előre, rendszerint a hívás felépítésekor történő lefoglalása. Mivel a forgalomleíró megadja a csúcs cellasebességet, a hálózatnak megvan a lehetősége arra, hogy az út mentén elegendő erőforrást foglaljon le ahhoz, hogy ezt a sebességet kezelni tudja. A sávszélességet úgy lehet lefoglalni, hogy a LÉTREHOZÁS (SETUP) üzenet minden olyan vonalon, amelyen áthalad, megjelöl sávszélességet. Természetesen azt biztosítani kell, hogy egy adott vonalon megjelölt teljes sávszélesség kisebb legyen, mint annak a vonalnak a kapacitása. Ha a LÉTREHOZÁS (SETUP) üzenet egy olyan vonalra kerül, amelyik tele van, vissza kell lépnie, és egy másik utat kell keresnie.

A forgalomleíró nemcsak a csúcs sávszélességet, hanem az átlagos sávszélességet is tartalmazhatja. Ha például egy hoszt 100 000 cella/s csúcs sávszélességet, de csak 20 000 cella/s átlagos sávszélességet akar, akkor elvben öt ilyen áramkör nyalábolható ugyanarra a fizikai trönkre. Ezzel kapcsolatban a gond az, hogy mind az öt összeköttetés tétlen lehet egy félóráig, majd elkezdhet csúcssebességgel adni, s ez nagy cellaveszteséget okoz. Mivel a VBR forgalom statisztikusan nyalábolható, ennél a szolgálati osztálynál is felléphetnek gondok. A lehetséges megoldásokat még tanulmányozzák.

Sebesség alapú torlódásvédelem

CBR és VBR forgalom esetén rendszerint az adó nem képes lassítani, még torlódás esetén sem, a forrás belső valós idejű vagy kvázi valós idejű természetéből adódóan. Az UBR-rel senki nem törődik: ha túl sok cella van, a fölöseket egyszerűen eldobják.

Viszont ABR forgalomnál lehetséges és ésszerű a hálózat számára, hogy egy vagy

több adónak jelezzen, és megkérje őket, hogy átmenetileg lassítsanak le, amíg a hálózat magához tér. Az adónak érdeke engedelmeskedni, mivel a hálózat mindig büntetheti azzal, hogy kidobja a (fölös) celláit.

Az ATM szabvány kifejlesztése során egy kényes téma volt az, hogy a torlódást hogyan észleljék, jelezzék és szabályozzák ABR forgalom esetén. Mielőtt megvizsgálnánk az elfogadott megoldást, nézzünk meg röviden néhány olyan megoldást, amelyet elutasítottak.

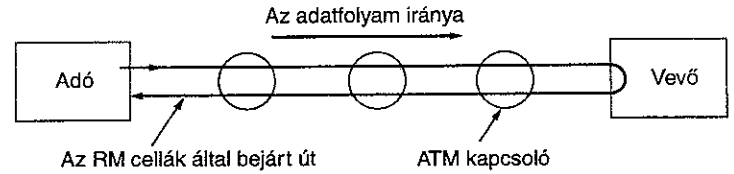
Egy javaslat szerint ahányszor csak egy adó adatok egy löketét kívánja elküldeni, először egy speciális, a szükséges sávszélességet lefoglaló cellát kellene küldenie. Miután a nyugta visszaérkezett, a löket elkezdődhetne. Ennek az előnye az, hogy a torlódás soha nem következik be, mivel a szükséges sávszélesség mindig rendelkezésre áll, amikor arra szükség van. Az ATM Forum ezt a javaslatot azért utasította el, mert egy potenciálisan hosszú késleltetés léphet fel, mielőtt egy hoszt adni kezdene.

Egy másik javaslatban a kapcsolók küldtek vissza lefojtó cellákat, ahányszor csak torlódás kezdett fellépni. Egy ilyen cella vételekor az adónak felére kellett visszavenni az aktuális cellaátviteli sebességét. Sokfajta sémát javasoltak arra, hogy hogyan nőjön meg újra a sebesség, amikor a torlódás elmúlik. Ezt a sémát azért utasították el, mert a lefojtó cellák elveszhetnek a torlódásban, és azért is, mert a séma nem tűnt igazságosnak a kis felhasználókkal szemben. Vegyünk például egy kapcsolót, amely öt felhasználó közül mindegyiktől 100 Mb/s-os folyamokat kap, és egy másiktól egy 100 kb/s-osat. Sok bizottsági tag úgy érezte, hogy nem lenne helyénvaló a 100 kb/s-os felhasználót arra utasítani, hogy 50 kb/s-ot adjon fel, mert túl nagy torlódást okoz.

Egy harmadik javaslat azt a tényt használta ki, hogy a csomaghatárokat egy bit jelöli az utolsó cellában. Itt az volt az ötlet, hogy a torlódást cellák eldobásával enyhítsük, de ezt igencsak szelektíven tegyük. A kapcsolónak végig kellett volna fésülnie a bejövő csomagfolyamot egy csomag végét keresve, majd eldobni a következő csomag minden celláját. Természetesen ezt az egy csomagot később újra kellene adni, de egy csomag *k* cellájának eldobása végül is csak egy csomagújraadáshoz vezet, és ez sokkal jobb, mintha *k* véletlenszerű csomagot dobnánk el, amely akár *k* csomagújraadáshoz is vezethet. Ezt a sémát igazságossági okokból utasították vissza, mivel a következő csomagvége jel lehet, hogy nem a kapcsolót túlterhelő adóhoz tartozik. Ezt a sémát nem is kellett szabványosítani. Bármely kapcsológyártó szabadon választhatja ki, melyik cellákat dobja el, amikor torlódás következik be.

A sok versenyző közül ketten maradtak porondon: egy hitel alapú megoldás (Kung és Morris, 1995) és egy sebesség alapú megoldás (Bonomi és Fendick, 1995). A hitel alapú megoldás tulajdonképpen egy dinamikus csúszóablakos protokoll volt. Ehhez minden kapcsolónak virtuális áramkörönként egy-egy hitelértéket kellett karbantartani, amely tulajdonképpen az azon áramkör számára fenntartott pufferek számát jelenti. Amíg minden átvitt cellához tartozik egy rá várakozó puffer, soha nem következhet be torlódás.

Az e megoldást vitató érveket a kapcsolók forgalmazói fogalmazták meg. Nem akarták mindazt a könyvelést elvégezni, amely a hitelértékek nyilvántartásához szükséges, és nem akartak ilyen sok puffert előre lefoglalni. A megkívánt többlet és veszteség mennyiségét túl nagynak ítélték, így végül a sebesség szerinti megoldást fogadták el. Ez a következőképpen működik:



5.75. ábra. *Az RM cellák által bejárt út*

Az alapmodell az, hogy minden *k* adatcella után minden egyes adó egy speciális **RM (Resource Management – erőforrás-kezelő)** cellát ad ki. Ez a cella ugyanazon az úton halad, mint az adatcellák, de az út menti kapcsolók különleges elbánásban részesítik. Amikor elér a célhoz, az megvizsgálja, frissíti, és visszaküldi az adóhoz. Az RM cellák teljes útja az 5.75. ábrán látható.

Továbbá, ezen kívül még két másik torlódásvédelmi mechanizmust biztosítanak. Először is, a túlterhelt kapcsolók spontán létrehozhatnak RM cellákat és visszaküldik azokat az adóhoz. Másodszor, a túlterhelt kapcsolók beállítják a középső *PTI* bitet az adótól a vevő felé tartó adatcellákban. De ezen eljárások közül egyik sem teljesen megbízható, mivel ezek a cellák elveszhetnek a torlódásban anélkül, hogy ezt bárki is észrevenné. Ezzel ellentétben egy RM cella elvesztét az adó észreveszi, amikor az nem tér vissza az elvárt időintervallumon belül. Mellékesen mondva, a *CLP* bitet nem használják az ABR torlódásvédelemhez.

Az ABR torlódásvédelem azon az ötleten alapszik, hogy minden adónak van egy aktuális cellasebessége, **ACR (Actual Cell Rate)**, amely *MCR* és *PCR* közé esik. Amikor torlódás lép fel, az *ACR*-t csökkentik (de nem kisebbre, mint az *MCR*). Amikor nincs torlódás, az *ACR*-t megnövelik (de nem nagyobbra, mint a *PCR*). Minden elküldött RM cella tartalmazza azt a sebességet, amivel az adó éppen adni szeretne. Ez esetleg *PCR*, esetleg annál kisebb. Ezt az értéket **ER-nek (Explicit Rate – explicit sebességnek)** nevezik. Ahogy az RM cella áthalad a vevő felé a különféle kapcsolókon, azok, amelyek torlódott állapotban vannak, csökkenthetik az *ER*-t. Egy kapcsoló sem növelheti ezt meg. A csökkentés az előreirányban vagy a visszairányban is megtörténhet. Amikor az adó visszakapja az RM cellát, láthatja, hogy mi az a maximális érték, amelyet az útbaeső összes kapcsoló elfogad. Ezután, ha szükséges, beállítja az *ACR*-t, a leglassabb kapcsolóval azonos szintre.

A középső *PTI* bitet használó torlódási mechanizmust is beleintegrálták az RM cellákba azáltal, hogy a vevőnek ezt a (legutolsó adatcellából vett) bitet minden visszaküldött RM cellába bele kell tennie. Ezt a bitet nem lehet magából az RM cellából venni, mert ez a bit minden RM cellában folyton be van állítva, ahogy az 5.63. ábrán látszik.

Az ATM réteg eléggé bonyolult. Ebben a fejezetben csak a kérdések egy részét emeltük ki. További információért lásd (De Prycker, 1993; McDysan és Spohn, 1995; Minoli és Vitella, 1994; La Porta és mások, 1994). Ám az olvasót figyelmeztetjük arra, hogy mindezek a hivatkozások az ATM 3 szabványt tárgyalják, nem pedig az ATM 4 szabványt, amelyet 1996-ig nem véglegesítettek.

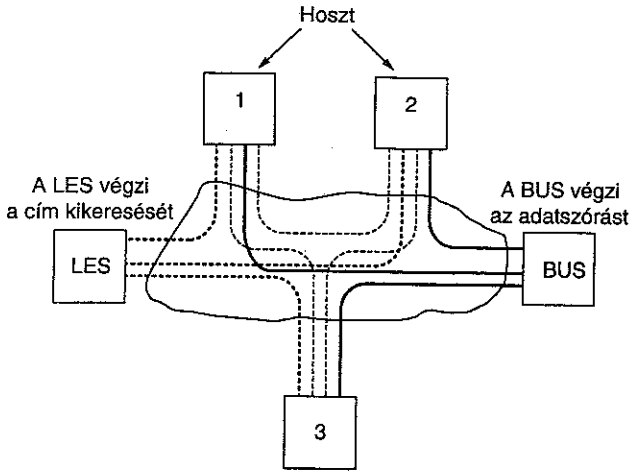
5.6.8. ATM LAN-ok

Ahogy egyre nyilvánvalóbbá válik, hogy az ITU eredeti célja, miszerint a nyilvános kapcsolt telefonhálózatot egy ATM hálózat váltsa fel, nagyon hosszú időt fog igénybe venni, az érdeklődés egyre inkább afelé fordul, hogy az ATM technológiát meglevő LAN-ok összekötésére használják fel. Ebben a megközelítésben az ATM hálózat vagy egyéni hosztokat összekötő LAN-ként, vagy több LAN-t összekötő hídként funkcionálhat. Bár mindkét elmélet érdekes, felvet néhány kihívást jelentő kérdést, amelyet a következőkben megtárgyalunk. Az ATM LAN-okról további információ található (Chao és mások, 1994; Newman, 1994; Truong és mások, 1995) könyveiben.

A legnagyobb megoldandó probléma az, hogyan biztosítsunk összeköttetés nélküli LAN szolgálatot egy összeköttetés alapú ATM hálózat felett. Egy lehetséges megoldás az, hogy bevezetünk egy összeköttetés nélküli szervert a hálózatba. Minden hoszt már induláskor felépít egy összeköttetést ezzel a szerverrel, és minden csomagot ennek küld el továbbításra. Bár ez a megoldás egyszerű, de nem használja ki az ATM hálózat teljes sávszélességét, és az összeköttetés nélküli szerver könnyen szűk keresztmetszet lehet.

Egy másik megközelítés, amelyet az ATM Forum javasolt, az 5.76. ábrán látható. Itt minden hosztnak (esetleg) van egy ATM virtuális áramköre minden másik hoszthoz. Ezeket a virtuális áramköröket dinamikusan, szükség szerint hozhatják létre és bonthatják fel, vagy ezek lehetnek állandó virtuális áramkörök is. Hogy elküldjön egy keretet, a forráshoszt először beágyazza azt egy ATM AAL üzenet adat mezejébe, és elküldi a célhoz, ugyanúgy, ahogy a kereteket egy Ethernet, vezérjeles gyűrű vagy más LAN továbbítja.

Ennek a sémának a legfőbb problémája az, hogy honnan tudjuk meg, melyik IP (vagy más hálózati rétegbeli) cím melyik virtuális áramkörhöz tartozik. Egy 802 LAN



5.76. ábra. *Az ATM LAN emuláció*

esetén ezt a problémát az ARP protokoll oldja meg, amelyben egy hoszt adatszórással elküldhet egy ilyen kérést: „Kinek az IP címe a 192.31.20.47?" Az ezt a címet használó hoszt ezután egy kétpontos üzenetet küld vissza, amely a gyorstárba kerül későbbi használatra.

Egy ATM LAN esetén ez a megoldás nem működik, mert az ATM LAN-ok nem támogatják az adatszórást. Ezt a problémát egy új szerver, a **LES (LAN Emulation Server)** bevezetésével oldják meg. Hogy kikeressen egy hálózati rétegbeli (pl. egy IP) címet, a hoszt egy csomagot (vagyis egy ARP kérést) küld a LES-nek, amely azután kikeresi a megfelelő ATM címet és visszaküldi az azt kérő gépnek. Ezt a címet azután arra lehet használni, hogy a célhosztnak beágyazott csomagokat küldjenek.

Ám ez a megoldás csak a hoszt elhelyezkedésének problémáját oldja meg. Néhány program az adatszórást vagy többesküldést az alkalmazás lényeges részeként használja. Ezen alkalmazások számára vezették be a **BUS (Broadcast/Unknown Server)** szervert. Ennek a szervernek összeköttetései vannak minden hoszthoz, és adatszórást tud szimulálni azáltal, hogy egymás után mindegyiknek egy csomagot küld. A hosztok felgyorsíthatják egy csomag ismeretlen hoszthoz történő kézbesítését azáltal, hogy a csomagot elküldik a BUS-nak adatszórásra, és (párhuzamosan) kikeresik a címet (jövőbeli használatra) a LES-t használva.

Egy ATM hálózaton IP csomagok szállítására az IETF egy, a fentihez hasonló modellt, mint a hivatalos Internet használati módot fogadott el. Ebben a modellben a LES szervert **ATMARP** szervernek hívják, de a feladatkör lényegében ugyanaz. Az IETF javaslat nem támogatja az adatszórást és a többesküldést. A modellt az RFC 1483 és az RFC 1577 írja le. Egy másik jó információforrás (Comer, 1995).

Az IETF eljárásban ATM hosztok egy halmazából egy **logikai IP alhálózatot (logical IP subnet, LIS)** hozhatunk létre. Minden LIS-nek saját ATMARP szerverje van. Lényegében egy LIS egy virtuális LAN-ként működik. Az ugyanazon a LIS-en levő hosztok közvetlenül cserélhetnek IP csomagokat, de a más LIS-eken levőknek routert kell használniuk erre. A LIS-ek létezésének oka, hogy egy LIS-en minden hosztnak (esetleg) egy megnyitott virtuális áramkörrel kell rendelkeznie az összes többi, azon a LIS-en levő hoszt felé. Azáltal, hogy a hosztok számát korlátozzák LIS-enként, a nyitott virtuális áramkörök száma kezelhető méretűvé csökkenthető.

Az ATM hálózatok egy mások haszna, hogy mint hidakat, meglevő LAN-ok összekötésére használjuk fel. Ebben a konfigurációban minden LAN-on csak egy gépnek kell ATM kapcsolódással rendelkeznie. Mint az összes átlátszó hídnak, az ATM hídnak is minden kapcsolódó LAN felé figyelnie kell, s ha szükséges, továbbítania kell a kereteket. Mivel a hidak csak MAC címeket használnak (IP címeket nem), az ATM hidaknak is fel kell építeniük egy feszítőfát, mint a 802 hidaknak.

Röviden, bár az ATM LAN emuláció egy érdekes ötlet, komoly kérdések merülnek fel a teljesítményével és az árával kapcsolatban, és nyilvánvalóan erős versenyben van a létező LAN-okkal és hidakkal, amelyek jól működnek és nagy mértékben optimalizáltak. Hogy az ATM LAN-ok és hidak valaha is felváltják-e a 802 LAN-okat és hidakat, az majd elválik.

5.7. Összefoglalás

A hálózati réteg szolgálatokat nyújt a szállítási rétegnek. Alapulhat virtuális áramkörökön vagy datagramokon. Mindkét esetben, a fő dolga, hogy a csomagok forgalomirányítását végezze a forrástól a célig. A virtuális áramkör alapú alhálózatokban akkor hoznak forgalomirányítási döntést, amikor a virtuális áramkör felépül. Datagram alapú alhálózatokban ezt minden csomagra elvégzik.

Sok forgalomirányítási algoritmus használatos a számítógép-hálózatokban. A statikus algoritmusok közé olyanok tartoznak, mint a legrövidebb út alapú forgalomirányítás, az elárasztás és a folyamalapú forgalomirányítás. A dinamikus algoritmusok közé tartoznak a távolságvektor és a kapcsolatállapot alapú forgalomirányítások. A legtöbb meglevő hálózat ezek közül valamelyiket használja. Egyéb fontos forgalomirányítási témák: a hierarchikus forgalomirányítás, forgalomirányítás mozgó hosztok számára, adatszóró forgalomirányítás és a többesküldés forgalomirányítása.

Az alhálózatokban torlódás léphet fel, amely megnöveli a csomagok késleltetését és csökkenti az átbocsátást. A hálózattervezők a torlódást megfelelő tervezéssel igyekeznek elkerülni. A módszerek között van a forgalomformálás, a folyammeghatározás és a sávszélesség-foglalás. Ha a torlódás mégis bekövetkezik, foglalkozni kell vele. Lefojtó csomagokat küldhetnek vissza, terhelést távolíthatnak el, vagy más módszereket alkalmazhatnak.

A hálózatok sokban különböznek egymástól, így amikor több hálózatot kapcsolnak össze, gondok léphetnek fel. Néha a gondokat megkerülhetjük, ha egy csomagot alagút típusú átvitellel viszünk át egy ellenséges hálózaton, de ha a forrás- és a célhálózat eltér, ez a megközelítés csődöt mond. Amikor különböző hálózatoknak különböző a maximális csomagméretük, segítségül hívhatjuk a darabolást.

Az Internetnek sokféle, a hálózati réteghez kapcsolódó protokollja van. Ezek között van az adatáviteli protokoll, az IP, de az ICMP, ARP és RARP vezérlőprotokollok is, továbbá az OSPF és BGP router protokollok. Az Internet gyorsan fogy ki az IP címekből, így kifejlesztették az IP új verzióját, az IPv6-ot.

A datagram alapú Internettől eltérően az ATM hálózatok belsőleg virtuális áramköröket használnak. Ezeket fel kell állítani, mielőtt adatot vihetnénk át, és meg kell szakítani, miután az átvitel befejeződött. A szolgálat minősége és a torlódásvédelem fő kérdések az ATM hálózatokban.

Feladatok

1. Adjon két példát olyan alkalmazásokra, amelyeknek az összeköttetés alapú szolgálat a megfelelő, majd adjon két olyan példát, amelyeknek az összeköttetés nélküli szolgálat a legjobb!

2. Előfordulhatnak-e olyan körülmények, amikor egy virtuális áramkör szolgálat a csomagokat nem sorrendhelyesen kézbesíti (vagy legalábbis ezt kellene tennie)? Indokolja!

3. A datagram alapú alhálózatok minden csomagot különálló egységként irányítanak, függetlenül az összes többitől. A virtuális áramkör alapú alhálózatoknak nem kell ezt megtenniük, mivel minden adatcsomag egy előre meghatározott útvonalat követ. Ez a megfigyelés jelenti-e azt, hogy a virtuális áramkör alapú alhálózatoknak nincs szükségük arra a képességre, hogy elszigetelt csomagokat tetszőleges forrásból tetszőleges célhoz tudjanak irányítani? Indokolja válaszát!

4. Adjon három példát olyan protokoll-paraméterekre, amelyek összeköttetés felépítésekor egyeztetés tárgyát képezhetik!

5. Fontoljuk meg a következő, a virtuális áramkör szolgálat megvalósítását érintő tervezési kérdést. Ha az alhálózaton belül virtuális áramköröket használunk, minden csomagnak 3 bájtos fejrésszel kell rendelkeznie, és minden routernek 8 bájtnyi tárhelyet kell lekötni az áramkörök azonosításához. Ha belül datagramokat használunk, 15 bájtos fejrészre van szükség, de nem kell a routerekben táblázathely. Az átviteli kapacitás 1 centbe kerül 10^6 bájtonként és átugrásonként. A routerek memóriái 1 centért vehetők meg bájtonként, és két év alatt amortizálódnak (csak munkaórákat számolva). A statisztikailag átlagos viszony 1000 másodpercig tart, amely idő alatt 200 csomagot visznek át. Az átlagos csomagnak négy átugrásra van szüksége. Melyik megvalósítás olcsóbb és mennyivel?

6. Feltételezve, hogy minden hoszt és router megfelelően működik, és mindkettő szoftvere hibáktól mentes, van-e valami, bármilyen kicsi esély arra, hogy egy csomagot rossz célhoz kézbesítenek?

7. Adjon egy egyszerű heurisztikus ötletet ahhoz, hogy két olyan utat találjunk a hálózaton keresztül egy adott forrástól egy adott célig, amelyek bármely kommunikációs vonal elvesztését túlélhetik (feltéve, hogy létezik két ilyen út)! A routereket eléggé megbízhatónak vesszük, így nem szükséges amiatt aggódni, hogy egy router összeomlik.

8. Vegyük az 5.15.(a) ábra alhálózatát. Távolságvektor alapú forgalomirányítást használunk, és a következő vektorok éppen most érkeztek meg a *C* routerhez: *B*-től: (5, 0, 8, 12, 6, 2), *D*-től: (16, 12, 6, 0, 9, 10), és *E*-től: (7, 6, 3, 9, 0, 4). A mért késleltetések sorrendben *B*-ig, *D*-ig és *E*-ig: 6, 3 és 5. Mi lesz *C* új forgalomirányító táblázata? Adja meg mind a használandó kimenő vonalat, mind a várt késleltetést!

9. Ha egy 50 routerből álló hálózatban a késleltetéseket mint 8 bites számokat jegyezzük fel, és a késleltetési vektorokat másodpercenként kétszer cseréljük ki, mekkora sávszélességet emészt fel (duplex) vonalanként az elosztott forgalomirányító algoritmus? Tegyük fel, hogy minden routernek három vonala van más routerek felé.

10. Az 5.16. ábrán a két ACF bitkészlet logikai VAGY kapcsolata minden sorban 111. Ez csak egy véletlen itt, vagy igaz minden alhálózatra minden körülmények között?

11. Ha hierarchikus forgalomirányítást akarunk 4800 routerrel csinálni, milyen tartomány- és kerületméreteket kell választani, hogy a forgalomirányító tábla mérete háromrétegű hierarchia esetén minimális legyen?

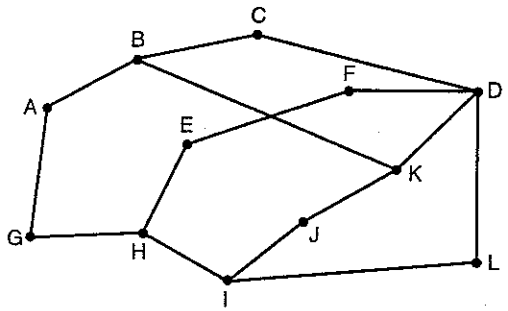
12. A szövegben azt állítottuk, hogy amikor egy mozgó hoszt nincs otthon, az otthoni LAN-jára küldött csomagokat a hazai ügynök fogja el. Egy 802.3 LAN feletti IP hálózaton hogyan viszi véghez ezt az elfogást a hazai ügynök?

13. Az 5.5. ábra alhálózatát tekintve mennyi csomagot generál egy adatszórás *B*-ből, ha a használt eljárás:

(a) A visszairányú továbbítás?

(b) A nyelőfa?

14. Számítson ki egy többesküldéses feszítőfát a *C* router számára az alábbi alhálózatban, ha a csoporttagok az *A*, *B*, *C*, *D*, *E*, *F*, *I* és *K* routerekben vannak!



15. Egy belül virtuális áramköröket használó alhálózat számára egy lehetséges torlódásvédelmi mechanizmus lehetne az, hogy egy router tartózkodik egy vett csomag nyugtázásától, amíg (1) meg nem tudja, hogy a virtuális áramkörön az utolsó átvitelét sikeresen vették és (2) nincs szabad puffere. Az egyszerűség kedvéért tételezzük fel, hogy a routerek egy megáll-és-vár protokollt használnak, és minden virtuális áramkörnek van egy kijelölt puffere a forgalom mindkét irányában. Ha *T* másodpercbe kerül egy csomagot átvinni (legyen az adat vagy nyugta), és *n* router van az útvonalon, milyen sebességgel kézbesítik a csomagot a célhosztnak? Tegyük fel, hogy az átviteli hibák ritkák és a hoszt–router összeköttetés végtelenül gyors.

16. Egy datagram alapú alhálózatban a routerek eldobhatnak csomagokat, amikor erre szükség van. Annak a valószínűsége, hogy egy router eldob egy csomagot, *p*. Vegyük azt az esetet, amikor a forráshoszt összeköttetésben áll a forrás-routerrel, amely összeköttetésben áll a cél-routerrel, azon keresztül pedig a célhoszttal. Ha bármelyik router eldob egy csomagot, a forráshosztnak végül is lejár az időzítése, és újra próbálkozik. Ha mind a hoszt-router, mind a router-router vonalakat átugrásnak vesszük, mi az átlagértéke:

(a) Az egy csomag által átvitelenként megtett ugrásoknak?

(b) A sikeresen véghezvitt átviteleknek?

(c) Az egy vett csomag által igényelt ugrásoknak?

17. Adjon egy érvet arra, hogy miért csak egy csomagot kell a lyukas vödör algoritmusnak óraütésenként átengednie, függetlenül attól, hogy milyen nagy a csomag!

18. Egy bizonyos rendszerben a lyukas vödör algoritmus bájtszámláló változatát használják. A szabály az, hogy egy 1024 bájtos csomagot, két 512 bájtos csomagot stb. lehet minden óraütéskor elküldeni. Adja meg ennek a rendszernek egy olyan súlyos korlátját, amelyet a szövegben nem említettünk!

19. Egy ATM hálózat vezérjeles vödör sémát használ a forgalomformáláshoz. Minden öt μs-ban kerül új vezérjel a vödörbe. Mi a legnagyobb fenntartható adatsebesség (a fejrészbitek leszámításával)?

20. Egy 6 Mb/s-os hálózaton elhelyezkedő számítógépet egy vezérjeles vödör szabályoz. A vezérjeles vödröt 1 Mb/s sebességgel töltik, és az kezdetben 8 megabitet tartalmaz. Milyen hosszan forgalmazhat a számítógép a teljes 6 Mb/s-os sebességen?

21. Az 5.27. ábra egy javasolt folyammeghatározás négy bemeneti jellemzőjét mutatja. Képzeljük el, hogy a maximális csomagméret 1000 bájt, a vezérjeles vödör sebessége 10 millió bájt/s, a vezérjeles vödör mérete 1 millió bájt, és a maximális átviteli sebesség 50 millió bájt/s. Milyen hosszan tarthat egy maximális sebességű löket?

22. Egy eszköz kereteket fogad el arról az Ethernetről, amelyhez csatlakozik. Minden keretből kiveszi a belsejében található csomagot, keretezési információt rak köré, és átviszi egy bérelt telefonvonalon (amely az egyetlen kapcsolata a külvilággal), amelynek a másik végén egy ugyanilyen eszköz fogadja. Ez az eszköz eltávolítja a keretezést, belehelyezi a csomagot egy vezérjeles gyűrű keretébe, és egy vezérjeles gyűrűs LAN-on keresztül átviszi egy helyi hoszthoz. Minek nevezné ezt az eszközt?

23. Szükséges-e a darabolás az egymás után kapcsolt virtuális áramkörökön alapuló összekapcsolt hálózatokban, vagy csak a datagram alapú rendszerekben?

24. Az alagút típusú átvitel egy egymás után kapcsolt virtuális áramkörökből álló alhálózaton magától értetődő: az egyik végen levő többprotokollos router egyszerűen felállít egy virtuális áramkört a másik végponthoz, és csomagokat ad át rajta. Használható az alagút típusú átvitel a datagram alapú alhálózatokban is? Ha igen, hogyan?

25. Egy *Szigorú forrás általi forgalomirányítás* opcióval ellátott IP datagramot fel kell darabolni. Gondolja, hogy az opciót minden darabba bemásolják, vagy elegendő csak az első darabba belerakni? Indokolja válaszát!

26. Tegyük fel, hogy a B osztályú címek hálózati részéhez 16 helyett 20 bitet használtunk volna. Hány B osztályú hálózat lett volna ekkor?

27. Alakítsa át a C22F1582 hexadecimális jelölésű IP címet pontokkal elválasztott decimális jelölésre.

28. Az Interneten egy B osztályú hálózatnak az alhálózati maszkja a következő: 255.255.240.0. Mi a hosztok maximális száma alhálózatonként?

29. Épp most magyarázta el az ARP protokollt a barátjának. Amikor Ön végzett, ő azt mondja: „Értem. Az ARP egy szolgálatot biztosít a hálózati rétegnek, így az adatkapcsolati réteg része." Mit mond neki?

30. Az ARP és a RARP mindketten címeket képeznek az egyik címtérből a másikba. Ebben a vonatkozásban hasonlóak. Mégis, a megvalósításaik alapvetően különbözőek. Mi az a fő dolog, amelyben különböznek?

31. Írjon le egy olyan módszert, amivel az IP darabokat össze lehetne állítani a célban.

32. A legtöbb IP datagram összeállító algoritmusnak van egy időzítője, hogy egy elveszett darab ne foglalhassa le örökké az összeállító puffereket. Tegyük fel, hogy egy datagramot négy darabra darabolnak. Az első három darab megérkezik, de az utolsót késleltetik. Végül is az időzítő lejár, és eldobjuk a már a vevő memóriájában levő három darabot. Egy kicsivel később bebotorkál a negyedik darab. Mit kell ezzel tenni?

33. A legtöbb IP router protokoll az átugrások számát használja a minimalizálandó távolságmértéknek, amikor forgalomirányítási számításokat végez. ATM hálózatoknál az átugrások száma nem túlságosan fontos. Miért nem? *Tipp:* Vessen egy pillantást a 2. fejezetre, hogy lássa, hogyan működnek az ATM kapcsolók. Tárol-és-továbbít elvet használnak?

34. Az IP-ben és az ATM-ben is az ellenőrző összeg csak a fejrészt védi, az adatot nem. Mit gondol, miért választották ezt így?

35. Egy Bostonban élő személy Minneapolisba utazik, és viszi magával a hordozható számítógépét is. Meglepetésére a minneapolisi úticéljánál levő LAN egy vezeték nélküli IP LAN, így nem kell hozzá csatlakoznia. Vajon szükséges azért még mindig végigcsinálni az egész ügyletet a hazai és idegen ügynökökkel, hogy az e-levelek és a többi forgalom helyesen érkezzen meg?

36. Az IPv6 16 bájtos címeket használ. Ha egy egymillió címből álló blokkot utalnak ki minden pikoszekundumban, meddig fognak kitartani a címek?

37. Az IPv4 fejrészben használt *Protokoll* mező nincs jelen a rögzített IPv6 fejrészben. Miért nem?

38. Amikor bevezetik az IPv6 protokollt, meg kell-e változtatni az ARP protokollt? Ha igen, akkor a változások elvi vagy technikai jellegűek?

39. Az 1. fejezetben a hálózat és a hosztok közötti interakciókat négy primitív-osztály segítségével osztályoztuk: *kérés*, *bejelentés*, *válasz* és *megerősítés*. Sorolja be az 5.65. ábra LÉTREHOZÁS (SETUP) és KAPCSOLÁS (CONNECT) üzeneteit ezekbe az osztályokba!

40. Egy ATM hálózatban egy új virtuális áramkört hozunk létre. A forrás és a cél közt három ATM kapcsoló helyezkedik el. Hány üzenetet fognak küldeni (a nyugtákat is beleértve) az áramkör létrehozásához?

41. Az 5.67. ábra létrehozásához használt logika egyszerű: mindig a legkisebb nem használt *VPI*-t utaljuk ki egy összeköttetés számára. Ha egy új virtuális áramkört igényelnek New York és Denver közt, melyik *VPI*-t fogjuk ennek kiutalni?

42. Az 5.73.(c) ábrán, ha egy cella korán érkezik, a következő még mindig $t_1 + 2T$-kor esedékes. Tegyük fel, hogy a szabály más, nevezetesen, hogy a következő cellát $t_2 + T$-kor várják, és az adó teljesen kihasználja ezt a szabályt. Milyen maximális cellasebességet lehet ekkor elérni? $T = 10$ μs és $L = 2$ μs esetére adja meg az eredeti és az új csúcs cellasebességeket is!

43. Mi a legnagyobb lökethossz egy 155,52 Mb/s-os ATM ABR összeköttetésen, amelynek *PCR* értéke 200 000 és L értéke 25 μs?

44. Írjon egy programot, amely elárasztásos forgalomirányítást szimulál. Minden csomag tartalmazzon egy számlálót, amelyet minden átugrásnál csökkentenek. Amikor a számláló eléri a nullát, a csomagot eldobják. Az idő diszkrét felosztású, és minden vonal egy csomagot kezel időintervallumonként. Készítsen három változatot a programból: minden vonal elárasztása, a bemeneti vonalon kívül minden

vonal elárasztása, és csak a legjobb k (statisztikusan választott) vonal elárasztása. Hasonlítsa össze az elárasztást a determinisztikus forgalomirányítással ($k = 1$) a késleltetés és a felhasznált sávszélesség szempontjából.

45. Írjon egy programot, amely diszkrét időosztást használva egy számítógép-hálózatot szimulál. Minden router minden várakozási sorában az első csomag egyetlen átugrást tesz meg időintervallumonként. Ha egy csomag megérkezik és nincs hely a számára, akkor eldobják és nem adják újra. Ehelyett van egy végpontok közötti protokoll, lejáró időzítésekkel és nyugtacsomagokkal kiegészítve, amely a csomagot a forrás-routertől újra elküldi. Ábrázolja a hálózat átbocsátóképességét, mint a végpontok közötti időzítési intervallum függvényét, a hibaaránnyal paraméterezve!

6. A szállítási réteg

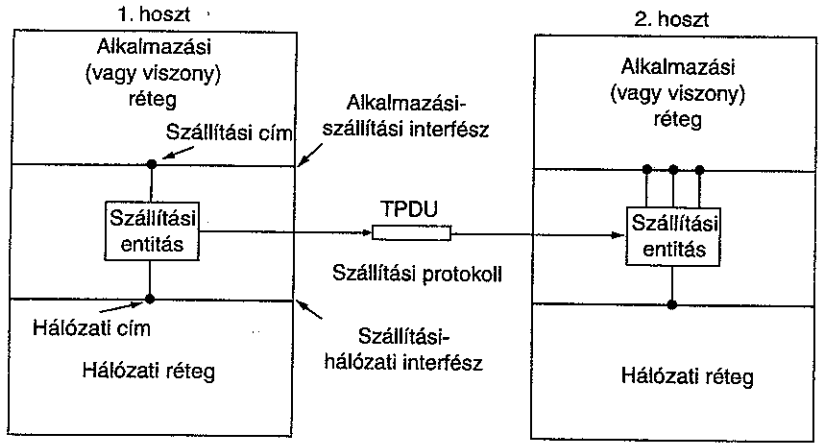
A szállítási réteg nem csak a hétrétegű architektúra egy újabb rétege, hanem az egész protokollhierarchia legfontosabb rétege. Feladata az, hogy megbízható, gazdaságos adatszállítást biztosítson a forráshoszttól a célhosztig, függetlenül magától a fizikai hálózattól vagy az aktuálisan használt kommunikációs alhálózatoktól. A szállítási réteg nélkül a rétegezett protokollkoncepciónak nem sok értelme lenne. Ebben a fejezetben a szállítási réteget fogjuk részletesen tanulmányozni, beleértve annak szolgálatait, tervezését, protokolljait és teljesítőképesség vizsgálatát.

6.1. A szállítási szolgálat

A következő néhány alfejezet a szállítási szolgálat alapjait mutatja be. Áttekintjük, hogy milyen szolgálatokat kínál az alkalmazási réteg (vagy ha egyáltalán van, a viszony réteg) felé, különös tekintettel a szolgáltatás minőségének jellemzésére. Végül megvizsgáljuk, hogy az alkalmazások hogyan érik el a szállítási szolgálatokat, azaz milyen interfész áll rendelkezésükre.

6.1.1. A felső rétegeknek nyújtott szolgálatok

A szállítási réteg legfőbb célja az, hogy hatékony, megbízható és gazdaságos szolgálatot nyújtson felhasználóinak, általában az alkalmazási rétegben futó folyamatoknak. E cél érdekében a szállítási réteg felhasználja a hálózati réteg által nyújtott szolgálatokat. A szállítási rétegen belül azt a hardver és/vagy szoftver elemet, amely a munkát végzi, **szállítási funkcionális elemnek** vagy **szállítási entitásnak (transport entity)** nevezzük. Ez lehet az operációs rendszer magjának (kernelének) része, önálló felhasználói folyamat, egy hálózati alkalmazáshoz tartozó könyvtár vagy a hálózati illesztő kártya. Néhány esetben, amikor a hálózati szolgáltató megbízható szállítási szolgálatot nyújt, a szállítási entitás az alhálózat határfelületén levő azon csomóponti gépekben helyezkedik el, melyekhez a hosztok kapcsolódnak. A hálózati, szállítási és alkalmazási réteg kapcsolatát a 6.1. ábra szemlélteti.



6.1. ábra. *A hálózati, szállítási és alkalmazási réteg*

Ahogy a hálózati szolgálatoknak két típusa van, összeköttetés alapú és összeköttetés nélküli, ugyanígy kétféle szállítási szolgálat létezik. Az összeköttetés alapú szállítási szolgálat sok tekintetben hasonló az összeköttetés alapú hálózati szolgálathoz. Mindkét esetben az összeköttetésnek három fázisa van: létesítés, adatátvitel és lebontás. A címzés és forgalomszabályozás szintén hasonló a két rétegben. Az összeköttetés nélküli szállítási szolgálat is nagyon hasonló az összeköttetés nélküli hálózati szolgálathoz.

Ezek után nyilvánvaló a kérdés: ha a két réteg szolgálatai ennyire hasonlóak, miért van szükség két külön rétegre? Miért nem elegendő egyetlen réteg? A válasz kényes, de döntő jelentőségű, és az 1.16. ábrával magyarázható. Ezen az ábrán látható, hogy a hálózati réteg a kommunikációs alhálózat része, és a hálózati szolgáltató működteti (legalábbis WAN-ok esetén). Mi történik akkor, ha a hálózat összeköttetés alapú, de megbízhatatlan szolgálatot nyújt? Feltételezzük, hogy gyakran veszít csomagokat. Mi történik, ha a csomópont időnként összeomlik?

A fellépő problémák a következők. A felhasználó nem nyúlhat bele az alhálózatba, így nem oldhatja meg az alacsony színvonalú szolgálat problémáját jobb minőségű csomóponti gép alkalmazásával, vagy az adatkapcsolati rétegben alkalmazott alaposabb hibakezeléssel. Az egyetlen lehetőség az, hogy a hálózati réteg tetejére még egy réteget kell helyezni, ami javítja a szolgáltatás minőségét. Ha a szállítási entitás egy hosszú átvitel közepén arról értesül, hogy a hálózati összeköttetés hirtelen fölbomlott, és nincsen semmilyen információ arról, hogy mi történt az elküldött adatokkal, akkor a távoli hálózati entitással új kapcsolatot építhet ki. Ezt az új kapcsolatot használva kérdést tehet fel a társentitásnak, hogy mely adatok érkeztek meg és melyek nem, és onnan folytathatja a működését, ahol abbahagyta.

Lényegében a szállítási réteg jelenléte teszi lehetővé, hogy a szállítási szolgálat megbízhatóbb legyen az alatta levő hálózati szolgálatnál. A szállítási réteg észleli az elveszett csomagokat és az adatok sérülését, és azok hatásait ellensúlyozza. Továbbá, a szállítási réteg primitívjei tervezhetők úgy, hogy függetlenek legyenek a hálózati réteg primitívjeitől, melyek hálózatonként jelentősen különbözhetnek (pl. egy összeköttetés nélküli LAN szolgálat teljesen eltérhet egy összeköttetés alapú WAN szolgálattól).

A szállítási rétegnek köszönhetően az alkalmazásokat egy szabványos primitívhalmaz segítségével úgy írhatjuk meg, hogy azok különböző hálózatokon képesek futni anélkül, hogy a különféle alhálózati interfészekkel és a megbízhatatlan átvitellel kellene törődniük. Ha minden valódi hálózat hibamentes lenne, és azonos szolgálati primitívekkel rendelkezne, akkor a szállítási rétegre valószínűleg nem lenne szükség. A valós világban azt a nagyon fontos kulcsszerepet tölti be, hogy a fölső rétegeket megkíméli az alhálózat technológiai, tervezési és megbízhatatlansági problémáitól.

Emiatt sokan elkülönítik az 1.–4. és az 5.–7. rétegeket. Az alsó négy réteget együttesen a **szállítási szolgáltatónak (transport service provider),** míg a fölső hármat a **szállítási szolgálat felhasználójának (transport service user)** tekintik. Ez a megkülönböztetés a szolgáltató és a felhasználó között jelentős befolyással van a rétegek tervezésére, és ez helyezi a szállítási réteget kulcspozícióba, mivel itt van a határvonal a megbízható átviteli szolgálat szolgáltatója és felhasználója között.

6.1.2. A szolgálat minősége

A szállítási réteg vizsgálatának másik szempontja az, hogy elsődleges feladatának a hálózati réteg által nyújtott **QoS (Quality of Service – szolgálatminőség)** javítását tekintjük. Ha a hálózati réteg tökéletes, a szállítási rétegnek könnyű dolga van. Ha viszont a hálózati szolgálat gyenge, a szállítási szolgálatnak kell áthidalni a szállítási szolgálatot igénylők elvárása és a hálózati réteg képessége közötti szakadékot.

Míg első pillantásra a szolgálat minősége bizonytalan fogalomnak tűnhet (nem könnyű mindenkit rávenni arra, hogy „jónak" tekintsen egy szolgálatot), a QoS, mint azt az 5. fejezetben láttuk, meghatározott paraméterekkel jellemezhető. A szállítási szolgálat lehetővé teheti a felhasználónak különböző paraméterek kívánt, elfogadható és minimális értékeinek megadását az összeköttetés felépítésekor. Némely paraméter összeköttetés nélküli szolgálatra is alkalmazható. A szállítási réteg feladata az, hogy megvizsgálja a kapott paramétereket, és az (általa) elérhető hálózati szolgálatok alapján eldöntse, hogy képes-e nyújtani a kért szolgálatot. Az alfejezet hátralevő részében

Összeköttetés-létesítési késleltetés
Összeköttetés-létesítési hibavalószínűség
Átbocsátóképesség
Átviteli késleltetés
Maradó hibaarány
Védelem
Prioritás
Rugalmasság

6.2. ábra. *A szállítási réteg jellegzetes szolgálatminőségi paraméterei*

néhány lehetséges QoS paramétert tárgyalunk, ezeket a 6.2. ábrán foglaljuk össze. Megjegyezzük, hogy néhány hálózat vagy protokoll az összes paramétert biztosítja. Több rendszerben mindent elkövetnek a maradó hibaarány csökkentésére, de mást nem garantálnak. Más esetben kifinomult QoS architektúrát alkalmaznak (l. Campbell és mások, 1994).

Az *összeköttetés-létesítési késleltetés (connection establishment delay)* paraméter az az idő, amely a szállítási összeköttetés igénylése és a beérkező megerősítés között eltelik. Magába foglalja a távoli entitás feldolgozási késleltetését is. Mint minden késleltetést leíró paraméternél, minél kisebb a késleltetés, annál jobb a szolgálat.

Az *összeköttetés-létesítési hibavalószínűség (connection establishment failure probability)* paraméter annak az esélyét jelenti, hogy az összeköttetés nem jön létre a maximális összeköttetés-létesítési késleltetés alatt. Ennek oka lehet pl. torlódás a hálózaton, egy táblázat betelése vagy egyéb belső probléma.

Az *átbocsátóképesség (throughput)* paraméter a másodpercenként átvitt felhasználói adatbájtok számát mutatja valamilyen időtartam alatt mérve. Az áteresztőképességet irányonként külön-külön mérik.

Az *átviteli késleltetés (transit delay)* a forráshoszt szállítási felhasználója üzenetének elküldése, a célhoszt szállítási felhasználójának üzenetvételi időpontja között eltelt időt adja meg. Mint az átbocsátóképességnél, itt is külön kezelik az egyes irányokat.

A *maradó hibaarány (residual error ratio)* az elveszett vagy sérült üzenetek számának aránya az összes elküldött üzenetek számához képest. A maradó hibaaránynak elméletileg nullának kellene lenni, mivel a szállítási réteg feladata a hálózati réteg hibáinak elfedése. A gyakorlatban fölvehet valamilyen (kis) véges értéket.

A *védelem (protection)* paraméter lehetőséget biztosít a szállítási felhasználónak, hogy a szállítási rétegtől meghatározott fokú védelmet igényeljen jogosulatlan harmadik fél (támadó) által végzett lehallgatás vagy adatmódosítás ellen.

A *prioritás (priority)* paraméter segítségével a szállítási felhasználó jelezheti a szállítási rétegnek, hogy némely összeköttetése fontosabb másoknál, és torlódás esetén a magasabb prioritású összeköttetéseket hamarabb szolgálja ki, mint az alacsony prioritásúakat.

Végül a *rugalmasság (resilience)* paraméter annak a valószínűségét mutatja, hogy maga a szállítási réteg szakítja meg az összeköttetést valamilyen belső hiba vagy torlódás miatt.

A QoS paramétereket a szállítási felhasználó rögzíti az összeköttetés létesítésekor. Mind a kívánt, mind a minimális érték megadható. Előfordulhat, hogy a QoS paramétereket látva a szállítási réteg azonnal felismeri, hogy némelyiket nem tudja teljesíteni. Ekkor anélkül, hogy egyáltalán kapcsolatba lépne a társfelhasználóval, közli a hívóval, hogy próbálkozása az összeköttetés létesítésére meghiúsult. A hibajelentésben megjelöli a hiba okát.

Más esetekben a szállítási réteg tudja, hogy nem képes elérni a kívánt célt (pl. 600 Mb/s áteresztőképesség), viszont egy alacsonyabb, de még elfogadható érték (pl. 150 Mb/s) teljesíthető. Ekkor az alacsonyabb, és a minimális elfogadható értéket az összeköttetés-kéréssel együtt elküldi a távoli gépnek. Ha a távoli gép nem képes kezelni a kívánt értéket, de egy minimumnál nagyobbat teljesíteni tudna, viszontajánlatot tehet. Ha a minimum fölött semmilyen értéket nem tud kezelni, visszautasítja az

összeköttetés-kérést. Végül a kezdeményező szállítási felhasználó értesítést kap a szállítási rétegtől az összeköttetés létesítéséről vagy elutasításáról, és siker esetén az elfogadott paraméterekről.

Ezt a folyamatot **opcióegyeztetésnek (option negotiation)** nevezik. Az egyszeri egyeztetéssel kialakult opciók az összeköttetés teljes időtartamára érvényben maradnak. Hogy elejét vegyék a felhasználók túlzott mohóságának, a tendencia az, hogy a legtöbb szolgáltató a jobb minőségű szolgáltatást magasabb áron kínálja.

6.1.3. Szállítási szolgálati primitívek

A szállítási szolgálati primitívek teszik lehetővé a szállítási felhasználóknak (pl. alkalmazói programoknak) a szállítási szolgálatok igénybevételét. Minden szállítási szolgálat saját primitívekkel rendelkezik. Ebben a fejezetben először megvizsgálunk egy egyszerű (képzeletbeli) szállítási szolgálatot, majd áttekintünk egy gyakorlatban használt rendszert.

A szállítási szolgálat hasonlít a hálózati szolgálathoz, azonban van néhány fontos eltérés. A fő különbség köztük az, hogy a hálózati szolgálat a valódi hálózatok által nyújtott szolgáltatásokat igyekszik modellezni azok gyengéivel együtt. Az igazi hálózatok csomagokat veszíthetnek, tehát a hálózati szolgálat általában nem megbízható.

Ezzel szemben az (összeköttetés alapú) szállítási szolgálat megbízható. Természetesen a valódi hálózatok nem hibamentesek, de pontosan a szállítási réteg feladata az, hogy egy nem megbízható hálózatra épülve megbízható szolgálatot nyújtson.

Vegyünk például két olyan folyamatot, amelyek a UNIX-ban csövekkel vannak összekötve. Ezek azt feltételezik, hogy a köztük levő összeköttetés tökéletes. Hallani sem akarnak nyugtázásokról, elvesztett csomagokról, torlódásról, vagy más, ehhez hasonló problémáról. Egy 100 százalékosan megbízható összeköttetést akarnak használni. Az *A* folyamat beteszi az adatokat a cső egyik végén, a *B* folyamat kiveszi a másik végén. Ez a lényege a szállítási szolgálatnak: elrejteni a hálózati szolgálat hiányosságait, hogy az alkalmazási folyamatok egy hibamentes bitfolyamot feltételezhessenek.

Emellett a szállítási réteg biztosíthat még megbízhatatlan (datagram) szolgálatot is, de viszonylag keveset kell erről mondani, így ebben a fejezetben az összeköttetés alapú szállítási szolgálatra koncentrálunk.

Egy másik különbség a hálózati szolgálat és a szállítási szolgálat között a szolgálat felhasználóinak köre. A hálózati szolgálatot csak a szállítási entitások használják. Kevesen írják meg a saját szállítási entitásaikat, ezért kevés program látja a csupasz hálózati szolgálatot. Ezzel ellentétben viszont sok alkalmazás (ezzel együtt programozó) használja a szállítási primitíveket, ezért a szállítási szolgálatnak kényelmesnek és könnyen használhatónak kell lennie.

A 6.3. ábrán bemutatunk öt lehetséges szállítási primitívet. Ez a szállítási szolgálat csak egy puszta váz, de ízelítőt ad egy összeköttetés alapú szállítási interfész lényeges feladataiból. Lehetővé teszi a felhasználói programoknak összeköttetések létesítését, használatát és lebontását, ami a legtöbb alkalmazásnak elegendő is.

Hogy lássunk egy példát a primitívek működésére, vegyünk egy alkalmazást egy szerverrel és több távoli klienssel. Először a szerver egy LISTEN (FIGYELÉS) primitívet

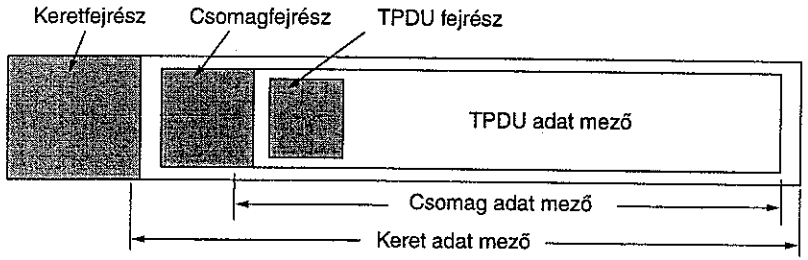
Primitív	Elküldött TPDU	Jelentés
LISTEN	(nincs)	Vár, amíg egy folyamat kapcsolódni nem próbál
CONNECT	CONNECTION REQ.	Összeköttetést próbál létrehozni
SEND	DATA	Adatot küld
RECEIVE	(nincs)	Vár, amíg adat (DATA TPDU) nem érkezik
DISCONNECT	DISCONNECTION REQ.	Ez az oldal bontani kívánja az összeköttetést

6.3. ábra. *Egy egyszerű szállítási szolgálat primitívjei*

hajt végre, tipikusan egy könyvtári függvény hívással, ami rendszerhívást eredményez, hogy a szerver egy kliens jelentkezéséig blokkolódjon. Amikor egy kliens beszélni akar a szerverrel, egy CONNECT (KAPCSOLÁS) primitívet hajt végre. A szállítási entitás ezt úgy valósítja meg, hogy a hívót blokkolja, és egy csomag adat mezejébe ágyazott szállítási üzenetet küld a szerver szállítási entitása részére.

Itt egy rövid terminológiai kitérőt kell tenni. Jobb híján az esetlen **TPDU (Transport Protocol Data Unit – szállítási protokoll adatelem)** elnevezést kényszerülünk használni szállítási entitások közötti üzenetekre. Ezek a TPDU-k (melyeket a szállítási réteg küld és fogad) csomagokba (amiket a hálózati réteg használ) vannak beágyazva. A csomagok viszont (adatkapcsolati réteg által kezelt) keretekben (frame) foglalnak helyet. Amikor egy keret megérkezik, az adatkapcsolati réteg földolgozza a keret fejrészét, és a keret adat mezejének tartalmát továbbadja a hálózati entitásnak. A hálózati entitás földolgozza a csomag fejrészét, és az adat mező tartalmát átadja a szállítási entitásnak. Ezt a beágyazott struktúrát szemlélteti a 6.4. ábra.

Visszatérve a kliens-szerver példához, a kliens CONNECT hívása hatására a szállítási entitás CONNECTION REQUEST (ÖSSZEKÖTTETÉS-KÉRÉS) TPDU-t küld a szerver felé. Amikor az megérkezik, a szállítási entitás meggyőződik arról, hogy a szerver LISTEN hívásban várakozik (azaz kész kéréseket kiszolgálni). Ekkor megszünteti a szerver blokkolását, és CONNECTION ACCEPTED (összekötetés kérés elfogadva) TPDU-t küld vissza a kliensnek. Amint a TPDU megérkezik, a kliens blokkolása is feloldódik, így az összeköttetés létrejön.

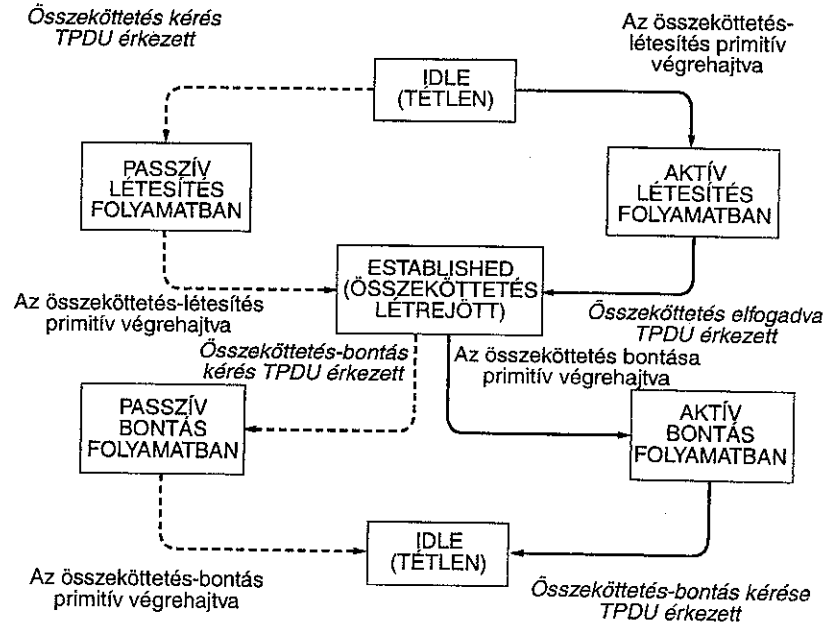


6.4. ábra. *A TPDU-k, csomagok és keretek beágyazása*

Ekkor megkezdődhet az adatátvitel a SEND és RECEIVE (ADÁS és VÉTEL) primitívek segítségével. A legegyszerűbb esetben bármelyik fél végrehajthat egy (blokkoló) RECEIVE hívást, hogy várakozzon a partner által (SEND primitívvel) küldött adatra. Amikor a TPDU megérkezik, a fogadó blokkolása megszűnik, földolgozza a kapott adatot, és választ küld. Amíg mindkét fél nyomon tudja követni, hogy ki mikor következik, ez a rendszer jól működik.

Megjegyezzük, hogy a hálózati rétegben még egy egyszerű egyirányú adatforgalom is jóval bonyolultabb, mint a szállítási rétegben. Minden adatcsomagot nyugtáznak, sőt a vezérlő TPDU-kat hordozó csomagokra is érkezik közvetett vagy közvetlen nyugtázás. Ezeket a nyugtázásokat a szállítási entitások kezelik a hálózati rétegbeli protokollok segítségével, és a szállítási felhasználók számára ezek nem láthatók. Hasonlóan, a szállítási entitásoknak kell foglalkozniuk az időzítésekkel és az ismétlésekkel. Ezen mechanizmusokból szintén semmit sem látnak a szállítási felhasználók. Számukra az összeköttetés egy megbízható csővezeték, azaz: amit egy felhasználó a cső egyik végén betölt, az a másik végén változatlanul megjelenik. A bonyolultság elrejtésének a képessége miatt a rétegezett protokollok nagyon hatékony eszköznek bizonyulnak.

Amikor egy összeköttetésre többé nincs szükség, azt le kell bontani, hogy ne foglaljon fölöslegesen táblahelyet a két szállítási entitáson belül. A bontásnak két változa-



6.5. ábra. *Egyszerű összeköttetés-kezelés állapotdiagramja. A dőlt betűvel szedett állapotátmeneteket beérkező csomagok váltják ki. A folytonos nyilak a kliens, a szaggatott nyilak a szerver állapotátmeneteit mutatják*

ta van: *aszimmetrikus* és *szimmetrikus*. Az aszimmetrikus esetben valamelyik szállítási felhasználó kiad egy DISCONNECT primitívet, aminek hatására a szállítási entitás egy DISCONNECT (ÖSSZEKÖTTETÉS-BONTÁS) TPDU-t küld a távoli szállítási entitásnak. A TPDU megérkezésekor az összeköttetés lebomlik.

A szimmetrikus esetben mindkét irányt külön, a másiktól függetlenül zárják le. Amikor az egyik fél DISCONNECT hívást kezdeményez, az azt jelenti, hogy nincs több elküldenivaló adata, de továbbra is hajlandó partnere adatait fogadni. Ebben a modellben az összekötetés akkor ér véget, amikor mindkét fél végrehajtotta a DISCONNECT primitívet.

Ezen egyszerű primitíveken alapuló összeköttetés-létesítés és -bontás állapotdiagramja látható a 6.5. ábrán. Minden állapotátmenetet valamilyen esemény vált ki: vagy egy, a helyi felhasználó által végrehajtott primitív, vagy egy beérkező csomag. Az egyszerűség kedvéért föltételezzük, hogy minden TPDU nyugtázása külön történik. Feltesszük továbbá, hogy szimmetrikus összeköttetés-bontást modellezünk úgy, hogy a kliens kezdeményez. Megjegyzendő, hogy az alábbi ábra meglehetősen elnagyolt. Később megvizsgálunk egy ennél valósághűbb modellt is.

Berkeley TCP primitívek

Vizsgáljunk meg röviden egy másik szállítási primitívkészletet, a Berkeley UNIX-ban használt TCP socket primitíveket (6.6. ábra). Nagy vonalakban követik az első példában bemutatott modellt, de több lehetőséget és rugalmasságot nyújtanak. Itt nem térünk ki a megfelelő TPDU-kra, annak tárgyalására a TCP tanulmányozása után kerül sor e fejezet későbbi részében.

Az első négy fölsorolt primitívet az ábrán látható sorrendben hajtja végre a szerver. A SOCKET primitív új végpontot hoz létre, és táblahelyet foglal le a szállítási entitásban. A hívás paraméterei rögzítik a használni kívánt címzési formát, a szolgálat típusát (pl. megbízható bitfolyam) és a protokollt. A sikeres SOCKET hívás közönséges állományleíróval tér vissza, amit a további hívások használnak, éppúgy, mint az OPEN rendszerhívásnál.

Primitív	Jelentés
SOCKET	Új kommunikációs végpont létrehozása
BIND	Helyi cím hozzárendelése a csatlakozóhoz
LISTEN	Összeköttetés-elfogadási szándék bejelentése, várakozási sor hosszának megadása
ACCEPT	Hívó blokkolása összeköttetés-létesítési kísérletig
CONNECT	Próbálkozás összeköttetés-létesítésre
SEND	Adatküldés az összeköttetésen keresztül
RECEIVE	Adatfogadás az összeköttetésről
CLOSE	Összeköttetés bontása

6.6. ábra. *TCP socket primitívek*

Az újonnan létrehozott csatlakozóknak (socket) nincs címük, a hozzárendelést a BIND primitív végzi. Amint a szerver címet rendelt a végponthoz, távoli kliensek csatlakozhatnak hozzá. Annak oka, hogy a cím megadása nem a SOCKET hívással történik az, hogy vannak olyan folyamatok, amelyek számára fontosak a címek (pl. évek óta ugyanazt a címet használják, és ez a cím mindenki által ismert), míg mások számára a cím megválasztása közömbös.

Ezt követi a LISTEN hívás, amely a beérkező hívások várakozási sorának foglal helyet arra az esetre, amikor a szerverhez egy időben több kliens is kapcsolódni kíván. Ellentétben az első példában használt LISTEN-nel, a TCP modellben a LISTEN nem blokkoló hívás.

A szerver ACCEPT primitívet hajt végre ahhoz, hogy blokkolja magát egy bejövő összeköttetés-kérésig. Amikor egy összeköttetést kérő TPDU érkezik, a transzportentitás az eredetivel azonos tulajdonságokkal rendelkező új végpontot hoz létre, és hozzárendel egy állományleírót. A szerver ekkor új folyamatot vagy szálat indít az új végponton létrejövő kapcsolat kezelésére, és tovább várja a következő kérést az eredeti végponton.

Most vegyük szemügyre a kliens oldalt. Itt ugyancsak egy végpontot kell először létrehozni a SOCKET primitív segítségével, de BIND hívás nem szükséges, mivel a használt cím nem érdekli a szervert. A CONNECT primitív blokkolja a hívót, és belekezd az összeköttetés-létesítési folyamatba. Amikor ezt befejezi (azaz a megfelelő TPDU megérkezett a szervertől), a kliens folyamat blokkolása megszűnik, és az összeköttetés létrejön. Ekkor mindkét fél a SEND és RECEIVE primitív segítségével küldhet és fogadhat adatokat a duplex összeköttetésen keresztül.

Az összeköttetés bontása szimmetrikus. Amikor mindkét fél végrehajtotta a CLOSE primitívet, az összeköttetés megszűnik.

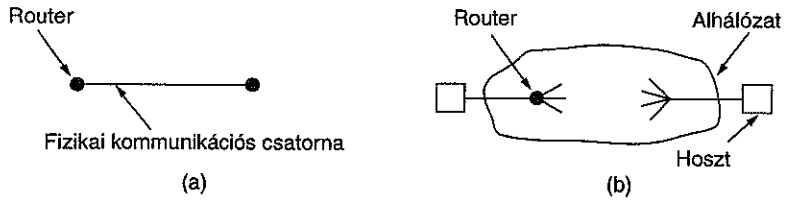
6.2. A szállítási protokollok elemei

A szállítási szolgálatot egy, a szállítási entitások között használt **szállítási protokoll** valósítja meg. Némely tekintetben a szállítási protokollok az adatkapcsolati protokollokra emlékeztetnek, amelyeket a 3. fejezetben tanulmányoztuk részletesen. Mindkettőnek többek között hibakezelést, sorszámozást és forgalomszabályozást kell végeznie.

Ugyanakkor jelentős eltérések is vannak a kettő között. A különbségek fő oka abban az alapvetően eltérő működési környezetben rejlik, melyben a két protokoll működik. Ezt a 6.7. ábrán láthatjuk. Az adatkapcsolati rétegben a két csomópont közvetlenül egy fizikai csatornán keresztül kommunikál, míg a szállítási rétegben a fizikai csatorna helyett egy egész alhálózat szerepel. Ez a különbség fontos hatással van a protokollokra.

Egyrészt az adatkapcsolati rétegben a routernek nem kell kijelölni, hogy melyik másik routerrel kíván kommunikálni – minden kimenő vonal egyértelműen azonosít egy adott routert. A szállítási rétegben a cél explicit címzése kötelező.

Másrészt amikor egy folyamat a 6.7.(a) ábrán látható vezetéken összeköttetést akar létesíteni, egyszerű dolga van: a másik végpont mindig jelen van (hacsak el nem rom-



6.7. ábra. *(a) Az adatkapcsolati réteg környezete. (b) Szállítási réteg környezete*

lott, amikor is nincs jelen). Akármelyik eset következik is be, nincs sok tennivaló. A szállítási rétegben, mint látni fogjuk, a kezdeti összeköttetés-létesítés jóval bonyolultabb.

Egy másik nagyon bosszantó különbség az adatkapcsolati réteg és a szállítási réteg között az alhálózat potenciális adattároló képessége. Ha egy router elküld egy keretet, az vagy megérkezik, vagy elvész, de nem fog bolyongani egy darabig, elrejtőzni a világ egy távoli sarkába, majd hirtelen egy váratlan pillanatban fölbukkanni mondjuk 30 másodperc múlva. Ha az alhálózat a belső forgalmat datagramokkal és adaptív forgalomszabályozással valósítja meg, nem elhanyagolható annak a valószínűsége, hogy a csomagot valamelyik csomópont tárolja pár másodpercig, és csak ezután kézbesíti. Az alhálózat adattároló képességének következménye néha katasztrofális lehet, és speciális protokollok használatát teszi szükségessé.

Az utolsó különbség az adatkapcsolati és a szállítási réteg között inkább mennyiségi, mint minőségi eltérés. Pufferelés és forgalomszabályozás mindkét esetben szükséges, de a szállítási rétegben jelenlevő nagy és változó számú összeköttetés eltérő megközelítést igényel, mint amit az adatkapcsolati rétegben használtunk. Néhány, a 3. fejezetben tárgyalt protokoll rögzített számú puffert rendel minden vonalhoz, így a beérkező keretek számára mindig van szabad puffer. A szállítási rétegben kezelendő nagyszámú összeköttetés láttán máris kevésbé vonzó ötlet mindegyiknek számos saját puffert lefoglalni. A következő alfejezetekben többek között ezekkel a fontos problémákkal fogunk foglalkozni.

6.2.1. Címzés

Amikor egy alkalmazási folyamat összeköttetést kíván létesíteni egy távoli társával, ki kell jelölnie, hogy melyikhez akar kapcsolódni. (Összeköttetés nélküli szállítási szolgálatnál ugyanez a probléma: kinek küldje az egyes üzeneteket?) Az általában használt módszer szerint szállítási címeket definiálunk, amelyeken a folyamatok az összeköttetés-kéréseket figyelhetik. Az Interneten ezek a végpontok az (IP cím, helyi port kettősök) ATM hálózatokon az AAL – SAP párok. Mi a semleges **TSAP (Transport Service Access Point – szállítási szolgálat elérési pont)** elnevezést fogjuk használni. A hálózati rétegben az ennek megfelelő végpont (azaz hálózati rétegbeli cím) neve NSAP. Az IP cím példa a NSAP-ra.

A 6.8. ábrán láthatjuk az NSAP, TSAP hálózati összeköttetés és szállítási összeköttetés közti összefüggést összeköttetés alapú alhálózat (pl. ATM) esetén. Jegyezzük

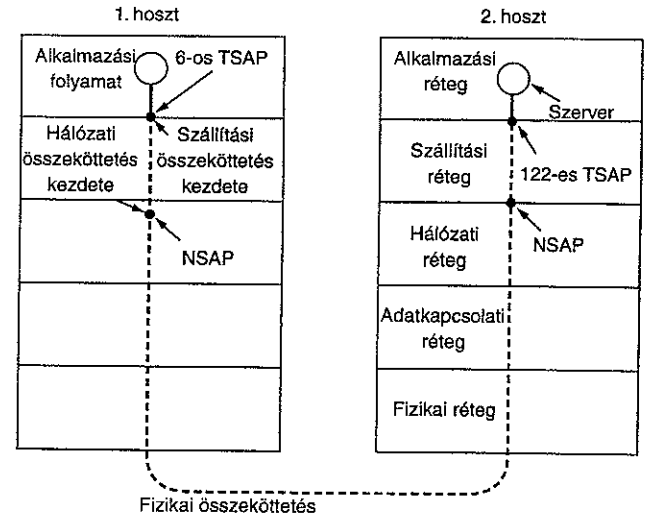
meg, hogy a szállítási entitás rendszerint támogatja a többszörös TSAP-okat. Néhány hálózatban többszörös NSAP-ok is létezhetnek, más hálózatokban egy gépen csak egyetlen NSAP lehetséges (pl. IP cím). Egy lehetséges forgatókönyv szállítási összeköttetés felépítésére összeköttetés alapú hálózati réteg fölött a következő:

1. A 2. hoszton egy, az aktuális időt szolgáltató szerver csatlakozik a 122-es TSAP-hoz, és hívás érkezésére várakozik. Hogy egy folyamat hogyan csatlakozik egy TSAP-hoz, az kívül esik a hálózati modellen, és teljes egészében csak a helyi operációs rendszertől függ. Például használhat egy, a mi LISTEN hívásunkhoz hasonló megoldást.

2. Egy, az 1. hoszton futó alkalmazási folyamat meg akarja tudni az aktuális időt, így végrehajt egy CONNECT kérést, megjelölve a 6-os TSAP-ot mint forrást és a 122-est mint célt.

3. Az 1. hoszt szállítási entitása kiválaszt egy hálózati címet a saját gépén (abban az esetben, ha több is van), és létrehoz a két gép között egy hálózati összeköttetést (összeköttetés nélküli alhálózatnál erre nem lenne szükség). Ezen a hálózati összeköttetésen keresztül az 1. hoszt szállítási entitása kommunikálni tud a 2. hoszton működő társával.

4. Az első, amit az 1. hoszt szállítási entitása a partnerének mond: „Szervusz! Egy szállítási összeköttetést szeretnék létesíteni az én 6-os TSAP-om és a te 122-es TSAP-od között. Mit szólsz hozzá?"



6.8. ábra. *TSAP-ok, NSAP-ok és összeköttetések*

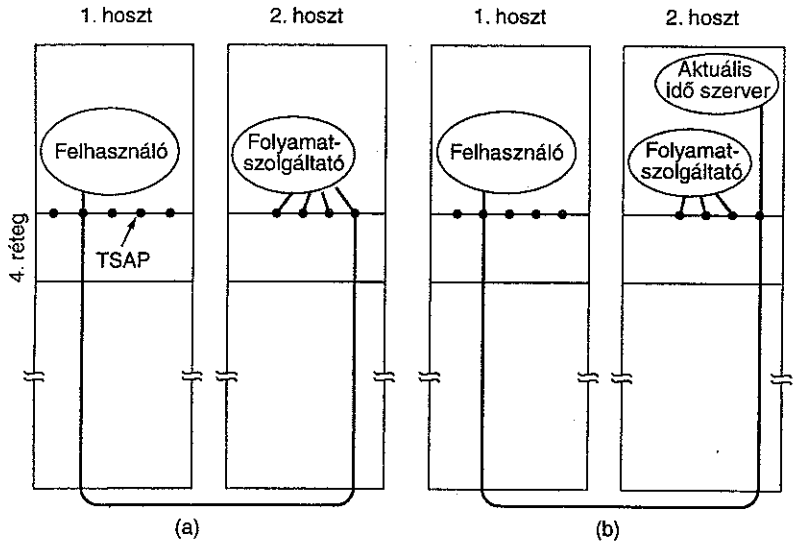
5. A 2. hoszt szállítási entitása ekkor a 122-es TSAP-ra csatlakozó aktuális idő szervertől megkérdezi, hogy hajlandó-e új összeköttetést fogadni. Ha beleegyezik, a szállítási összeköttetés létrejön.

Jegyezzük meg, hogy amíg a szállítási összeköttetés TSAP-tól TSAP-ig terjed, a hálózati összeköttetés ennek csak egy része, NSAP-tól NSAP-ig tart.

A fönt lefestett kép gyönyörű, azonban egy kisebb problémát ügyesen a szőnyeg alá söpörtünk. Honnan tudja az 1. hoszton futó folyamat, hogy az aktuális időt szolgáltató szerver a 122-es TSAP-ra csatlakozik? Az egyik lehetőség az, hogy a szóban forgó szerver már évek óta a 122-es TSAP-ot használja, és ezt a hálózati felhasználók fokozatosan megtanulták. Ebben a modellben a szolgáltatások rögzített TSAP címmel működnek, amit papírra nyomtatva oda lehet adni a hálózathoz csatlakozó új felhasználóknak.

Bár a rögzített TSAP címek megfelelőek lehetnek kisszámú kulcsfontosságú szolgáltatásra, melyek soha sem változnak, általában nem ez a helyzet. A felhasználói folyamatok gyakran akarnak kommunikálni más felhasználói folyamatokkal, melyek csak rövid ideig léteznek, és TSAP címük sem ismert előre. Továbbá, ha potenciálisan nagyszámú ritkán használt szerver folyamat van, pazarlás mindet egész nap futtatni, miközben egy rögzített TSAP címen várakoznak. Röviden: egy jobb módszerre van szükség.

Egy ilyen megoldás – amit UNIX hosztok alkalmaznak az Interneten – látható egyszerűsített formában a 6.9. ábrán és **kezdeti összeköttetést létesítő protokoll (Initial Connection Protocol)** néven ismert. Ahelyett, hogy minden számba vehető szerver a maga jól ismert TSAP címén várakozna, minden gépen, amely szolgáltatást kínál tá-



6.9. ábra. *Az 1. hoszton futó felhasználói folyamat és a 2. gépen futó aktuális idő szerver közötti összeköttetés felépítése*

voli felhasználók részére, egy speciális **folyamatszolgáltató (process server)** fut. Ez a folyamat a többi, kevésbé gyakran használt szerver megbízottjaként tevékenykedik. Több portot figyel egyidejűleg TCP összeköttetés-kérésre várva. A szolgálat potenciális felhasználói egy CONNECT kérést kezdeményeznek, amiben a kívánt szolgáltatás TSAP címét (TCP port) adják meg. Ha nincs rájuk várakozó szerver, a folyamatszolgáltatóval létesül összeköttetés, amint azt a 6.9.(a) ábrán láthatjuk.

Miután a folyamatszolgáltató megkapja a beérkező kérést, létrehozza a megfelelő szervert, aminek átadja a felhasználóval már fennálló összeköttetését. A szerver elvégzi a kért feladatot, miközben a folyamatszolgáltató továbbra is kérésekre várakozik. Ezt mutatja be a 6.9.(b) ábra.

Míg a kezdeti összeköttetést létesítő protokoll ragyogóan működik olyan szerverek esetén, melyeket elegendő akkor létrehozni, amikor szükség van rájuk, sok eset van, amikor szolgáltatások a folyamatszolgáltatótól függetlenül léteznek. Például egy állományszolgáltatónak speciális hardveren (nagykapacitású merevlemezzel ellátott gépen) kell futnia, nem lehet csak úgy menetközben létrehozni, amikor valaki használni akarja.

Az ilyen esetek kezelésére gyakran egy alternatív módszert alkalmaznak. Ebben a modellben egy **névszolgáltatónak (name server)**, vagy olykor **katalógusszolgáltatónak (directory server)** is nevezett speciális folyamat működik. Hogy a felhasználó egy adott szolgáltatás nevéhez (pl. „aktuális idő") tartozó TSAP címet megtudja, összeköttetést létesít a névszolgáltatóval (ami a jól ismert TSAP címén várakozik). A felhasználó üzenetet küld a kért szolgáltatás nevével, mire a névszolgáltató visszaküldi annak TSAP címét. A felhasználó ezután megszünteti az összeköttetést a névszolgáltatóval, és újat létesít a kívánt szolgáltatással.

Ebben a modellben egy új szolgáltatás létesítésekor a szolgáltatás nevével (rendszerint egy ASCII füzér) és TSAP címével be kell jelentkezni az állományszolgáltatónál, s az a kapott információt feljegyzi belső adatbázisába. Ha később kérés érkezik erre a szolgáltatásra, tudni fogja a választ.

A névszolgáltató feladata analóg a tudakozóéval a távbeszélőrendszerekben – egy nevekről számokra történő leképezést valósít meg. Éppúgy, mint a távbeszélőrendszerekben, lényeges, hogy a névszolgáltató (vagy kezdeti összeköttetést létesítő protokoll esetén a folyamatszolgáltató) jól ismert TSAP címe valóban mindenki által ismert legyen. Ha nem tudjuk a tudakozó telefonszámát, nem lehet fölhívni a tudakozót, hogy megkérdezzük. Ha azt gondolnánk, hogy a tudakozó telefonszáma nyilvánvaló, próbáljuk meg valamikor fölhívni egy másik ország tudakozóját.

Most tegyük fel, hogy a felhasználónak sikerült megtudnia az elérni kívánt TSAP címét. Egy újabb érdekes kérdés: honnan tudja a helyi szállítási entitás, hogy mely gépen van a kívánt TSAP? Pontosabban fogalmazva, honnan tudja a szállítási entitás, hogy melyik hálózati címen létesítsen hálózati összeköttetést azzal a távoli szállítási entitással, ami a kért TSAP-ot kezeli?

A válasz a TSAP cím struktúrájától függ. Egy lehetséges megoldás **hierarchikus címek** használata. A hierarchikus cím mezők sorozatából áll, melyek diszjunkt részekre osztják a címtartományt. Például egy igazi világegyetemi TSAP cím struktúrája a következő lehet:

cím = <galaxis> <csillag> <bolygó> <ország> <hálózat> <hoszt> <port>

Ezzel a módszerrel egyszerűen megtalálható egy TSAP bárhol az ismert világegyetemben. Ugyanígy, ha egy TSAP cím NSAP cím és portszám (helyi azonosító, ami a helyi TSAP-ok közül jelöl ki egyet) összefűzéséből áll, akkor ha a szállítási entitás kap egy TSAP címet, amelyhez összeköttetést kell létesítenie, a TSAP címben található NSAP címet használja a megfelelő távoli szállítási entitás eléréséhez.

A hierarchikus címre egyszerű példaként vegyük az 19076543210 telefonszámot. Ez a telefonszám 1-907-654-3210 struktúrájúnak tekinthető, ahol 1 az ország kódja (Egyesült Államok és Kanada), 907 a területi kód (Alaszka), 654 egy Alaszkában levő helyi központ, és 3210 az egyik port (előfizetői vonal) a helyi központban.

A hierarchikus cím mellett egy másik alternatíva az **egyszerű cím (flat address space)**. Ha a TSAP cím nem hierarchikus, akkor egy második leképzés szükséges a megfelelő hoszt megtalálásához. Ekkor egy olyan névszolgáltatóra lenne szükség, ami szállítási címeket fogadna a kérésekben, és válaszul hálózati címeket adna. Alternatív megoldásként néhány helyzetben (pl. LAN esetén) lehetséges üzenetszórásos (broadcast) kérést küldeni, ami arra kérné a célgépet, hogy azonosítsa magát egy csomag elküldésével.

6.2.2. Összeköttetés létesítése

Az összeköttetés felépítése egyszerűnek tűnik, de valójában meglepően trükkös folyamat. Első pillantásra elegendő lenne, hogy az egyik szállítási entitás CONNECTION REQUEST TPDU-t küldene a másik félnek, és CONNECTION ACCEPT (ÖSSZEKÖTTETÉS ELFOGADVA) válaszra várna. A probléma akkor merül fel, ha a hálózat csomagokatt veszít, tárol, vagy akár megkettőz.

Képzeljük el, hogy egy alhálózat annyira zsúfolt, hogy a nyugtázások nemigen érkeznek vissza időben, minden csomag időtúllépéssel érkezik meg, a csomagokat kétszer-háromszor újraküldik. Tegyük föl, hogy az alhálózat belül datagramokat használ, és minden csomag különböző útvonalon halad. Néhány csomag közlekedési dugóba kerülhet, és hosszú ideig nem érkezik meg, vagyis az alhálózat jelentős ideig tárolja őket, majd jóval később bukkannak elő.

A létező legrosszabb rémálom a következő: egy felhasználó összeköttetést létesít egy bankkal, és üzenetet küld azzal a megbízással, hogy a bank utaljon át nagy pénzösszeget egy nem igazán megbízható személy számlájára, majd lebontja az összeköttetést. Szerencsétlenségére minden elküldött csomag megkettőződik, és valahol a hálózat mélyén tárolódik. Az összeköttetés befejezése után ezek sorban előbukkannak az alhálózatból, és helyes sorrendben megérkeznek a bankhoz újabb összeköttetést és tranzakciót kérve, majd ez az összeköttetés is megszűnik. A bank nem tudhatja, hogy kettőzött üzenetekről van szó. Feltételezi, hogy ez egy független, második tranzakció, így ismét átutalja a pénzt. Jelen alfejezetben a késleltetett kettőzések problémáját fogjuk tanulmányozni. Különös figyelmet szentelünk az összeköttetések megbízható módon történő létesítésére kifejlesztett algoritmusokra, hogy a föntiekhez hasonló rémálmok ne válhassanak valóra.

A probléma alapvető oka a késleltetett kettőzések létezése. Többféle ellenszer létezik, de egyik sem teljesen kielégítő. Az egyik módszer azt mondja, hogy használjunk

eldobható szállítási címeket. Ebben a megközelítésben minden esetben, amikor egy szállítási címre van szükség, újat generálunk. Az összeköttetés lebontása után a régi címet elvetjük. Ez a stratégia a 6.9. ábrán látható folyamatszolgáltató modell működését lehetetlenné teszi.

Egy másik lehetőség minden összeköttetésnek egy olyan egyedi összeköttetés-azonosítót adni (egy sorszámot, ami minden újabb összeköttetés létesítésekor eggyel nő), amit a kezdeményező fél generál és minden TPDU-ba (beleértve az összeköttetést kérőt is) beletesz. Az összeköttetés lebontása után minden szállítási entitás frissíti a befejeződött összeköttetések tábláját, amely bejegyzései (társ szállítási entitás, összeköttetés sorszám) párokból áll. Minden újabb összeköttetés-kéréskor ellenőrizhető, hogy az nem egy régi összeköttetéshez tartozik-e.

Sajnos ennek a módszernek van egy alapvető hibája: minden szállítási entitás határozatlan ideig történeti információt kell hogy tároljon. Ha egy gép összeomlik, és memóriatartalmát elveszti, többé nem tudja, hogy mely összeköttetés azonosítót használta már.

Ehelyett más megközelítést kell vennünk. Ahelyett, hogy egy csomagot örök időre életben hagynánk az alhálózatban, ki kell fejlesztenünk egy olyan mechanizmust, amely az öreg és még mindig bolyongó csomagokat kiirtja. Ha garantálni tudjuk, hogy egyetlen csomag sem él tovább egy adott időtartamnál, a probléma valamivel kezelhetőbbé válik.

A csomagok élettartama ismert maximumra korlátozható az alábbi módszerek közül valamelyikkel:

1. Korlátozott alhálózat tervezése.

2. Átugrásszámláló (hop counter) alkalmazása a csomagokban.

3. A csomagok időbélyeggel való ellátása.

Az első módszer magába foglal bármilyen módszert, amely megakadályozza, hogy a csomag hurokba kerüljön, kombinálva valamilyen olyan megoldással, amely korlátozza a torlódás okozta késleltetést a (pillanatnyilag ismert) létező leghosszabb úton. A második módszer esetén minden egyes csomópont elhagyásakor növeljük az átugrásszámlálót. Az adatkapcsolati protokoll egyszerűen eldob minden csomagot, melynek átugrásszámlálója egy adott értéket meghalad. A harmadik módszer megköveteli, hogy minden csomag tartalmazza az előállításának időpontját, és ha a csomag öregebbé válik egy előre meghatározott időtartamnál, a csomagot éppen továbbítani szándékozó router egyszerűen dobja el. Ez utóbbi eljáráshoz szükségessé válik a routerek óráinak szinkronizálása, ami önmagában sem triviális feladat, hacsak nem áll rendelkezésre hálózaton kívüli szolgáltatás, pl. WWV vagy más rádióállomás (pl. DCF-77), ami periodikusan sugározza a pontos időt.

Gyakorlatban nem elég azt biztosítanunk, hogy egy csomag halott, hanem ennek igaznak kell lenni minden rá vonatkozó nyugtára is, ezért bevezetjük a T időtartamot, ami a valódi maximális csomagélettartam kis egész számú többszöröse. Az alkalmazott szorzó protokollfüggő, és szerepe egyszerűen csak T növelése. A csomag elküldé-

se után T idő várakozás után biztosak lehetünk abban, hogy a csomag már minden nyom nélkül eltűnt, és sem a csomag, sem a nyugtázások nem fognak hirtelen előtűnni a ködből, és további bonyodalmakat okozni.

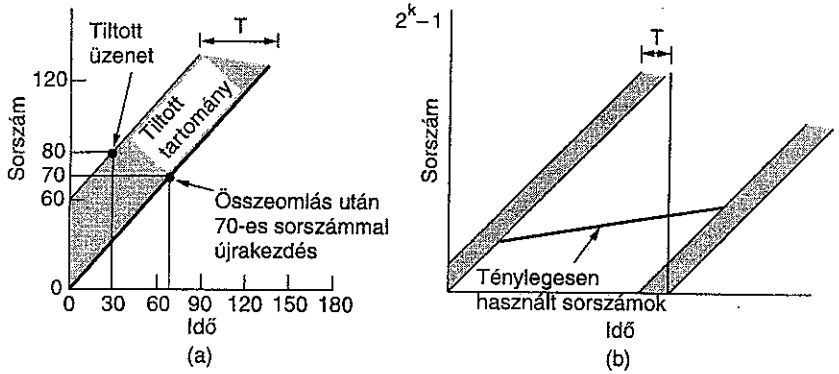
Korlátozott élettartamú csomagokat felhasználva bolondbiztos eljárást lehet kifejleszteni az összeköttetés biztonságos felépítésére. Az alább ismertetett eljárás Tomlinson nevéhez fűződik (1975). Ezzel ugyan megoldódik a probléma, viszont egyéb gondok jelentkeznek. A módszert továbbfinomította Sunshine és Dalal (1978), ennek különböző változatait széles körben alkalmazzák a gyakorlatban.

Annak a problémának megkerülésére, hogy egy gép egy összeomlás után nem tudja megállapítani, hogy hol is tartott, Tomlinson azt javasolta, hogy minden hosztot időt mutató órával lássanak el. A különböző hosztokon levő óráknak nem szükséges szinkronban járniuk. Minden óra egy bináris számlálóval valósítható meg, ami egységes időközönként növeli értékét. Ezenkívül a számlálóban levő bitek számának egyenlőnek vagy nagyobbnak kell lennie, mint a sorszámokban levő bitek száma. Végül, és ami a legfontosabb, a hoszt meghibásodásakor is tovább kell járnia az órának.

Az alapötlet az, hogy az algoritmus nem engedi két azonos sorszámú TPDU egyidejű létezését. Az összeköttetés fölépítésekor az óra értékének alsó k bitje alkotja a kezdeti (szintén k bites) sorszámot. Így, eltérően a 3. fejezetben leírt protokolloktól, minden összeköttetés más sorszámmal kezdi számozni a TPDU-it. A használt tartománynak olyan nagynak kell lennie, hogy mire a sorszámok körbeérnek, az azonos sorszámú TPDU már rég eltűnjön. Ezt az időt és a kezdeti sorszámok közti lineáris összefüggést mutatja a 6.10. ábra.

Ha valamikor a szállítási entitások már megegyeztek a kezdeti sorszámban, bármilyen csúszóablakos protokoll használható forgalomszabályozásra. Valóságban a kezdeti sorszám időfüggését jelző vastag görbe nem igazán egyenes, hanem lépcsőzött, mivel az óra értéke diszkrét lépésekben növekszik. Az egyszerűség kedvéért ezt a részletet elhanyagoljuk.

Probléma akkor lép föl, ha egy hoszt összeomlik. Újraindulásakor a benne működő



6.10. ábra. *(a) TPDU nem kerülhet a tiltott tartományba. (b) Az újraszinkronizációs probléma*

szállítási entitás nem fogja tudni, hogy milyen sorszámnál tartott. Egy lehetséges megoldás szerint, a szállítási entitások újraindulás után várjanak T ideig, hogy az összes régi TPDU addigra eltűnhessen. Egy összetett, több hálózatot összekapcsoló hálózatban azonban T igen nagy lehet, így ez a stratégia kevésbé vonzó.

Egy összeomlás utáni T időtartamnyi tétlenség elkerülésére a sorszámokra bevezetünk egy másik korlátozást is. Ennek szükségességét legkönnyebben egy példán keresztül mutathatjuk be. Legyen T, a maximális csomagélettartam, pontosan 1 perc. Az óra másodpercenként növeli értékét. Mint a 6.10. ábrán láthatjuk, az x időpontban fölépített összeköttetés kezdeti sorszáma x lesz. Tegyük föl, hogy $t = 30$ másodperckor egy közönséges adat TPDU indul a (már korábban létrehozott) 5-ös összeköttetésen keresztül 80-as sorszámmal, legyen ez az X TPDU. Közvetlen elküldése után a hoszt összeomlik, majd gyorsan újraindul. $t = 60$ pillanatban megkezdi 0-tól 4-ig terjedő sorszámú összeköttetéseit újra fölépíteni. $t = 70$-kor az 5-ös összeköttetést is újra létrehozza 70-es kezdeti sorszámmal, ahogy az elő van írva. A következő 15 másodperc alatt 11 TPDU-t küld el 70-től 80-ig terjedő sorszámokkal. Így $t = 85$ másodperckor egy új, 80-as sorszámú TPDU indul el az 5-ös összeköttetésen keresztül az alhálózatban. Sajnos azonban az X TPDU még mindig létezik. Ha ez az új, 80-as számú TPDU előtt érkezik meg, a vevő az X TPDU-t fogadná el, és az igazi, 80-as számút, mint kettőzést eldobná.

Az ilyen problémák megelőzésére el kell kerülnünk a sorszámok használatát (azaz új TPDU-hoz rendelését) a potenciális kezdeti sorszámként történő alkalmazás előtti T időtartamban. Az illegális (sorszám–idő) párosítást a 6.10. ábrán látható **tiltott tartomány** jelöli. Bármely TPDU bármely összeköttetésen történő továbbítása előtt a szállítási entitásnak le kell olvasnia az óráját, hogy ellenőrizze, nem a tiltott tartományban van-e.

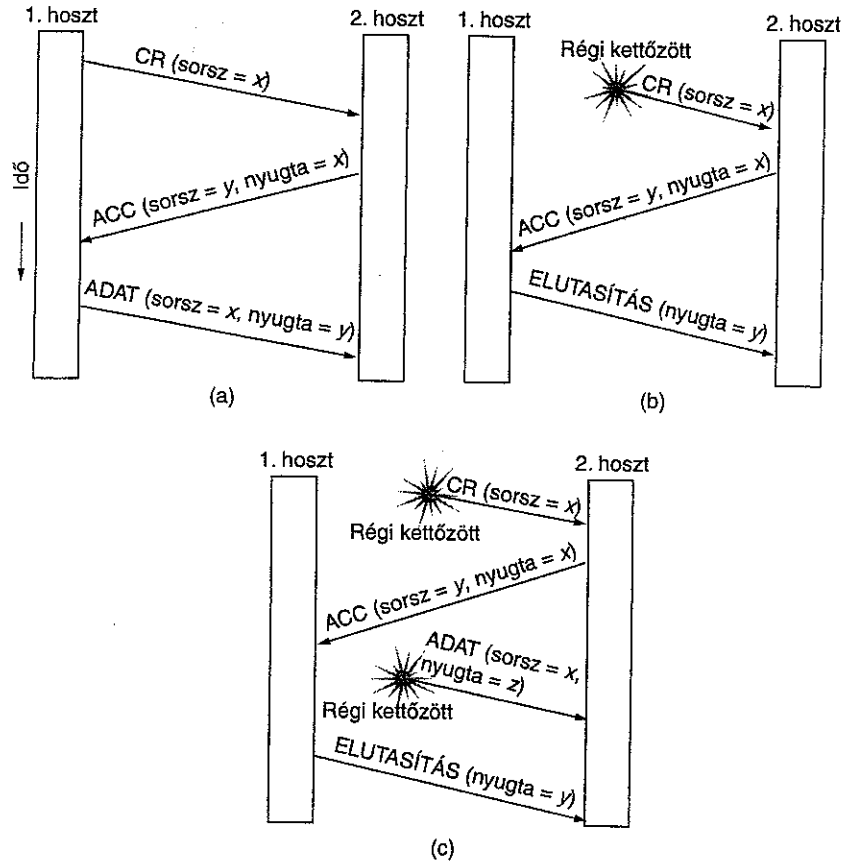
A protokoll így is kétféleképpen kerülhet bajba. Ha egy hoszt túl gyorsan küld túl sok adatot egy frissen létesített összeköttetésen, az aktuális sorszám–idő görbe meredekebben emelkedhet, mint a kezdeti sorszám–idő görbe. Ez azt jelenti, hogy a maximális adatsebesség bármelyik összeköttetésen egyenlő egy TPDU-val óraütésenként. Azt is jelenti, hogy egy összeomlás utáni újraindításkor egy új összeköttetés megnyitása előtt a szállítási entitásnak egy óraütésig kell várni, hogy elkerülje egy sorszám ismételt használatát. Mindkét probléma rövidebb óraütem (pár ezredmásodperc) használatát teszi indokolttá.

Sajnos, nem csak úgy lehet bajba kerülni, hogy túl gyors adással alulról kerül a szállítási entitás a tiltott tartományba. A 6.10.(b) ábra alapján világossá kell válnia annak, hogy tetszőleges, az óra sebességénél kisebb adási sebességnél a tényleges idő–sorszám görbe végül balról fog belépni a tiltott tartományba. Minél meredekebb az említett görbe, annál később következik be ez az esemény. Mint fönt említettük, minden TPDU elküldése előtt a szállítási entitás meg kell hogy vizsgálja, hogy belépne-e a tiltott tartományba, és ha igen, vagy vár T s-ot az elküldés előtt vagy újraszinkronizálja a sorszámokat.

Az óra alapú módszer megoldja az adat TPDU-k késleltetett kettőzési problémáit, de hogy ez a módszer használható legyen, először létre kell hozni az összeköttetést. Mivel a vezérlő TPDU-k szintén késhetnek, a két fél potenciális problémája az, hogy hogyan egyezzenek meg a kezdeti sorszámban. Tegyük fel például, hogy az összeköt-

tetés úgy épül föl, hogy az 1. hoszt elküldi a CONNECTION REQUEST TPDU-t a használni kívánt kezdeti sorszámmal és port számmal a távoli 2. hosztnak. Ez utóbbi nyugtázza a kérést egy CONNECTION ACCEPTED TPDU visszaküldésével. Ha a CONNECTION REQUEST TPDU elvész, de később fölbukkan egy kettőzött példánya a 2. hosztnál, az összeköttetés helytelenül jön létre.

Ennek a problémának megoldására vezette be Tomlinson a **háromutas kézfogás (three-way handshake)** módszert (1975). Ez az összeköttetés-létesítési protokoll nem igényli, hogy mindkét fél azonos sorszámmal kezdje az adást, így a globális időtől eltérő módszer is használható szinkronizálásra. Egy normális összeköttetés-létesítési eljárást, amikor az 1. hoszt kezdeményez, láthatunk a 6.11.(a) ábrán. Az 1. hoszt kivá-



6.11. ábra. *Három forgatókönyv a háromutas kézfogás protokollal történő összeköttetés-létesítésre. CR és ACC rendre CONNECTION REQUEST és CONNECTION ACCEPTED rövidítései. (a) Normális működés. (b) Régi kettőzött CR bukkan elő. (c) Kettőzött CR és kettőzött adat TPDU esete*

laszt egy x sorszámot, és egy CONNECTION REQUEST TPDU-ban elküldi a 2. hosztnak. Az egy CONNECTION ACCEPTED TPDU-val nyugtázza x értékét, és bejelenti saját y kezdeti sorszámát. Végül az 1. hoszt jóváhagyja a 2. hoszt által választott kezdeti sorszámot az első általa küldött adat TPDU-ban.

Lássuk tehát, hogy működik a háromutas kézfogás protokoll késleltetett kettőzött vezérlő TPDU-k esetén. A 6.11.(b) ábrán az első TPDU egy régebbi összeköttetés késleltetett kettőzött CONNECTION REQUEST üzenete. Ez a 2. hoszthoz anélkül érkezik meg, hogy az 1. hoszt tudna róla. A 2. hoszt válaszul CONNECTION ACCEPTED TPDU-t küld egy kéréssel bővítve, hogy ellenőrizze, partnere tényleg új összeköttetést akar-e létesíteni. Mivel az 1. hoszt elutasító választ küld, a 2. hoszt rájön, hogy egy késleltetett kettőzés csapta be, és felhagy az összeköttetés-létesítéssel. Ily módon egy késleltetett kettőzött TPDU nem okoz kárt.

Legrosszabb az az eset, amikor mind egy késleltetett CONNECTION REQUEST és egy CONNECTION ACCEPTED üzenet nyugtája (vagyis adat TPDU) bolyong az alhálózatban, ezt láthatjuk a 6.11.(c) ábrán. Mint az előző példában, a 2. hoszt kap egy késleltetett CONNECTION REQUEST üzenetet, és válaszol rá. Ezen a ponton nagyon fontos észrevennünk azt, hogy a 2. hoszt úgy ajánlotta y-t a tőle az 1. hoszt felé menő forgalom kezdeti sorszámául, hogy teljesen biztos volt abban, hogy sem y sorszámú üzenet, sem ennek nyugtája már nem létezik. Amikor a második késleltetett TPDU megérkezik a 2. hoszthoz, az abból a tényből, hogy partnere z-t nyugtázza y helyett, fölismeri, hogy most is egy régi másolattal áll szemben. Kiemeljük, hogy nincs olyan kombinációja a régi CONNECTION REQUEST és CONNECTION ACCEPTED, vagy más TPDU-knak, ami félrevezeti a protokollt, és véletlenül összeköttetést létesít, mikor senki sem kezdeményezte.

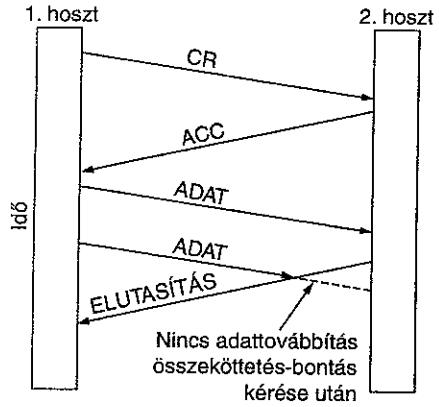
Késleltetett kettőzések esetén használható, összeköttetések megbízható létesítésére alkalmas alternatív módszert írt le Watson (1981). Ő több időzítő felhasználásával előzi meg a nem kívánt eseményeket.

6.2.3. Az összeköttetés bontása

Az összeköttetést bontani jóval egyszerűbb, mint felépíteni. Azonban több buktató van itt, mind gondolnánk. Mint korábban említettük, az összeköttetés kétféleképpen bontható: *aszimmetrikus* és *szimmetrikus* módon. Aszimmetrikus bontást alkalmaznak pl. a távbeszélőhálózatokban: amikor az egyik fél leteszi a kagylót, az összeköttetés megszakad. Szimmetrikus bontás esetén az összeköttetést két független egyirányú összeköttetésként kezelik, ahol mindkettőt külön kell lebontani.

Az aszimmetrikus összeköttetés-bontás váratlanul történik és adatvesztéssel járhat. Vegyük például a 6.12. ábra forgatókönyvét. Az összeköttetés létrejötte után az 1. hoszt egy TPDU-t küld a 2. hosztnak, s az rendben meg is érkezik. Az 1. hoszt újabb TPDU-t küld. Sajnos a 2. hoszt kiad egy DISCONNECT-et mielőtt a második TPDU megérkezik. Az összeköttetés lebomlik, és ezzel együtt az adat elvész.

Világos, hogy kifinomultabb bontási protokoll szükséges az adatvesztés elkerüléséhez. Egyfajta megoldás a szimmetrikus bontás használata, amikor is mindkét irányt a másiktól függetlenül bontjuk le. Itt egy hoszt azután is fogadhat adatot, hogy már elküldött egy DISCONNECT TPDU-t.



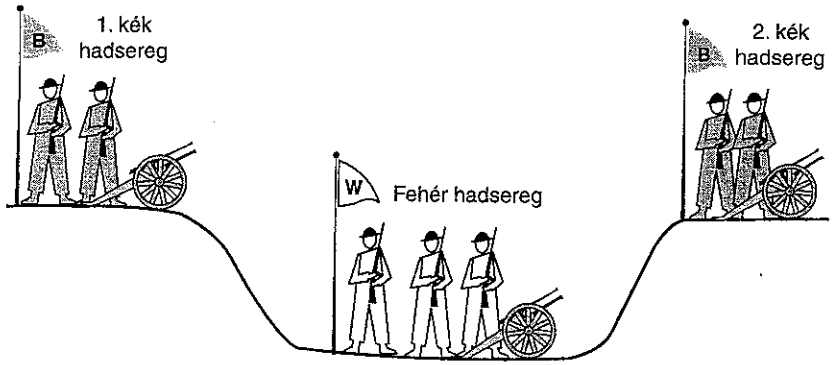
6.12. ábra. *Hirtelen összeköttetés-bontás adatvesztéssel*

A szimmetrikus bontás megfelelően működik, ha mindkét félnek rögzített mennyiségű elküldendő adata van, és mindegyik pontosan tudja, hogy mikor küldte el. Más helyzetekben annak eldöntése, hogy végeztek dolgukkal, és a kapcsolat bontható, nem annyira nyilvánvaló. Elképzelhető egy olyan protokoll, melyben az 1. hoszt így szól:

„Végeztem. Te is készen vagy?" „Én is elkészültem. Szervusz!" – válaszol a 2. hoszt, és az összeköttetés biztonságosan lebontható.

Sajnos, ez a protokoll nem mindig működik. Egy híres, a **két-hadsereg probléma** néven ismert feladat pont erről szól. Képzeljük el, hogy a fehér hadsereg a völgyben táborozik, mint a 6.13. ábra mutatja. Mindkét környező dombot a kék hadsereg birtokolja. A fehér sereg bármelyik kék hadseregnél nagyobb, azonban azok együtt nagyobbak a fehér seregnél. Ha bármelyik kék hadsereg egyedül támad, biztos vereséget szenved, de együttes támadásuk során megsemmisíthetnék a fehér sereget.

A kék seregek össze akarják egyeztetni támadásukat. Azonban az összes kommuni-



6.13. ábra. *A két-hadsereg probléma*

kációs lehetőségük a völgyön gyalog átküldött futárokra korlátozódik, akiket persze elfoghatnak, így az üzenet elvész (tehát megbízhatatlan kommunikációs csatornát használnak). Az a kérdés, hogy létezik-e olyan protokoll, ami győzelemre segíti a kék seregeket?

Tegyük fel, hogy az 1. kék hadsereg parancsnoka a következő üzenetet küldi: „Azt javaslom, hogy március 29-én támadjunk. Mi a véleményetek?" Tegyük fel továbbá, hogy az üzenet megérkezik, a 2. kék hadsereg parancsnoka egyetért, és válasza szerencsésen megérkezik az 1. kék sereghez. Végrehajtják a támadást? Valószínűleg nem, mert a 2. sereg parancsnoka nem tudja, hogy üzenete átjutott-e a völgyön. Ha nem, az 1. sereg nem fog támadni, így egymaga bolond lenne fölvenni a küzdelmet.

Bővítsük hát a protokollt a háromutas kézfogás technikával. Az eredeti javaslat kezdeményezőjének nyugtáznia kell a kapott választ. Feltéve, hogy nem veszett el üzenet, a 2. kék hadsereg megkapja a nyugtát, de most az 1. kék sereg parancsnoka fog habozni. Végül is ő nem tudja, hogy a nyugta átjutott-e vagy sem, és ha nem, tudja, hogy nem számíthat a 2. sereg támadására. Használhatnánk négyutas kézfogást is, de az sem segítene.

Valójában könnyen bebizonyítható, hogy nem lehet működő protokollt létrehozni. Tegyük fel mégis, hogy van ilyen. A protokoll utolsó üzenete vagy lényeges, vagy nem. Utóbbi esetben hagyjuk el az összes többi fölösleges üzenettel együtt, amíg olyan protokollhoz nem jutunk, melynek minden üzenete nélkülözhetetlen. Mi történik, ha az utolsó üzenet nem jut át? Mivel erről megállapítottuk, hogy lényeges, így ha elvész, nem kezdődik el a támadás. Mivel az utolsó üzenet küldője sohasem lehet biztos annak célbaérkezésében, nem kockáztatja meg a támadást. Sőt, ami még rosszabb, ezt a másik kék sereg is tudja, így ők sem támadnak.

Hogy észrevegyük a két-hadsereg probléma és az összeköttetések bontása között vonható párhuzamot, helyettesítsük a „támadás" szót a „bontás"-sal. Ha egyik fél sem készült föl a bontásra addig, míg meg nem győződött arról, hogy partnere is fölkészült, az összeköttetés bontására sohasem kerül sor.

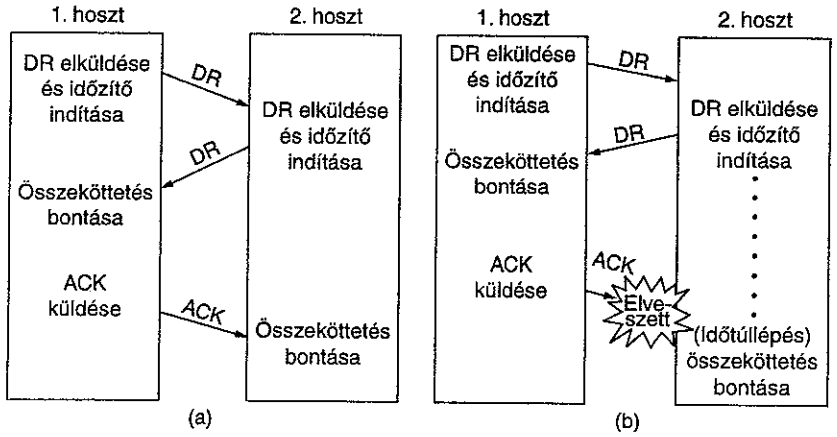
A gyakorlatban a felek több kockázatot vállalnak az összeköttetés lebontásánál, mint ha a fehér hadsereget kellene megtámadniuk, ezért a helyzet nem teljesen reménytelen. A 6.14. ábrán négy forgatókönyvet mutatunk az összeköttetés-bontásra háromutas kézfogás használata esetén. Bár ez a protokoll nem sebezhetetlen, mégis kielégítően viselkedik.

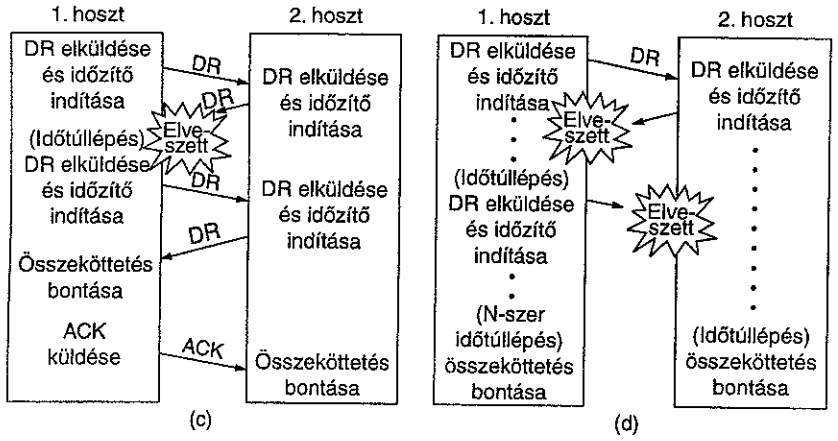
A 6.14.(a) ábrán a normális működést láthatjuk, ahol az egyik partner az összeköttetés bontásának kezdeményezéseként DR (DISCONNECTION REQUEST) TPDU-t küld és elindít egy időzítőt. Amikor ez megérkezik, a vevő szintén visszaküld egy DR TPDU-t, és elindít egy időzítőt arra az esetre, ha az általa küldött DR elveszne. Amikor ez a DR megérkezik, a kezdeményező visszaküld egy ACK TPDU-t, és bontja az összeköttetést. Végül, mikor az ACK TPDU megérkezik, a vevő szintén bontja az összeköttetést. Az összeköttetés bontása azt jelenti, hogy a szállítási entitás az összeköttetésre vonatkozó információt törli a nyitott kapcsolatok táblájából, és jelzi az összeköttetés befejezését a tulajdonosnak (a szállítási felhasználónak). Ez a művelet eltér attól, amikor a szállítási felhasználó kiad egy DISCONNECT primitív hívást.

Ha az utolsó ACK TPDU elvész, ahogy az a 6.14.(b) ábrán látható, a helyzetet az időzítő menti meg. Amikor az lejár, az összeköttetés mindenképpen befejeződik.

Most tekintsük azt az esetet, amikor a második DR vész el. Az összeköttetés bontását kezdeményező fél nem kapja meg a várt választ, lejár az időzítője, és az egészet elkezdi elölről. A 6.14.(c) ábrán látható ez a működés, feltételezve, hogy a második próbálkozás során nem vesznek el a TPDU-k és mindegyik időben meg is érkezik.

Az utolsó eset, amit a 6.14.(d) ábrán láthatunk, azonos az előzővel, kivéve, hogy most feltételezésünk szerint minden további kísérlet a DR újraküldésére meghiúsul a TPDU-k elvesztése következtében. *N* próbálkozás után a küldő föladja, és fölbontja az összeköttetést. Eközben a vevő időzítése szintén lejár, és ő is bontja az összeköttetést.





6.14. ábra. *Négy protokoll forgatókönyv az összeköttetés lebontására. (a) Normális működés a háromutas kézfogással. (b) Az ACK vész el. (c) A válasz vész el. (d) A válasz és a rákövetkező DR-ek vesznek el*

Bár ez a protokoll rendszerint sikeresen működik, elméletileg kudarcot vallhat, ha a kezdeti DR és a további *N* újraküldött TPDU mind elvesznek. A küldő feladja és bontja az összeköttetést, míg a másik fél semmit sem tud a bontási kísérletekről, és még mindig teljesen aktív. Ennek a helyzetnek az eredménye egy félig nyitott összeköttetés.

Ez a probléma elkerülhető lenne, ha nem hagynánk, hogy a küldő *N* próbálkozás után föladja a kísérletezést, hanem kényszerítenénk, hogy örökké folytassa, amíg választ nem kap. Ha viszont a másik fél időzítést figyel, és az lejár, a küldő ténylegesen örökké fog próbálkozni, mert többé nem is kaphat választ. Ha nem engedjük a fogadó oldalt, hogy időtúllépés esetén a várakozást feladja, a 6.14.(b) ábrán látható protokoll elakad.

Egyik módja a félig nyitott összeköttetések kilövésének a következő: bevezetünk egy olyan szabályt, miszerint ha adott ideig nem érkezik TPDU, az összeköttetést automatikusan bontjuk. Ily módon, ha valamelyikük befejezi az összeköttetést, a másik észreveszi a forgalom hiányát és szintén bontja az összeköttetést. Természetesen ezen szabályozás bevezetése szükségessé teszi egy időzítő alkalmazását minden szállítási entitásnál, amit az minden TPDU elküldésekor nulláz és újraindít. Ha lejár, akkor egy üres (adatot nem hordozó) TPDU-t küld, hogy visszatartsa a vevőt az összeköttetés bontásától. Ha azonban az automatikus lebontási szabályt alkalmazzuk, és túl sok egymás után küldött üres TPDU elvész az amúgy kihasználatlan összeköttetésen, először az egyik, majd a másik oldal is bontja az összeköttetést.

Nem ragozzuk tovább ezt a témát, de mostanra már világossá kellett váljon, hogy az összeköttetés lebontása nem olyan egyszerű, mint amilyennek tűnik.

6.2.4. Forgalomszabályozás és pufferelés

Az összeköttetés-létesítés és -lebontás többé-kevésbé részletes tárgyalása után vizsgáljuk meg, hogyan kezelik az összeköttetéseket használat közben. Már korábban is felmerült az egyik kulcskérdés, a forgalomszabályozás. Bizonyos tekintetben a szállítási réteg forgalomszabályozással kapcsolatos problémái azonosak az adatkapcsolati rétegben tapasztaltakkal, de vannak eltérések is. Az alapvető hasonlóság az, hogy mindkét esetben csúszóablakos, vagy más módszer alkalmazása szükséges minden összeköttetésre, hogy a lassú vevőt megvédhessük a túl gyors adó üzenettömegétől. A fő különbség az, hogy egy routernek viszonylag kevés vonala van, míg egy hoszt számos összeköttetéssel rendelkezhet. Ez az eltérés az oka annak, hogy a szállítási rétegben gazdaságtalan volna az adatkapcsolati rétegben használt pufferelési eljárást megvalósítani.

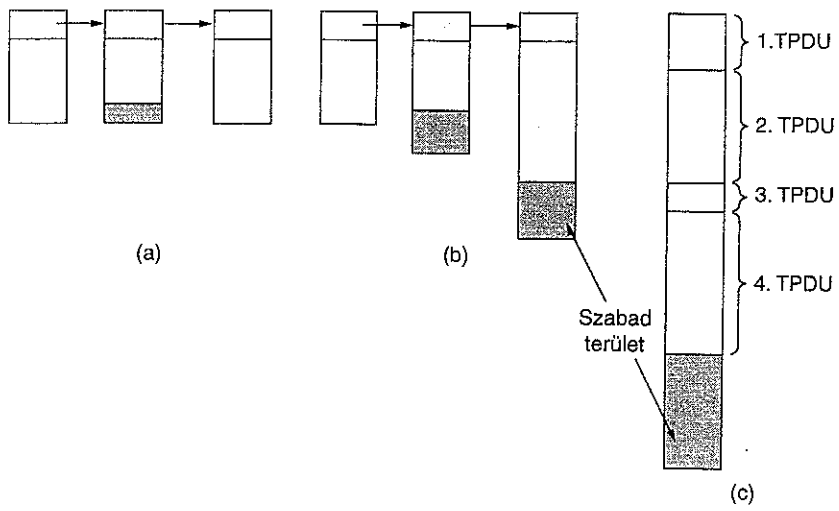
A 3. fejezetben tárgyalt adatkapcsolati protokollok esetében a kereteket mind a küldő, mind a fogadó csomópont pufferelte. Például a 6. protokollban mind az adó, mind a vevő *MAX_SEQ* + 1 puffert kell hogy rendeljen minden vonalához, felét bemenetre, felét kimenetre használva. Egy, mondjuk legfeljebb 64 összeköttetéssel rendelkező hoszt és négybites sorszámok használata esetén ez a protokoll 1024 puffert igényelne.

Az adatkapcsolati rétegben a küldő oldalnak pufferelnie kell a kimenő kereteket, mert azokat esetleg újra kell küldenie. Ha az alhálózat datagram szolgálatot nyújt, a küldő szállítási entitásnak hasonló okból szintén puffert kell használnia. Ha a vevő tudja, hogy a küldő minden TPDU-t tárol addig, míg nyugtát nem kap rájuk, a vevő

tetszés szerint rendelhet külön puffert különböző összeköttetéseihez, ahogy azt jónak látja. Esetleg egyetlen közös pufferterületet tarthat fenn az összes összeköttetése részére. Amikor egy újabb TPDU érkezik, megpróbál dinamikusan puffert igényelni. Ha sikerrel jár, elfogadja a TPDU-t, kudarc esetén eldobja. Mivel a küldő kész újraküldeni az alhálózatban elveszett TPDU-kat, nem probléma, ha a vevő eldob TPDU-kat, bár ezzel erőforrást pocsékol. A küldő addig próbálkozik, míg végül nyugtát nem kap.

Összefoglalva, megbízhatatlan hálózati szolgálat esetén a küldő minden kimenő TPDU-t pufferelni kényszerül, éppen úgy, mint az adatkapcsolati rétegben. Megbízható hálózati szolgálat esetén viszont további kompromisszumok lehetségesek. Ha a küldő biztos abban, hogy a vevőnek mindig van szabad puffere, nem szükséges a kimenő TPDU-król másolatot fönntartania. Ha azonban a vevő nem ad garanciát, hogy minden beérkező TPDU-t elfogad, az adónak mindenképpen pufferelnie kell. Ez utóbbi esetben a küldő nem bízhat a hálózati réteg nyugtájában, mivel az csak azt jelenti, hogy a TPDU megérkezett, arról nem ad választ, hogy elfogadták-e. Később még visszatérünk erre a fontos kérdésre.

Ha megállapodás született is arról, hogy a vevő pufferelni fog, még mindig kérdéses a puffer mérete. Ha a legtöbb TPDU nagyjából azonos méretű, kézenfekvő a pufferterületet azonos méretű pufferek készletébe szervezni úgy, hogy egy TPDU-ra egy puffer jusson, mint azt a 6.15.(a) ábra mutatja. Ha azonban a TPDU-k mérete széles határok közt változhat egy terminálon begépelt pár betűtől az állományátvitelkor mozgatott több ezer karakterig, a rögzített méretű pufferek problémát jelentenek. Ha a pufferméretet a leghosszabb előforduló TPDU méretében állapítjuk meg, rövid TPDU vétele esetén a hely nagy része kihasználatlan marad. Ha a pufferméret kisebb a TPDU maximális hosszánál, egy nagy TPDU több puffert és bonyolultabb kezelést igényel.



6.15. ábra. *Pufferkezelés. (a) Láncolt, rögzített méretű pufferek. (b) Láncolt, változó méretű pufferek. (c) Összeköttetésenként egyetlen nagy körpuffer*

A pufferméret problémájának egy másik kezelési módja a változó méretű pufferek használata, mint azt a 6.15.(b) ábra mutatja. Ennek előnye a jobb memóriakihasználtság, ami viszont bonyolultabb pufferkezeléssel jár. A harmadik lehetőség összeköttetésenként egyetlen nagy körpuffer alkalmazása, mint az a 6.15.(c) ábrán látható. Ez a megoldás szintén jó memóriakihasználtságot nyújt akkor, ha az összeköttetések erősen terheltek, de ez jelentősen romlik, ha néhány összeköttetés gyér forgalmat bonyolít le.

A forrásnál és a célnál folyó pufferelés közötti optimális egyensúly függ az összeköttetésen lebonyolított forgalom típusától. Kis sávszélességű, löketszerű forgalom céljára, amilyet pl. egy interaktív terminál állít elő, legjobb, ha sehol se különítünk el puffert, hanem szükség esetén dinamikusan igényeljük mindkét félnél. Mivel az adó nem tudhatja biztosan, hogy a vevő megkapta az igényelt puffert, a nyugta beérkezéséig tárolnia kell az elküldött TPDU-t. Másrészről állománytovábbítás és más, nagy sávszélességű átvitel esetén jobb, ha a vevő egy teljes ablaknyi puffert rendel a bejövő forgalomhoz, hogy teljes sebességgel folyhasson az adatátvitel. Tehát kis sávszélességű, löketszerű forgalom esetén célszerű a küldőnél megvalósítani a pufferelést, míg egyenletes, nagy sávszélességű adatfolyamot a vételi oldalon előnyösebb pufferelni.

Ahogy összeköttetések létrejönnek és lebomlanak, és ahogy a forgalom jellege változik, úgy kell az adónak és a vevőnek dinamikusan hozzáigazítani ehhez a pufferfoglalást. Eszerint a szállítási protokollnak lehetővé kell tennie a küldő entitás számára, hogy a másik féltől pufferterület lefoglalását kérje. Puffereket összeköttetésenként, vagy a két hoszt között élő minden összeköttetésre együtt lehet lefoglalni. Alternatív megoldásként a vevő a jövendő forgalmat nem, de saját puffereinek adatait ismerve közölheti az adóval: „Lefoglaltam részedre *X* db puffert."

Ha az éppen használt összeköttetések száma növekedne, szükség lehet a pufferek számára lefoglalt terület csökkentésére. A protokollnak ezt a lehetőséget is biztosítania kell.

A pufferek dinamikus kezelésének egy ésszerű, általános módja a pufferelésnek a nyugtázástól való különválasztása. Ez a 3. fejezetben leírt csúszóablakos protokollal ellentétes működést jelent. A dinamikus pufferkezelés valójában változó méretű csúszóablakot jelent. Először az adó a forgalom becsült igénye alapján puffert kér a vevőtől. A vevő annyi puffert ad, amennyit adhat. Az adó minden TPDU küldésekor ezt a számot csökkenti, és leáll az átvitellel, ha eléri a nullát. A vevő közben a visszafelé menő forgalomra ülteti a nyugtákat és a pufferek foglaltsági jellemzőit.

A 6.16. ábrán egy példát mutatunk arra, hogy hogyan működne a dinamikus ablak kezelés datagram alhálózat fölött négybites sorszámokkal. Tegyük fel, hogy a pufferfoglaltsági információ – mint az ábra jelzi – nem az ellentétes irányba közlekedő TPDU-kra ültetve, hanem külön TPDU-ban utazik. Először az *A* hoszt 8 puffert szeretne, de ezekből csak négyet kaphat. *A* ezután elküld három TPDU-t, melyek közül az utolsó elvész. A 6. TPDU nyugtáz az 1-es sorszámúval bezárólag (azt is beleértve) minden elküldött TPDU-t, lehetővé téve *A*-nak, hogy azok puffereit fölszabadítsa. Ezen felül arról is értesíti *A*-t, hogy újabb három TPDU-t küldhet 1-es fölötti sorszámmal (tehát a 2-es, 3-as és 4-est). *A* tudja, hogy a kettes sorszámút már továbbította, úgy gondolja, hogy elküldheti a rákövetkező kettőt, és így is tesz. Ezen a ponton blokkolódik, és további pufferek lefoglalására kell, hogy várakozzon. Viszont az időtúllépés miatti újraküldés blokkolt állapotban is lehetséges (9-es sor), mivel ilyenkor

	A	Üzenet	B	Megjegyzés
1	→	< 8 puffer kérése >	→	A 8 puffert szeretne
2	←	< nyug = 15, puff = 4 >	←	B csak 0-3 sorszámú üzeneteket engedélyezi
3	→	< sorsz = 0, adat = m0 >	→	A-nak 3 puffere maradt
4	→	< sorsz = 1, adat = m1>	→	A-nak 2 puffere maradt
5	→	< sorsz = 2, adat = m2 >	...	Az üzenet elveszett, de A azt hiszi, hogy már csak egy puffere maradt
6	←	< nyug = 1, puff = 3 >	←	B nyugtázza 0 és 1 sorszámúakat, 2-4-et engedélyezi
7	→	< sorsz = 3, adat = m3 >	→	A-nak 1 puffere maradt
8	→	< sorsz = 4, adat = m4 >	→	A-nak nincs több puffere, le kell állnia
9	→	< sorsz = 2, adat = m2 >	→	A időzítése lejár és újraküldi a 2-es üzenetet
10	←	< nyug = 4, puff = 0 >	←	Minden nyugta megérkezett, de A még mindig blokkolt
11	←	< nyug = 4, puff = 1 >	←	A most küldheti az 5-ös üzenetet
12	←	< nyug = 4, puff = 2 >	←	B valahol talált egy újabb puffert
13	→	< sorsz = 5, adat = m5 >	→	A-nak 1 puffere maradt
14	→	< sorsz = 6, adat = m6 >	→	A most ismét blokkolódik
15	←	< nyug = 6, puff = 0 >	←	A továbbra sem küldhet
16	...	< nyug = 6, puff = 4 >	←	Potenciális holtpont

6.16. ábra. *Dinamikus pufferfoglalás. A nyilak az átvitel irányát mutatják, a három pont (...) elveszett TPDU-t jelöl*

az elküldendő TPDU már pufferben van. A 10-es sorban *B* nyugtázza minden TPDU megérkezését a 4-es sorszámúval bezárólag (azt is beleértve), de nem hagyja *A*-t tovább adni. Ez a helyzet nem fordulhat elő a 3. fejezetben tárgyalt rögzített ablakméretű protokolloknál. A következő *B* által küldött TPDU újabb puffert foglal, így *A* továbbhaladhat.

Ilyen jellegű pufferfoglalási módszernél problémát jelenthet, ha a szállítási réteg datagram alhálózatra épül, és egy vezérlő TPDU vész el. Nézzük a 16. sort! *B* újabb puffereket foglalt le *A* részére, de a TPDU, amivel erről *A*-t tájékoztatná, elveszett. Mivel a vezérlő TPDU-k nem kapnak sorszámot, és időzítő se figyeli hiányukat, az *A* holtpontba kerül. Ezen helyzet elkerülésére mindkét hoszt rendszeres időközönként vezérlő TPDU-kat küld társának, amelyekben nyugta és az összeköttetésekhez tartozó pufferek állapotáról szóló információ van. Ily módon előbb-utóbb föloldódik a holtpont.

Eddig hallgatólagosan feltételeztük, hogy a küldő adási sebességének egyedül a vevő szabad puffereinek száma szab határt. A memóriaárak rohamos csökkenése miatt viszont lehetséges annyi memóriát zsúfolni a hosztokba, hogy a szabad pufferek hiánya ritkán vagy egyáltalán nem okoz problémát.

Amikor a pufferterület többé nem korlátozza a maximális forgalmat, egy újabb szűk keresztmetszet bukkan föl: a hálózat szállítókapacitása. Ha szomszédos routerek

legfeljebb *x* keret/s sebességre képesek, és *k* független út van két hoszt között, semmilyen módon nem valósíthat meg a szóban forgó két hoszt *kx* TPDU/s-nál nagyobb sebességű átvitelt, akármennyi szabad puffere van is a két félnek. Ha a küldő túl sűrűn ad (tehát *kx*-nél több TPDU-t küld másodpercenként), az alhálózatban torlódás keletkezik, mert nem képes olyan gyorsan továbbítani a TPDU-kat, mint amilyen gyorsan kapja azokat.

Valójában olyan mechanizmusra van szükség, amely az alhálózat szállítókapacitására, és nem a vevő pufferfoglalási lehetőségeire épít. Világos, hogy a forgalomszabályozási algoritmust a küldőnél kell megvalósítani, nehogy túl sok általa küldött TPDU nyugtázatlan maradjon. Belsnes (1975) egy olyan csúszóablakos forgalomszabályozási módszert javasolt, melyben a küldő dinamikusan a hálózat szállítási kapacitásához igazítja az ablak méretét. Ha a hálózat másodpercenként *c* TPDU átvitelére képes, és a ciklusidő *r* (ami magában foglalja a küldési, átviteli és vevő várakozási soraiban eltöltött időt, a vevő által végzett feldolgozás idejét és a nyugta visszaérkezésének idejét), a küldő ablakmérete *cr*. Ekkora ablakmérettel normális állapotban a küldő folyamatosan tele csővel dolgozik, így a hálózati teljesítőképesség legkisebb csökkenése is a küldő blokkolását okozza.

Az ablakméret rendszeres beállításához a küldőnek mindkét paramétert folyamatosan figyelnie kell, és abból az ablak méretét újra kell számolnia. A hálózat szállítási kapacitása egyszerűen meghatározható úgy, hogy adott időtartamig számolja a beérkező nyugták számát, majd ezt az időintervallum hosszával elosztja. A mérés során a küldőnek olyan gyorsan kell adnia, amilyen gyorsan csak bír, hogy ne a kis adási sebesség, hanem a hálózat teljesítőképessége jelentse a fölső korlátot a beérkező nyugták számára. Az éppen elküldött TPDU-ra beérkező nyugta késleltetését pontosan mérni lehet, és a mérési adatok átlagát az adó folyamatosan számolhatja. Mivel a hálózat kapacitása függ az éppen zajló forgalomtól, az ablakméretet gyakran pontosítani kell, hogy jól kövesse a hálózat teljesítményének ingadozását. Mint később látni fogjuk, az Internet hasonló mechanizmust alkalmaz.

6.2.5. Nyalábolás

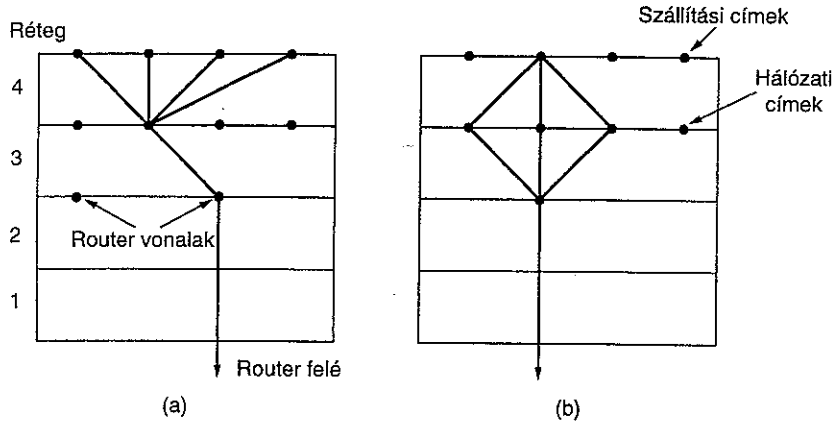
Több kommunikáció összeköttetésekre, virtuális áramkörökre, fizikai kapcsolatokra nyalábolása (multiplexing) fontos szerepet játszik a hálózati architektúra több rétegében. A szállítási rétegben a nyalábolásra több okból is szükség lehet. Például, azokban a hálózatokban, amelyek virtuális áramköröket alkalmazó alhálózatokra épülnek, minden felépített összeköttetés teljes élettartama alatt táblahelyet foglal a routerekben. Ha ezen fölül puffert is rendelnek a routerek minden virtuális áramkörhöz, egy felhasználó, aki kávészünet idejére egy távoli gépre bejelentkezve hagyja terminálját, fölöslegesen foglalja a drága erőforrásokat. Bár a csomagkapcsolás ilyen megvalósítási módja megkérdőjelezi a csomagkapcsolás első helyre való rangsorolásának fő indokát, miszerint a számlázás alapja nem a kapcsolási idő, hanem az átvitt adatok mennyisége, több szolgáltató ezt a megközelítést követi, mert ez emlékeztet leginkább a vonalkapcsolás modelljére, amihez hosszú évtizedek során hozzászoktak.

Egy olyan díjszabás, ami keményen bünteti a hosszú ideig sok virtuális áramkört

nyitva tartó elrendezéseket, vonzóvá teszi több szállítási összeköttetés egyazon hálózati összeköttetésen keresztül történő megvalósítását. A nyalábolás ezen formáját, melyet **felfelé nyalábolásnak (upward multiplexing)** nevezünk, mutatjuk be a 6.17.(a) ábrán. A példában négy különböző szállítási entitás közös hálózati összeköttetést (pl. ATM virtuális áramkört) használ a távoli hoszt eléréséhez. Amikor a kapcsolási idő adja a szolgáltató számlájának fő tételét, a szállítási réteg feladata a szállítási összeköttetéseket céljuk szerint csoportosítani, és minden csoportot minimális számú hálózati összeköttetésre leképezni. Ha túl sok szállítási összeköttetést rendel egy hálózati összeköttetéshez, a rendszer teljesítménye lecsökken, mert az ablak rendszerint tele lesz, és a felhasználóknak ki kell várni a sorukat, hogy egy újabb üzenetet elküldhessenek. Ha ellenben túl kevés szállítási összeköttetést hordoz a hálózati összeköttetés, akkor a szolgálat túlságosan drága lesz. Amikor felfelé nyalábolást ATM hálózaton alkalmazunk, az az ironikus (tragikus?) helyzet áll elő, hogy az összeköttetést a szállítási fejrész egy mezeje azonosítja, bár az ATM minden virtuális úton több mint 4000 virtuális áramkör sorszámot biztosít erre a célra.

A nyalábolás más indokból is hasznos lehet a szállítási rétegben. Ez inkább a szolgáltatók technikai döntésével, mint árpolitikájával függ össze. Tegyük föl például, hogy egy fontos felhasználónak időnként nagy sávszélességű összeköttetésre van szüksége. Ha az alhálózat n bites sorszámokkal dolgozó csúszóablak használatát teszi lehetővé, a felhasználónak meg kell állnia, miután $2^n - 1$ csomagot elküldött, és meg kell várnia, míg eljutnak a távoli hoszthoz és a nyugták visszaérkeznek. Ha a fizikai összeköttetés műholdon keresztül valósul meg, a felhasználó lényegében csak $2^n - 1$ csomagot tud továbbítani 540 ms-onként. Ha például $n = 8$ és a csomagok 128 bájt hosszúak, a kihasználható sávszélesség nagyjából 484 kb/s, bár a fizikai csatorna sávszélessége ennek több mint százszorosa.

Egy lehetséges megoldás, ha a szállítási réteg több hálózati összeköttetést létesít, és a forgalmat ciklikusan megosztja közöttük, mint azt a 6.17.(b) ábrán láthatjuk. Ezt a működési módot **lefelé nyalábolásnak (downward multiplexing)** nevezzük. Egyide-



6.17. ábra. *Nyalábolási módok. (a) Felfelé nyalábolás. (b) Lefelé nyalábolás*

jűleg k hálózati összeköttetést használva a tényleges sávszélesség k-szorosára nő. 4095 virtuális áramkörrel, 128 bájtos csomagok és 8 bites sorszámok használatával elméletileg 1,6 Gb/s átviteli sebesség is elérhető. Természetesen ez a teljesítmény csak akkor érhető el, ha a kimenő vonal lehetővé teszi az 1,6 Gb/s sebességű átvitelt, mivel a 4095 virtuális áramkör – legalábbis a 6.17.(b) ábrán – közös fizikai csatornán kap helyet. Ha több kimenő vonal áll rendelkezésre, lefelé nyalábolással még tovább növelhető a teljesítmény.

6.2.6. Összeomlás utáni helyreállítás

Ha a routerek és hosztok összeomlás veszélyének vannak kitéve, fontos kérdéssé válik a feléledés megvalósítása. Ha a teljes szállítási entitás a hoszton belül van megvalósítva, a hálózat és a routerek összeomlás utáni helyreállítása nyilvánvaló. Ha a hálózati réteg datagram szolgálatot nyújt, a szállítási entitások mindig számolnak elveszett TPDU-kkal, és tudják, hogyan kezeljék azokat. Ha a hálózati réteg összeköttetés alapú szolgálatot biztosít, egy megszakadt virtuális áramkör kezelése úgy történhet, hogy a szállítási entitás egy újat hoz létre, és azon megpróbálja megtudni távoli társától, hogy mely TPDU-kat kapta meg, és melyek vesztek el. Ez utóbbiakat aztán újraküldi.

Ennél sokkal komolyabb probléma a hosztok összeomlás utáni újraindítása. Általános cél lehet, hogy a kliensek a szerver összeomlása és gyors föléledése után tovább tudják folytatni munkájukat. A nehézség illusztrálására tegyük föl, hogy az egyik hoszt, a kliens, egy nagy állományt küld a másik hoszt, az állományszolgáltató számára egyszerű megáll-és-vár (stop-and-wait) protokollal. A szerver szállítási rétege egyszerűen egyenként átadja a beérkező TPDU-kat a szállítási felhasználónak. Az átvitel közepén a szerver összeomlik. Amikor újraéled, az összes táblázatát újrainicializálja, így nem tudja, hogy hol tartott.

Előző állapotának kiderítése érdekében a szerver minden kliens részére elküldhet egy TPDU-t, melyben bejelenti, hogy tönkrement, és mindenkit kér, hogy küldjék el használt összeköttetéseik állapotát. Minden kliens az alábbi két állapot egyikében lehet: egy elküldött TPDU van függőben (*S1*), vagy nincs ilyen TPDU (*S0*). Csupán ezen információ birtokában el kell döntenie a kliensnek, hogy újraküldje-e a legutolsó TPDU-t, vagy sem.

Első pillantásra nyilvánvalónak tűnhet, hogy a kliensnek, amikor megtudja, hogy a szerver összeomlott, csak akkor kell újraküldenie a kérdéses TPDU-t, ha az nyugtázatlan (azaz ha *S1* állapotban van). Közelebbi vizsgálatok során azonban kiderül ennek a naiv megközelítésnek a veszélye. Vegyük például azt a helyzetet, amikor a szerver szállítási entitása elküldi a nyugtát, és miután a nyugta elindult, csak ezután adja át a TPDU-t az alkalmazói folyamatnak. A TPDU kimenő folyamba írása és a nyugta elküldése két külön oszthatatlan esemény, melyeket nem lehet egyszerre végrehajtani. Ha az összeomlás a nyugta elküldése után, de még a TPDU átadása előtt történik meg, a kliens megkapja a nyugtát, és így *S0* állapotban lesz, amikor az újraindulás bejelentése megérkezik. A kliens ekkor nem fogja újraküldeni a TPDU-t, mivel (helytelenül) abban a tudatban él, hogy az megérkezett. Ezen döntése egy hiányzó TPDU-t eredményez.

Ezen a ponton azt gondolhatjuk: „Ez a probléma könnyen megoldható. Csak annyit

kell tenni, hogy átírjuk a szállítási entitást, és ezentúl először a folyamba ír és csak azután küldi a nyugtát."

Játsszuk el a fönti kísérletet ismét! Képzeljük el, hogy a folyamba írás már megtörtént, de az összeomlás éppen a nyugta elküldése előtt következik be. A kliens így *S1* állapotban lesz, és újraküldi a TPDU-t, ami így észrevétlenül megkettőződik a szerver alkalmazásnak átadott kimenő folyamban.

Mindegy, hogyan programozzuk a szervert és a klienst, mindig vannak olyan helyzetek, melyekben ez a protokoll kudarcot vall. A szerver kétféleképpen valósítható meg: először nyugtáz vagy először ír. A kliens négyféleképpen programozható: mindig újraküldi az utolsó TPDU-t, sohasem küldi újra, csak az *S0* állapotban küldi újra, vagy csak az *S1*-ben. Ez nyolc kombinációt jelent, de mint látni fogjuk, ezek közül mindegyikhez van egy eseményhalmaz, aminek hatására a protokoll hibázik.

A szerver oldalán három esemény lehetséges: nyugta küldése (*Ny*), kimeneti folyamba írás (*K*) és összeomlás (*Ö*). A három esemény hat különböző sorrendben történhet: *NyÖ(K)*, *NyKÖ*, *Ö(NyK)*, *Ö(KNy)*, *KNyÖ* és *KÖ(Ny)*, ahol a zárójelek azt jelölik, hogy az összeomlást már sem nyugtázás, sem írás nem követi (azaz ha a rendszer összeomlott, akkor tényleg összeomlott). A 6.18. ábrán a kliens és szerver stratégiák nyolc kombinációját mutatjuk be az érvényes eseménysorrendekkel együtt. Vegyük észre, hogy minden stratégiára található valamilyen eseménysorrend, amire a protokoll kudarcot vall. Például, ha a kliens újraküld, az *NyKÖ* eseménysorozat észrevétlen kettőzést eredményez, miközben a másik két sorozatra helyesen működik a protokoll.

A protokoll további finomítása sem segít. Még ha a szerver és a kliens – mielőtt a szerver írni próbálna – több TPDU-t váltana is egymással, hogy a kliens pontosan tudja, mi fog történni, arról semmiképpen sem szerezhet tudomást, hogy az összeomlás közvetlenül az írás előtt vagy utána közvetlen következett-e be. A következtetés elkerülhetetlen: alapszabályunk – mely szerint az események nem történhetnek párhuzamosan – kizárja annak lehetőségét, hogy a rendszerösszeomlás és újraindulás a felső rétegek számára transzparens módon mehessen végbe.

Általánosabban szólva, ez az eredmény azt jelenti, hogy *N*. réteg összeomlásából

A vevő hoszt (szerver) stratégiája

Az adó hoszt (kliens) stratégiája	Először nyugta, aztán kiírás			Először kiírás, aztán nyugta		
	NyÖ(K)	NyKÖ	Ö(NyK)	Ö(KNy)	KNyÖ	KÖ(Ny)
Mindig újraküld	OK	KETTŐZ	OK	OK	KETTŐZ	KETTŐZ
Soha nem ad újra	VESZT	OK	VESZT	VESZT	OK	OK
S0-ban újraküld	OK	KETTŐZ	VESZT	VESZT	KETTŐZ	OK
S1-ben újraküld	VESZT	OK	OK	OK	OK	KETTŐZ

OK = a protokoll helyesen működik
KETTŐZ = a protokoll üzenetet kettőz
VESZT = a protokoll üzenetet veszít

6.18. ábra. *A kliens és szerver stratégia különböző kombinációi*

történő újraindulás csak az $N+1$. réteg segítségével lehetséges, feltéve, hogy a magasabb réteg elegendő információt tárol az alatta levő állapotáról. Mint fönt említettük, a szállítási réteg akkor képes kezelni a hálózati rétegben történt összeomlásokat, ha az összeköttetés mindkét vége állandóan nyomon követi, hogy hol tartanak.

Ez a probléma végül elvezet ahhoz a kérdéshez, hogy mit is jelent valójában az ún. végpontok közötti nyugtázás. Alapjában véve a szállítási protokoll két végpont között működik, és nem láncolt, mint az alatta levő rétegek. Most vegyük azt az esetet, amikor a felhasználó a távoli adatbázisban tranzakciókat kezdeményez. Tegyük föl, hogy a távoli szállítási entitás programja szerint először átadja a TPDU-t a fölötte levő rétegnek, és ezután nyugtáz. Még ebben az esetben sem jelenti a nyugta vétele a felhasználó gépén, hogy a távoli hoszt addig működőképes maradt, amíg a tranzakció lezárása meg nem történt. Egy igazi végpontok közötti nyugtát – melynek vétele azt jelenti, hogy a tevékenység ténylegesen végbement, és hiánya annak elmaradását jelzi – valószínűleg lehetetlen elérni. A kérdésről bővebben Saltzer és munkatársai (1984) művében olvashatunk.

6.3. Egyszerű szállítási protokoll

Az eddig tárgyalt gondolatok kézzelfoghatóbbá tétele érdekében ebben az alfejezetben részletesen megvizsgálunk egy példa szállítási protokollt. A modellt úgy választottuk ki, hogy eléggé valósághű legyen, ugyanakkor az érthetőség kedvéért viszonylag egyszerű maradjon. A példa tárgyalása során használt absztrakt szolgálati primitívek a 6.3. ábrán látható összeköttetés alapú primitívek.

6.3.1. A példa szolgálati primitívjei

Első gondunk az, hogy miképpen írhatjuk le pontosan ezeket a szállítási primitíveket. A CONNECT esete egyszerű: lesz egy *connect* könyvtári eljárásunk, ami az összeköttetés felépítéséhez a megfelelő paraméterekkel meghívható. Paraméterei a helyi és távoli TSAP-ok (szállítási szolgálat elérési pontok, azaz szállítási címek). A rutin végrehajtása közben, mialatt a szállítási entitás összeköttetést próbál létesíteni, a hívó blokkolódik (azaz futása fölfüggesztődik). Ha az összeköttetés sikeresen felépült, a hívó blokkolása megszűnik, és elkezdheti az adatátvitelt.

Amikor egy folyamat bejövő hívásokra akar várakozni, a figyelni kívánt TSAP címmel meghívja a *listen* eljárást. A folyamat ezután addig blokkolva marad, amíg egy távoli folyamat összeköttetést nem próbál létesíteni ehhez a TSAP-hoz.

Megjegyezzük, hogy ez a modell erősen aszimmetrikus. Az egyik oldal passzív, egy *listen* végrehajtása után vár, amíg nem történik valami. A másik fél aktív, az kezdeményezi az összeköttetést. Felmerül egy érdekes kérdés: mi a teendő, ha az aktív fél kezdi előbb a tevékenységét. Az egyik stratégia szerint a próbálkozás összeköttetés létesítésére kudarcot vall, ha a távoli TSAP-on nem figyel senki. Egy másik megközelítésben viszont a hívó is blokkolódik (akár örökre), amíg egy hallgató fel nem tűnik.

A példánkban használt megoldás kompromisszum a két lehetőség között. A fogadó oldal az összeköttetés-kérést adott ideig fenntartja. Ha azon a hoszton egy folyamat *listen* hívást hajt végre mielőtt az időzítő lejár, az összeköttetés létrejön. Ellenkező esetben a fogadó oldal a kérést elutasítja, a hívó blokkolása megszűnik, és hiba visszajelzést kap.

Az összeköttetés bontásához a *disconnect* eljárást fogjuk alkalmazni. Miután mindkét fél bekapcsolódott, az összeköttetés lebomlik, más szóval szimmetrikus összeköttetés-bontási modellt használunk.

Az adatátvitellel pontosan ugyanaz a probléma, mint az összeköttetés felépítésével: az adat elküldése aktív tevékenység, de a vétel passzív folyamat. Ugyanazt a megoldást fogjuk használni adatátvitelre, mint összeköttetés-létesítésre. Megvalósítunk egy aktív *send* hívást, ami továbbítja az adatot, és egy passzív *receive* hívást, ami a vevőt egy TPDU beérkezéséig blokkolja.

Konkrét szolgálatdefiníciónk így öt primitívből áll: CONNECT, LISTEN, DISCONNECT, SEND és RECEIVE. Mindegyik primitív pontosan annak a könyvtári rutinnak felel meg, amely végrehajtja azt. A szolgálati primitívek és a könyvtári eljárások paraméterei a következők:

```
össz_szám  = LISTEN(helyi)
össz_szám  = CONNECT(helyi, távoli)
státus     = SEND(össz_szám, puffer, bájtszám)
státus     = RECEIVE(össz_szám, puffer, bájtszám)
státus     = DISCONNECT(össz_szám)
```

A LISTEN primitív bejelenti a hívó szándékát összeköttetés elfogadására a megadott TSAP-ra. A primitív addig blokkolja használóját, amíg nem érkezik összeköttetés-létesítési kérés. Nincs időzítés.

A CONNECT primitív két paramétert vár, a *helyi*-ben egy helyi TSAP-ot (szállítási címet) és *távoli*-ban egy távoli címet. A két TSAP között megpróbál összeköttetést létesíteni. Ha sikeres, egy nemnegatív *össz_szám* értékkel tér vissza, ami a további hívások során az összeköttetést azonosítja. Sikertelen kísérlet esetén a kudarc okát negatív *össz_szám* értékkel jelzi a primitív. Az általunk használt egyszerű modellben minden TSAP csak egyetlen összeköttetésben szerepelhet, így a hiba oka az is lehet, hogy az egyik TSAP már foglalt. Emellett hibát okozhat a távoli gép üzemzavara, érvénytelen helyi vagy távoli cím használata.

A SEND primitív a megadott szállítási összeköttetésen elküldi a puffer tartalmát mint üzenetet, a jelzett szállítási összeköttetésen. Túl hosszú üzenet átvitele esetleg több darabra vágva történik. A *státus*-ban jelzett lehetséges hibaokok: nincs összeköttetés, illegális puffercím vagy negatív *bájtszám* paraméter.

A RECEIVE primitív jelzi a hívó adatvételi szándékát. A beérkező üzenet méretét a *bájtszám* paraméterben kell megadni. Ha a távoli folyamat már lebontotta az összeköttetést, vagy érvénytelen a puffercím (pl. az alkalmazói programon kívüli területre mutat), a *státus* visszatérési értéke a probléma okát jelző hibakód lesz.

A DISCONNECT primitív befejezi az *össz_szám* paraméterrel jelzett szállítási összeköttetést. Lehetséges hibák: *össz_szám* egy másik folyamathoz tartozik vagy érvénytelen az összeköttetés-azonosító. A *státus* visszatérési értéke a hibakód vagy siker esetén 0.

6.3.2. Egy példa szállítási entitásra

Mielőtt rátérnénk a példa szállítási entitás programkódjának tárgyalására, gondoljuk végig, hogy az itt bemutatott példa analóg a 3. fejezetben látott korai példákkal: sokkal inkább pedagógiai, mint komoly felhasználói célokat szolgál. Az egyszerűség kedvéért sok, valódi rendszerekben nélkülözhetetlen technikai részletet (mint pl. a kimerítő hibakezelést) kihagytunk belőle.

A szállítási réteg a hálózati szolgálati primitívek segítségével küld és fogad TPDU-kat. Ki kell tehát választanunk a példában használt hálózati szolgálati primitíveket. Egy lehetséges választás a megbízhatatlan datagram szolgálat, de hogy egyszerű maradjon a példa, nem ezt használjuk. A megbízhatatlan datagram szolgálat használata esetén a szállítási primitívek kódja nagy és összetett lenne, mert leginkább elveszett és késleltetett csomagokkal kéne bajlódnia. Emellett ezeket a módszereket már részletesen megvizsgáltuk a 3. fejezetben.

Ehelyett összeköttetés alapú megbízható hálózati szolgálatot fogunk használni. Ezzel a választással olyan, a szállítási rétegben felmerülő témákra tudunk összpontosítani, melyek nem fordulnak elő az alsóbb rétegekben. Többek között ide tartozik az összeköttetés felépítése és lebontása, és a kreditek kezelése. Példánk leginkább az ATM hálózatra épülő egyszerű szállítási szolgálathoz fog hasonlítani.

Általában véve a szállítási entitás része lehet a hoszt operációs rendszerének, de megvalósítható felhasználói programként futó könyvtári rutinok formájában is. Emellett előfordulhat, hogy társprocesszoron vagy hálózati illesztőkártyán helyezkedik el. Az egyszerűség kedvéért példánkat úgy alakítottuk ki, mintha egy programkönyvtári csomag lenne, de minimális változtatással az operációs rendszer részévé is lehet tenni (elsősorban a pufferek elérési módját kell módosítani).

Érdemes azonban megjegyeznünk, hogy ebben a példában a „szállítási entitás" egyáltalán nem különálló funkcionális elem, hanem a felhasználói folyamat része. Így, amikor a felhasználó egy blokkoló primitívet (mint például a LISTEN) hajt végre, a teljes szállítási entitás szintén blokkolódik. Amíg ez a tervezés megfelelő egy egyetlen felhasználói folyamatot tartalmazó hoszt esetén, többfelhasználós hoszt esetén természetesebb lenne, ha a szállítási entitás a felhasználói folyamatoktól független külön processzként lenne megvalósítva.

A hálózati réteg felé kialakított interfészt a *to_net* és *from_net* eljárások képezik. Mindkettőnek hat paramétere van. Az első az összeköttetés azonosítója, amely egy egytípusú leképezést végez a hálózati virtuális áramkörökre. Ezt követik a Q és M bitek, melyek egybe állítása vezérlőüzenetet, illetve az adott üzenet további csomagjait jelöli. Ezután a csomag típusa következik, ami a 6.19. ábrán látható hat csomagtípus közül az egyik lehet. Végül az adat kezdetét jelző mutató és az adatbájtok száma szerepel.

A szállítási entitás *to_net* hívás előtt az összes paramétert kitölti a hálózati réteg számára, hogy az olvassa ki a *from_net* hívás során a paramétereket a beérkező csomagból a szállítási entitás számára. Az információt eljárás paramétereiben adjuk át a hálózati rétegnek, és nem magát az elküldött vagy beérkező csomagot, így a szállítási rétegnek nem kell ismernie a hálózati protokoll részleteit. Ha a szállítási entitás akkor próbálna elküldeni egy csomagot, amikor az alatta levő virtuális áramkör csúszóabla-

Hálózati csomag	Jelentés
CALL REQUEST (CALL_REQ)	Összeköttetés-létesítés kérése
CALL ACCEPTED (CALL_ACC)	Válasz CALL REQUEST csomagra
CLEAR REQUEST (CLEAR_REQ)	Összeköttetés-bontás kérése
CLEAR CONFIRMATION (CLEAR_CONF)	Válasz CLEAR REQUEST csomagra
DATA (DATA_PKT)	Adatcsomag
CREDIT (CREDIT)	Vezérlőcsomag az ablakkezeléshez

6.19. ábra. *A példánkban használt hálózati csomagok*

ka tele van, a *to_net* eljárással együtt blokkolódik, amíg szabad hely nem lesz az ablakban. Ez a mechanizmus a szállítási entitás számára teljesen átlátszó, vezérlését – *enable_transport_layer* és *disable_transport_layer* jellegű parancsokkal – a hálózati réteg biztosítja a 3. fejezetben leírt protokollokhoz hasonló módon. A csomag csúszóablakának kezelését szintén a hálózati réteg végzi.

A fönt említett transzparens blokkoló mechanizmuson kívül explicit *sleep* és *wakeup* eljárásokat is (melyeket itt nem részletezünk) használ a szállítási entitás. A *sleep* rutint akkor hívja a szállítási entitás, amikor külső eseményre – rendszerint egy csomag érkezésére – várva logikailag blokkolódik. A *sleep* hívás után a szállítási entitás (és vele együtt természetesen a felhasználói folyamat is) felfüggesztődik.

A szállítási entitás tényleges kódját a 6.20. ábrán láthatjuk. Minden összeköttetés mindenkor az alábbi hét állapot valamelyikében van:

1. IDLE – Összeköttetés még nem létesült.

2. WAITING – CONNECT eljárás lefutott, és a CALL REQUEST csomagot elküldte.

3. QUEUED – Egy CALL REQUEST csomag beérkezett, de még nem történt LISTEN hívás.

4. ESTABLISHED – Az összeköttetés létrejött.

5. SENDING – A felhasználó engedélyre vár a csomag elküldéséhez.

6. RECEIVING – Egy RECEIVE hívás történt.

7. DISCONNECTING (DISCONN) – Egy helyi DISCONNECT hívás történt.

Állapotátmenet a következő események hatására történik: primitív végrehajtása, csomag érkezése vagy időtúllépés.

A 6.20. ábrán látható eljárások két típusba sorolhatók. Legtöbbjüket közvetlenül hívja a felhasználói program, azonban a *packet_arrival* és *clock* rutinok eltérően működnek. Spontán aktivizálódnak külső események hatására: csomag érkezésére, illetve óraütemre. Ezek valójában megszakítás-kezelő rutinok. Feltételezzük, hogy sohasem kerülnek meghívásra szállítási entitás eljárás végrehajtása közben, csak akkor kerülhet


```
#define MAX_CONN 32                 /* egyidejű összeköttetések maximális száma */
#define MAX_MSG_SIZE 8192           /* üzenet maximális hossza bájtokban */
#define MAX_PKT_SIZE 512            /* csomag maximális hossza bájtokban */
#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;

/* globális változók */
transport_address listen_address;   /* a hallgatni kívánt helyi cím */
int listen_conn;                    /* összeköttetés-azonosító hallgatásra */
unsigned char data[MAX_PKT_SIZE];   /* átmeneti terület adatcsomagok részére */

struct conn {
  transport_address local_address, remote_address;
  cstate state;                     /* az összeköttetés állapota */
  unsigned char *user_buf_addr;     /* mutató a vételi pufferre */
  int byte_count;                   /* adási/vételi mennyiség */
  int clr_req_received;             /* CLEAR_REQ vételekor beállítva */
  int timer;                        /* CALL_REQ csomagok időtúllépés számlálója */
  int credits;                      /* elküldhető csomagok száma */
} conn[MAX_CONN];

/* függvénydeklarációk */
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)
{       /* A felhasználó egy összeköttetésre vár. Ellenőrizzük, hogy érkezett-e
        CALL_REQ csomag. */
  int i = 1, found = 0;

  for (i = 1; i <= MAX_CONN; i++)       /* A táblázatban CALL_REQ -t keresünk. */
```

6.20. ábra. *Egy példa szállítási entitásra*

```
      if (conn[i].state == QUEUED && conn[i].local_address == t) {
        found = i;
        break;
      }
    if (found == 0) {
      /* Nincs várakozó CALL_REQ.  Amíg nem érkezik egy, vagy le nem jár az időzítő,
      aludj.*/
      listen_address = t; sleep(); i = listen_conn ;
    }
    conn[i].state = ESTABLISHED;          /* Az összeköttetés létrejött. */
    conn[i].timer = 0;                    /* Időzítőt nem használjuk. */
    listen_conn = 0;                      /* Érvénytelen címet állítunk be. */
    to_net(i, 0, 0, CALL_ACC, data, 0);   /* A hálózatot összeköttetés elfogadására
                                          szólítjuk fel. */
    return(i);                            /* Visszatérési érték az összeköttetés
                                          azonosítója. */
}

int connect(transport_address l, transport_address r)
{       /* A felhasználó egy távoli folyamattal akar összeköttetést létesíteni;
        CALL_REQ csomagot küldünk. */
  int i;
  struct conn *cptr;

  data[0] = r;  data[1] = l;              /* CALL_REQ csomag tartalma */
  i = MAX_CONN;                           /* Visszafelé keresünk a táblázatban. */
  while (conn[i].state != IDLE && i > 1) i = i - 1;
  if (conn[i].state == IDLE) {
    /* A CALL_REQ csomag elküldését jelző bejegyzést hozunk létre. */
    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();                              /* CALL_ACC vagy CLEAR_REQ csomagra
                                          várunk. */
    if (cptr->state == ESTABLISHED) return(i);
    if (cptr->clr_req_received) {
      /* A másik fél elutasította a kérést. */
      cptr->state = IDLE;                 /* Visszatérünk IDLE állapotba. */
      to_net(i, 0, 0, CLEAR_CONF, data, 0);
      return(ERR_REJECT);
    }
  } else return(ERR_FULL);                /* Sikertelen CONNECT: nincs hely a
                                          táblázatban. */
}
```

6.20. ábra. *Egy példa szállítási entitásra (folytatás)*


```
int send(int cid, unsigned char bufptr[], int bytes)
{       /* A felhasználó üzenetet akar küldeni. */
  int i, count, m;
  struct conn *cptr = &conn[cid];

  /* SENDING állapotba lépünk. */
  cptr->state = SENDING;
  cptr->byte_count = 0;                   /* az üzenet már továbbított bájtjainak száma */
  if (cptr->clr_req_received == 0 && cptr->credits == 0) sleep();
  if (cptr->clr_req_received == 0) {
    /* Van hitelünk, szükség esetén több csomagra vágjuk az üzenetet. */
    do {
      if (bytes - cptr->byte_count > MAX_PKT_SIZE) {      /* többcsomagos üzenet */
        count = MAX_PKT_SIZE;  m = 1;    /* további csomagok következnek */
      } else {                            /* egycsomagos üzenet */
        count = bytes - cptr->byte_count;  m = 0;     /* utolsó csomag az üzenetből */
      }
      for (i = 0; i < count; i++) data[i] = bufptr[cptr->byte_count + i];
      to_net(cid, 0, m, DATA_PKT, data, count);     /* Egy csomagot elküldünk. */
      cptr->byte_count = cptr->byte_count + count;  /* Növeljük a már továbbított
                                                    bájtok számát. */
    } while (cptr->byte_count < bytes); /* ciklus amíg a teljes üzenet el nem ment */
    cptr->credits--;                      /* Minden üzenet egy hitelt használ fel. */
    cptr->state = ESTABLISHED;
    return(OK);
  } else {
    cptr->state = ESTABLISHED;
    return(ERR_CLOSED);                   /* Sikertelen átvitel: a társentitás bontást
                                          kezdeményezett. */
  }
}

int receive(int cid, unsigned char bufptr[], int *bytes)
{ /* A felhasználó fölkészül egy üzenet vételére. */
  struct conn *cptr = &conn[cid];

  if (cptr->clr_req_received == 0) {
    /* Az összeköttetés él, próbálunk venni. */
    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); /* CREDIT csomagot küldünk. */
    sleep();                              /* Blokkol, míg adat nem érkezik. */
    *bytes = cptr->byte_count;
  }
```

6.20. ábra. *Egy példa szállítási entitásra (folytatás)*

```
    cptr->state = ESTABLISHED;
    return(cptr->clr_req_received ? ERR_CLOSED : OK);
}

int disconnect(int cid)
{  /* A felhasználó bontani akarja az összeköttetést. */
   struct conn *cptr = &conn[cid];

   if (cptr->clr_req_received) {          /* A másik fél már bontást kezdeményezett. */
      cptr->state = IDLE;                 /* Az összeköttetés véget ért. */
      to_net(cid, 0, 0, CLEAR_CONF, data, 0);
   } else {                               /* Mi kezdeményezünk bontást. */
      cptr->state = DISCONN;              /* Nem történik bontás, míg a társentitás bele
                                             nem egyezik. */
      to_net(cid, 0, 0, CLEAR_REQ, data, 0);
   }
   return(OK);
}

void packet_arrival(void)
{       /* Egy csomag megérkezett, fogjuk és feldolgozzuk. */
   int cid;                               /* az összeköttetés, melyen a csomag befutott */
   int count, i, q, m;
   pkt_type ptype;                        /* CALL_REQ, CALL_ACC, CLEAR_REQ,
                                             CLEAR_CONF, DATA_PKT, CREDIT */
   unsigned char data[MAX_PKT_SIZE];      /* a beérkező csomag adatrésze */
   struct conn *cptr;

   from_net(&cid, &q, &m, &ptype, data, &count);      /* elhozzuk */
   cptr = &conn[cid];

   switch (ptype) {
      case CALL_REQ:                      /* A távoli felhasználó összeköttetést akar
                                             létrehozni. */
         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:                      /* A távoli felhasználó elfogadta a CALL_REQ
                                             csomagunkat. */
         cptr->state = ESTABLISHED;
         wakeup();
         break;
```

6.20. ábra. *Egy példa szállítási entitásra (folytatás)*


```
      case CLEAR_REQ:                    /* A távoli felhasználó bontani akarja az
                                            összeköttetést, vagy visszautasítja hívásunkat. */
         cptr->clr_req_received = 1;
         if (cptr->state == DISCONN) cptr->state = IDLE;   /* elkerüljük az ütközést */
         if (cptr->state ==  WAITING || cptr->state == RECEIVING || cptr->state ==
                                                                    SENDING) wakeup();
         break;
      case CLEAR_CONF:                   /* A távoli felhasználó beleegyezik a bontásba. */
         cptr->state = IDLE;
         break;
      case CREDIT:                       /* A távoli felhasználó adatra vár. */
         cptr->credits += data[1];
         if (cptr->state == SENDING) wakeup();
         break;
      case DATA_PKT:                     /* A távoli felhasználó adatot küldött. */
         for (i = 0; i < count; i++) cptr->user_buf_addr[cptr->byte_count + i] = data[i];
         cptr->byte_count += count;
         if (m == 0 ) wakeup();
   }
}

void clock(void)
{       /* Ugrott egyet az óramutató, megvizsgáljuk a várakozó összeköttetés-
           kérések időzítéseit. */
   int i;
   struct conn *cptr;
   for (i = 1; i <= MAX_CONN; i++) {
      cptr = &conn[i];
      if (cptr->timer > 0) {             /* Jár az időzítő. */
         cptr->timer--;
         if (cptr->timer == 0) {         /* Most járt le az időzítő. */
            cptr->state = IDLE;
            to_net(i, 0, 0, CLEAR_REQ, data, 0);
         }
      }
   }
}
```

6.20. ábra. *Egy példa szállítási entitásra (folytatás)*

rájuk a vezérlés, amikor a felhasználói program alszik, vagy a szállítási entitáson kívüli része fut. Ennek biztosítása elengedhetetlen a szállítási entitás helyes működésének érdekében.

A csomag fejrészében elhelyezkedő Q (Qualifier – megkülönböztető) bit lehetővé teszi a szállítási protokoll fejrész elhagyását, ami továbbítási overheaddel járna. A közönséges adatüzeneteket $Q = 0$ beállítással küldjük. A szállítási protokoll vezérlőüzenetei – melyekből példánkban csak egy van (CREDIT) – $Q = 1$ értékkel ellátott adatüzenetek. A fogadó szállítási entitás detektálja és feldolgozza ezeket az üzeneteket.

A szállítási entitásban használt legfontosabb adatstruktúra az *conn* tömb, melyben minden lehetséges összeköttetés számára egy rekord van fönntartva. Ebben nyilvántartjuk az összeköttetés állapotát, beleértve mindkét végpont szállítási címét, az összeköttetésen elküldött és fogadott üzenetek számát, a jelenlegi állapotot [a fordító megjegyzése: ez beleérthető az „összeköttetés állapotá"-ba], a felhasználói puffer mutatóját, az aktuális üzenet már elküldött vagy vett bájtjainak számát, egy bitet, amely jelzi, hogy a távoli felhasználó DISCONNECT hívást adott ki, egy időzítőt, és egy hitelszámlálót, ami az üzenetek küldését szabályozza. Egyszerű példánkban nem használjuk az összes itt felsorolt mezőt, de egy összetett szállítási entitás esetleg további mezőket igényelne. Feltételezzük, hogy inicializáláskor minden *conn* bejegyzés *IDLE* állapothoz rendelődik.

Amikor a felhasználó meghívja a CONNECT rutint, a szállítási entitás a hálózati réteget CALL REQUEST csomag küldésére utasítja, a felhasználó folyamatot pedig elaltatja. Amint a másik félhez megérkezik a CALL REQUEST csomag, a szállítási entitás megszakítást kap, melyet a *packet_arrival* eljárás kezel le. Megvizsgálja, hogy figyel-e felhasználó a megadott címen. Ha igen, CALL ACCEPTED csomagot küld vissza, aminek hatására a távoli felhasználó felébred, különben a CALL REQUEST *IDŐTARTAM* számú óraütésig várakozó sorban várakozik. Ha ezen idő alatt LISTEN hívás történik, az összeköttetés létrejön, különben lejár az időzítő és a szállítási entitás a kérést egy CLEAR REQUEST csomaggal elutasítja. Ez a mechanizmus azért szükséges, hogy elkerülhessük a kezdeményező örökös blokkolását, amikor a távoli folyamat nem kíván vele összeköttetést létrehozni.

Jóllehet kiküszöböltük a szállítási protokoll fejrész használatát, mivel egyidejűleg több összeköttetés létezhet, szükségünk van egy olyan módszerre, melynek segítségével nyomon követhetjük, hogy mely csomag melyik szállítási összeköttetéshez tartozik. A legegyszerűbb megközelítés szerint használjuk a hálózati réteg virtuális áramkör számát az összeköttetés azonosítójaként is. Ezenfelül a virtuális áramkör sorszáma indexként használható az *conn* tömbben. Amennyiben egy csomag a hálózati réteg *k* sorszámú virtuális áramkörén érkezik, a *k*-ik szállítási összeköttetéshez tartozik, melynek állapotát az *conn*[*k*] rekord tartalmazza. Hosztban kezdeményezett összeköttetések esetén az összeköttetés sorszámát a kezdeményező szállítási entitás választja. Beérkező hívások esetén a hálózati rétegé a választás joga. Tetszőleges, eddig nem használt virtuális áramkör sorszámot választhat.

Ahhoz, hogy ne kelljen puffereket fenntartani és kezelni a szállítási entitáson belül, a hagyományos csúszóablakos módszertől eltérő forgalomszabályzási mechanizmust alkalmaztunk. Amikor a felhasználó meghívja a RECEIVE eljárást, egy speciális **hitelüzenetet (credit message)** küld az adási oldal szállítási entitásának, ami ezt feljegyzi az *conn* tömbben. SEND végrehajtásakor a küldő szállítási entitás megvizsgálja, hogy érkezett-e hitel az adott összeköttetésen. Ha igen, elküldi az üzenetet (szükség esetén több csomagban), és csökkenti a hitelszámlálót, különben újabb hitelüzenet érkezéséig alvó állapotba lép. Ez a mechanizmus garantálja, hogy a küldő fél nem továbbít üzenetet, amíg a fogadó nem hajtott végre RECEIVE eljárást. Ennek eredményeként mindig garantáltan lesz szabad puffer a beérkező üzenet számára. Ez a módszer egyszerűen általánosítható oly módon, hogy a vevők többszörös puffereket kapjanak, és így többszörös üzeneteket is kérhessenek.

Figyeljük meg a 6.20. ábrán látható program egyszerűségét. Egy valóságos szállítási entitás minden felhasználó által biztosított paraméter érvényességét ellenőrizné, támogatná a hálózati réteg összeomlása utáni újraindulást, foglalkozna a hívásütközésekkel, és egy jóval általánosabb szállítási szolgálatot biztosítana olyan lehetőségekkel, mint megszakítások, datagramok és a receive és send primitívek nem blokkolódó változatai.

6.3.3. A példa, mint véges állapotú gép

Egy szállítási entitás megírása nehéz és munkaigényes feladat, különösen összetettebb szállítási protokollok használata esetén. A hibázás lehetőségének csökkentése érdekében gyakran hasznos lehet a protokoll állapotát véges állapotú gépként ábrázolni.

Már láttuk, hogy mintaprotokollunk összeköttetésenként hét állapottal rendelkezik. Tizenkét esemény különböztethető meg, melyek hatására az összeköttetés egyik állapotból a másikba léphet. Ezek közül öt az öt szolgálati primitív meghívása, további hat a hat különböző legális csomagtípus, az utolsó pedig az időtúllépés. A 6.21. ábrán láthatjuk a protokoll főbb tevékenységeit mátrix formában. Az oszlopok az állapotoknak, a sorok a 12 eseménynek felelnek meg.

A mátrix (azaz a véges automata) összes bejegyzése legfeljebb három mezővel rendelkezhet: egy predikátummal, egy tevékenységgel és az új állapottal. A predikátum azt jelzi, hogy milyen feltételek mellett fut le a tevékenység. Például a bal fölső mezőben, ha LISTEN-t hajtunk végre és nincs több hely a táblázatban (*P1* predikátum), a LISTEN hívás sikertelen lesz és nem történik állapotváltás. Másrészt, ha olyan címre érkezik CONNECT REQUEST, amelyen már hallgat egy felhasználó (*P2* predikátum), az összeköttetés azonnal létrejön. Egy másik lehetőség az, ha *P2* hamis, tehát nem érkezett CALL REQUEST csomag, így az összeköttetés *IDLE* állapotban marad továbbra is CALL REQUEST-re várakozva.

Érdemes rámutatnunk, hogy a mátrixban használt állapotokat nem határozza meg teljes egészében maga a protokoll. Ebben a példában nincs *LISTENING* állapot, ami a LISTEN végrehajtásának logikus következménye lehetne. Erre azért nincs szükség, mert az állapot az összeköttetés bejegyzéséhez van hozzárendelve, és a LISTEN nem hoz létre új rekordot az összeköttetés számára. Miért nem? Mert úgy döntöttünk, hogy a hálózati réteg virtuális áramkör sorszámait használjuk az összeköttetések azonosítóiként. A LISTEN számára a virtuális áramkör számát a hálózati réteg határozza meg, amikor a CALL REQUEST csomag megérkezik.

Az *A1–A12* tevékenységek a főbb tevékenységek, úgy mint csomag küldése vagy időzítő elindítása. Nem soroltuk fel az összes melléktevékenységet (például egy összeköttetés rekordjának inicializálása). Ha egy tevékenység végrehajtása során egy alvó folyamatot is fölébresztünk, az ébredés utáni tevékenységek szintén számítanak. Például ha egy CALL REQUEST csomag érkezik, melyre egy alvó folyamat várakozik, a CALL REQUEST csomag által kiváltott tevékenységbe beletartozik a CALL ACCEPT csomag elküldése közvetlenül ébredés után. Miután az összes tevékenység lefutott, az összeköttetés új állapotba léphet, ahogy azt a 6.21. ábrán láthatjuk.

Állapot

		Idle	Waiting	Queued	Established	Sending	Receiving	Dis-connecting
Primitívek	LISTEN	P1: ~/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			
Beérkező csomagok	Call_req	P3: A1/Estab P3: A4/Queu'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		
Időzítő	Időtúllépés			~/Idle				

Predikátumok

P1: Betelt az összeköttetések táblázata
P2: Call_req függőben
P3: LISTEN függőben
P4: Clear_req függőben
P5: Van felhasználható hitel

Tevékenységek

A1: Call_acc küldése
A2: Várakozás Call_req-ra
A3: Call_req küldése
A4: Időzítő indítása
A5: Clear_conf küldése
A6: Clear_req küldése
A7: Üzenet küldése
A8: Várakozás Creditre
A9: Credit küldése
A10: Clr_Req-Vétel 1-re állítása
A11: Credit feljegyzése
A12: Üzenet elfogadása

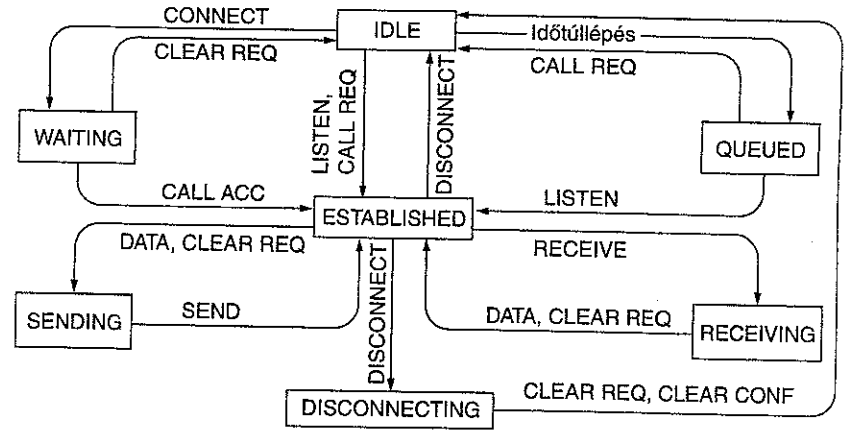
6.21. ábra. *A mintaprotokoll, mint véges állapotú gép. Minden bejegyzés egy opcionális predikátumot, egy opcionális tevékenységet és egy új állapotot tartalmaz. A hullám (~)azt jelzi, hogy nincs jelentősebb esemény. Felülvonás a predikátum felett annak negáltját jelenti. Az üres mezők érvénytelen vagy lehetetlen eseményt jelölnek*

A protokoll mátrix formában történő ábrázolása háromszoros előnnyel jár. Egyrészt ebben az alakban jóval könnyebb a programozónak szisztematikusan ellenőrizni az események és állapotok összes kombinációjában, hogy szükséges-e valamilyen tevékenységet elvégezni. Valóságos alkalmazásoknál egyes kombinációk hibakezelésre használhatók. A 6.21. ábrán nem különböztettük meg a lehetetlen és az érvénytelen szituációkat. Például, ha egy összeköttetés *várakozik* állapotban van, a DISCONNECT esemény lehetetlen, mert a felhasználó blokkolva van, és egyáltalán nem hajthat végre primitíveket. Másrészt *küld* állapotban a szállítási entitás nem vár adatcsomagot, mert nem adott hitelt a távoli entitásnak. Adatcsomag érkezése ilyenkor protokollhibát jelent.

A protokoll mátrix alakú ábrázolásának második előnye megvalósításkor jelentkezik. Képzeljünk el egy kétdimenziós tömböt, amelynek minden $a[i][j]$ eleme egy mutató vagy index egy eljárásra, ami az i esemény kezelésére szolgál, ha az összeköttetés a j állapotban van. Egy lehetséges megvalósítás az, amikor a szállítási entitást egy rövid ciklus formájában írjuk meg, melynek elején az entitás egy esemény bekövetkezésére várakozik. Amikor az esemény megtörtént, az entitás megkeresi a megfelelő összeköttetést, és megállapítja az állapotát. Az esemény és az állapot ismeretében a szállítási entitás egyszerűen meghívja az a tömb megfelelő eljárását. Ez a megközelítés a mi példánknál jóval szabályosabb, szisztematikusabb tervezést tesz lehetővé.

A véges automata megközelítés harmadik előnye a protokollok leírásánál jelentkezik. Némely szabvány dokumentumában a protokollok a 6.21. ábráéhoz hasonló véges állapotú géppel vannak megadva. Az ilyen jellegű leírástól a működő szállítási entitásig jóval egyszerűbb út vezet, ha a szállítási entitást a szabványban rögzítetteken alapuló véges állapotú gép vezérli.

A véges automata megközelítés legfőbb hátránya az, hogy esetleg jóval nehezebben érthető, mint a kezdetben használt közvetlenül kódolt példánk. Ezt a problémát azonban részben megoldhatjuk úgy, hogy a véges állapotú gépet gráfként is ábrázoljuk, mint azt a 6.22. ábrán is láthatjuk.



6.22. ábra. *A mintaprotokoll grafikus formában. Az összeköttetés állapotát nem módosító állapotátmeneteket elhagytuk az áttekinthetőség kedvéért*

6.4. Internet szállítási protokollok (TCP és UDP)

Az Internet szállítási rétegében két fő protokoll található, egy összeköttetés alapú és egy összeköttetés nélküli. A következő alfejezetben mindkettőt tanulmányozni fogjuk. Az összeköttetés alapú protokoll a TCP, az UDP pedig az összeköttetés nélküli. Mivel az UDP lényegében csak egy rövid fejrésszel kibővített IP, a TCP-re fogunk összpontosítani.

A **TCP-t (Transmission Control Protocol)** megbízhatatlan hálózatok összekapcsolására épülő megbízható két végpont közötti bájtfolyam biztosítására tervezték. Egy összekapcsolt hálózat jelentős mértékben eltér egy különálló hálózattól, mert különböző részei erősen eltérő topológiával, sávszélességgel, késleltetéssel, csomagmérettel és egyéb tulajdonságokkal rendelkezhetnek. A TCP-t úgy fejlesztették ki, hogy dinamikusan alkalmazkodjon az összekapcsolt hálózatok tulajdonságaihoz, és sok hiba tekintetében robusztusan viselkedjen.

A TCP-t hivatalosan az RFC 793-ban definiálták. Ahogy telt az idő, különböző hibákat és inkonzisztenciákat fedeztek föl benne, és némely területen a követelmények is megváltoztak. A szükséges módosításokat és javításokat az RFC 1122 részletezi, a bővítéseket az RFC 1323 tartalmazza.

Minden TCP-t támogató gépnek van egy TCP szállítási entitása, akár felhasználói program formájában, akár a kernel részeként, amely a TCP adatfolyamok menedzselésével foglalkozik, és interfészt biztosít az IP réteg felé. A TCP entitás felhasználói adatfolyamokat kap a helyi folyamatoktól, 64 kilobájtot meg nem haladó darabokra tördeli azokat (a gyakorlatban ez a méret 1500 bájt körül mozog), és minden darabot külön IP datagramként továbbít. Amikor a TCP adatot tartalmazó IP datagram megérkezik egy gépre, átkerül a TCP entitáshoz, amely rekonstruálja az eredeti bájtfolyamot. A későbbiekben az egyszerűség kedvéért néha a „TCP" rövidítéssel jelöljük a TCP szállítási entitást (a programot) a szoftver egy darabját vagy a TCP protokollt (szabályok halmazát). A szövegkörnyezetből egyértelműen kiderül, hogy melyikre gondolunk, például az „A felhasználó átadja az adatot a TCP-nek." mondatban nyilvánvaló, hogy a TCP szállítási entitásról van szó.

Az IP réteg nem garantálja a datagramok biztonságos kézbesítését, így a TCP feladata időzítések segítségével a hibákat jelezni és a csomagokat szükség esetén újraküldeni. Ha még rendben meg is érkeznek a datagramok, előfordulhat, hogy megváltozik a sorrendjük, ezért szintén a TCP dolga helyes sorrendben üzenetekké alakítani azokat. Egyszóval, a TCP-nek kell a legtöbb felhasználó által elvárt és az IP által nem támogatott megbízhatóságot biztosítani.

6.4.1. A TCP szolgálati modell

A TCP szolgálat úgy valósul meg, hogy mind a küldő, mind a fogadó létrehoz egy csatlakozónak (socket) nevezett végpontot, ahogy azt a 6.1.3. pontban tárgyaltuk. Minden csatlakozónak van egy száma, azaz csatlakozócíme, ami a hoszt IP címéből és

egy hoszton belül érvényes 16 bites számból, a **port** azonosítójából tevődik össze. A port a TCP környezetben használt elnevezése a TSAP-nak. A TCP szolgálat megvalósításához egy közvetlen összeköttetést kell létesíteni a küldő és fogadó gép csatlakozói (socketjei) között. A csatlakozókat kezelő hívásokat a 6.6. ábrán foglaltuk össze.

Egy csatlakozó egyidejűleg több összeköttetés kezelésére is használható, azaz két vagy több összeköttetés közös csatlakozóban is végződhet. Az összeköttetéseket a két végükön található csatlakozók azonosítói azonosítják: (*socket1, socket2*). Nem használnak virtuális áramkör sorszámot vagy más azonosítót.

Az 256-nál kisebb portszámokat **jól ismert portoknak** nevezzük, és standard szolgálatok részére vannak fenntartva. Például, ha egy folyamat összeköttetést akar létesíteni egy hoszttal, hogy oda fájlt továbbítson FTP felhasználásával, a célhoszt 21-es portjára csatlakozhat, hogy kapcsolatba lépjen az ottani FTP démonnal. Hasonló módon TELNET-tel történő távoli bejelentkezés megvalósításához a 23-as port használható. A jól ismert portok listáját az RFC 1700 tartalmazza.

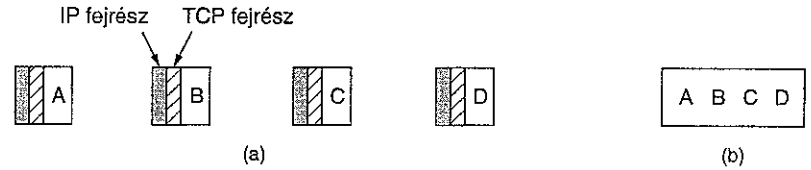
Minden TCP összeköttetés duplex és két pont közötti. A „duplex" azt jelenti, hogy egyidejűleg mindkét irányú forgalom lehetséges. A két pont közötti összeköttetésnek pontosan két végpontja van. A TCP nem támogatja a többesküldést (multicasting) vagy az adatszórást (broadcasting).

A TCP összeköttetés bájtfolyam és nem üzenetfolyam jellegű, az üzenetek határai nem maradnak változatlanok a két végpont között. Például, ha a küldő folyamat négy 512 bájtos blokkot ír egy TCP folyamba, az adatok a vevőhöz megérkezhetnek négy 512 bájtos darabban, két 1024 bájtos részletben, egy 2048 bájtos blokkban (lásd a 6.23. ábrát), vagy más formában. A fogadónak nem áll módjában megállapítani, hogy milyen egységekben küldték az adatot.

A UNIX fájlok szintén ilyen tulajdonságúak. A fájlt olvasó folyamat nem tudja, hogy blokkonként, bájtonként esetleg egy darabban írták-e a fájlt. A TCP-t használó programoknak a UNIX fájlok esetéhez hasonlóan fogalmuk sincs arról, hogy melyik bájt mit jelent. Egy bájt csupán bájt.

Amikor egy alkalmazás adatot ad át a TCP-nek, a TCP magánügye, hogy azonnal továbbítja, vagy (annak érdekében, hogy egyszerre nagyobb mennyiséget küldhessen el) pufferbe teszi az adatot.

Néha viszont az alkalmazás megköveteli az adat késedelem nélküli továbbítását. Tegyük fel például, hogy a felhasználó bejelentkezett egy távoli gépre. Miután begépelte a parancssort, és leütötte az újsor billentyűt, lényeges, hogy a begépelt sor azonnal eljusson a távoli gépre, és ne várakozzon a pufferben, míg egy újabb sort be nem ír



6.23. ábra. *(a) Négy 512 bájtos blokk mint különálló IP datagramok. (b) Egyetlen* READ *hívással az alkalmazásnak átadott 2048 bájtnyi adat*

a felhasználó. Az adat továbbítását az alkalmazások a PUSH jelzőbit beállításával kényszeríthetik ki. Ez jelzi a TCP-nek, hogy ne késleltesse az átvitelt.

Néhány korai alkalmazás a PUSH bitet üzenethatárok jelzésére használta. Habár van, amikor működik ez a trükk, néha kudarcot vall, mivel nem minden TCP implementáció adja át a PUSH bitet a vevőoldali alkalmazásnak. Ezenkívül ha az első PUSH elküldése előtt újabbak érkeznek (például ha foglalt a kimenő vonal), a TCP-nek jogában áll az összes PUSH bittel jelzett adatot egyetlen IP datagramba összefogni, ahol az egyes darabok már nem lesznek különválasztva.

A TCP utolsó, említésre érdemes szolgálata a **sürgős adat (urgent data).** Amikor egy interaktív felhasználó DEL-t vagy CTRL–C-t üt egy már megkezdett távoli számítás megszakítására, a küldő alkalmazás további vezérlőinformációt ad az adatfolyamhoz, és *sürgős* jelzéssel átadja a TCP-nek. Ennek hatására a TCP abbahagyja az adatok egybegyűjtését az adott összeköttetésre, és azonnal továbbítja az eddig összegyűlt adatot.

Amikor a sürgős adat célba ér, a vevő alkalmazás végrehajtása megszakad (UNIX terminológiában *signal*-t kap), abbahagyja, amit éppen csinált és beolvassa az adatfolyamot, hogy megtalálja a sürgős adatot. A sürgős adat vége jelezve van, így az alkalmazás tudja, hogy meddig tart. A sürgős adat kezdetén ezzel ellentétben nincs jelzés, azt az alkalmazásnak kell megtalálnia. Ez a megoldás csak egy kezdetleges jelzési rendszert biztosít, és minden más feladatot az alkalmazásra hagy.

6.4.2. A TCP protokoll

Ebben a részben általános áttekintést adunk a TCP protokollról. A következőben a fejrész mezőit vizsgáljuk meg részletesen. Minden TCP összeköttetésen továbbított bájt rendelkezik egy 32 bites sorszámmal. Egy 10 Mb/s sebességű LAN-on teljes sebességgel adó hoszt sorszámai elméletileg nagyjából egy óra alatt körbeérhetnének, de a gyakorlatban ez jóval tovább tart. A sorszámokat egyaránt használják nyugtázásra és a csúszóablakos mechanizmushoz, amely külön 32 bites mezőket használ a fejrészben.

A küldő és fogadó TCP entitások között szegmensek formájában folyik az adatcsere. A **szegmens** egy fix 20 bájtos fejrészből (amit további opcionális rész követhet) és nulla vagy több adatbájtból áll. A TCP szoftver dönti el, hogy mekkorák legyenek a szegmensek. Különböző írások adatait egy szegmensbe gyűjtheti, vagy egyetlen írás tartalmát több szegmensre oszthatja. A szegmens méretére mindössze két korlátozás van. Egyrészt minden szegmensnek – beleértve a TCP fejrészt is – el kell férnie a 65 535 bájtos IP adat mezőben. Másrészt minden hálózatban van egy **leghosszabb átvihető adategység (maximum transfer unit, MTU)** korlát. A szegmens mérete nem haladhatja meg az MTU méretét. Gyakorlatban az MTU általában néhány ezer bájt, így ez jelent a szegmens méretére felső korlátot. Ha egy szegmens áthalad egy sor hálózaton anélkül, hogy a hosszát változtatni kellene, majd egy olyanba kerül, amelynek MTU értéke kisebb a szegmens hosszánál, akkor a hálózatok határán levő router a szegmenst két vagy több kisebb szegmensre darabolja.

Egy szegmenst, ha túl nagy az azt továbbító hálózat számára, a router több kisebb szegmensre darabolhat. Minden újabb szegmens saját IP fejrészt kap, így a szegmens

darabolása többletterhet jelent (minden további szegmens 20 bájt extra fejrész információt mint IP fejrészt kap).

A TCP entitások fő protokollja a csúszóablakos protokoll. Amikor a küldő egy szegmenst továbbít, egy időzítőt is elindít. Amint a szegmens célba ér, a fogadó TCP entitás visszaküld egy szegmenst (amely adatot is tartalmaz, ha volt adat, különben üres) egy nyugtasorszámmal, ami megegyezik a következő bájt várt sorszámával. Ha a küldő időzítője a nyugta vétele előtt lejár, újraküldi a szegmenst.

Bár ez a protokoll egyszerűnek tűnik, számos rejtett részletet fogunk érinteni az alábbiakban. Például, mivel a szegmenseket szét lehet darabolni, elképzelhető, hogy az elküldött szegmens egy része megérkezik, de az összes többi elvész. A szegmensek rossz sorrendben is célba érhetnek, így bár a 3072–4095 sorszámú bájtok megérkezhettek, nem lehet elküldeni a nyugtát, mert a 2048–3071 sorszámú bájtok még nem bukkantak elő. A szegmensek késleltetése továbbítás közben olyan nagy is lehet, hogy a küldő entitás időzítése lejár, és újraküldi a késő szegmenseket. Ha egy újraküldött szegmens az eredetitől eltérő útvonalon halad, melynek során a korábbitól eltérő módon darabolódik fel, előfordulhat, hogy az eredeti és a másolat részei összekeveredve érkeznek meg. Ekkor csak gondos adminisztrációval lehet megbízható bájtfolyamot elérni. Emellett az Internetet alkotó nagyszámú hálózatok között a szegmens útja során alkalmanként eldugult vagy összeomlott hálózatra kerülhet.

A TCP-t fel kell készíteni ezen problémák hatékony kezelésére. Jelentős erőfeszítések folytak a TCP folyamok teljesítőképességének optimalizálására, figyelembe véve a hibás hálózatok okozta problémákat is. Az alábbiakban számos olyan algoritmust tárgyalunk, melyeket több TCP implementációban is alkalmaztak.

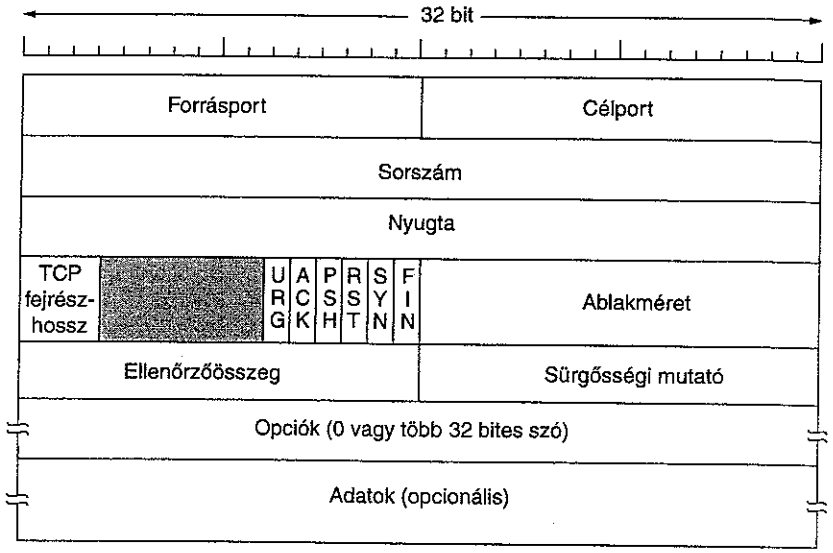
6.4.3. A TCP szegmens fejrésze

A 6.24. ábrán láthatjuk a TCP szegmens felépítését. Minden szegmens egy fix kiosztású 20 bájtos fejrésszel kezdődik, amit fejrészopciók követhetnek. Ezek után – ha van – legfeljebb 65 535 – 20 – 20 = 65 495 bájt adat állhat. A kivonandók közül az első 20 bájt az IP, a második 20 bájt a TCP fejrészt jelenti. Az adatot nem tartalmazó szegmensek érvényesek, általában nyugtázásra és vezérlő szegmensként használják őket.

Vegyük szemügyre a TCP fejrész minden mezőjét! A *forrásport (source port)* és *célport (destination port)* mezők azonosítják az összeköttetés helyi végpontjait. Minden hoszt maga döntheti el, hogyan foglaljon portokat 1024-től kezdődően. Egy portszám és a hoszt IP címe együtt alkotja a 48 bites egyedi TSAP-ot. A forrás és cél csatlakozószámok (socket numbers) együttese azonosítja az összeköttetést.

A *sorszám (sequence number)* és *nyugtaszám (acknowledgement number)* mezők szerepe szokásos. Jegyezzük meg, hogy az utóbbi a következő várt bájt sorszámát tartalmazza, nem az utolsó rendben beérkezett bájtét. Mindkét mező 32 bit széles, mivel a TCP folyamban minden adatbájt sorszámot visel.

A *TCP fejrészhossz (TCP header length)* mondja meg, hány 32 bites szóból áll a TCP fejrész. Ez az információ azért szükséges, mert a fejrész mérete az *opciók (options)* mező változó hossza miatt szintén változó. Tulajdonképpen ez a mező jelzi az



6.24. ábra. *A TCP fejrész*

adat kezdetét (32 bites szavakban mérve) a szegmensen belül, de mivel ez egyben a fejrész szavakban mért hossza is, a végeredmény ugyanaz.

Ezután egy használaton kívüli 6 bites mező következik. A TCP jól átgondolt tervezésére szolgál tanúbizonyságul ezen mező több mint egy évtizedes változatlan állapota. Kevésbé jól sikerült protokollokban már fölhasználták volna az eredeti változat hibáinak kiküszöbölésére.

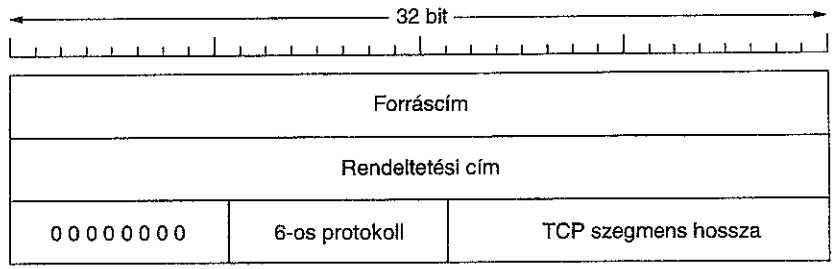
Ezt hat egybites mező követi. Az *URG* bit értéke 1, ha *sürgősségi mutatót* használt. A *sürgősségi mutató* a sürgős adat bájtban mért helyét jelzi a jelenlegi bájtsorszámhoz viszonyítva. Ez a mechanizmus a megszakítás üzeneteket helyettesíti. Mint korábban említettük, ez a végsőkig leegyszerűsített módszer lehetőséget ad a küldőnek, hogy jelzést küldjön a vevő felé anélkül, hogy a TCP-nek külön megszakításokkal kelljen foglalkoznia.

Az *ACK* bit 1 értéke jelzi a *nyugta* mező érvényességét. Ha $ACK = 0$, a szegmens nem tartalmaz nyugtát, tehát a *nyugta* mező figyelmen kívül hagyható.

A *PSH* bit jelzi a késedelem nélküli adat továbbítását (PUSH). Ez egyben a vevő felé is udvarias kérést jelent: ne pufferelje a vett adatot (amit amúgy hatékonysági okokból megtehetne), hanem azonnal továbbítsa az alkalmazás felé.

Az *RST* bit egy hoszt összeomlása, vagy más okból összezavart összeköttetés helyreállítására szolgál. Ezenkívül érvénytelen szegmens elutasítására és összeköttetés létesítésének megtagadására is használják. Rendszerint ha valaki $RST = 1$ értéket viselő szegmenst kap, akkor felkészülhet valamilyen probléma megoldására.

A *SYN* bit összeköttetés létesítésére szolgál. Az összeköttetés-kérésben $SYN = 1$ és $ACK = 0$ jelzi, hogy ráültetett *nyugta* mezőt nem használnak. Az összeköttetés-kérésel adott válaszban már van nyugta, így abban $SYN = 1$ és $ACK = 1$. Lényegében a *SYN* bit jelzi a CONNECTION REQUEST és CONNECTION ACCEPTED üzeneteket, melyeket az *ACK* bit különbözteti meg egymástól.



6.25. ábra. *A TCP ellenőrző összeg képzésében szerepelő pszeudofejrész*

A *FIN* bit szolgál egy összeköttetés bontására. Jelzi, hogy a küldőnek nincs több továbbítandó adata. Az összeköttetés bontása után azonban még korlátlan ideig folytathatja az adatok vételét. Mind a *SYN*, mind a *FIN* szegmensek rendelkeznek sorszámokkal, így garantált a helyes sorrendben történő feldolgozás.

A TCP forgalomszabályozása változó méretű csúszóablakkal történik. Az *ablakméret* mező határozza meg, hogy a *nyugtázott* bájttal kezdődően hány bájtot lehet elküldeni. Az *ablakméret* 0 értéke is érvényes, és azt jelzi, hogy a *nyugtázott* bájtnál eggyel kisebb sorszámú bájtok mind rendben megérkeztek, viszont a vevőnek nagy szüksége van egy kis pihenésre és köszöni szépen, több adatot jelenleg nem kér. Később azonos *nyugta* értékkel rendelkező, és nem nulla *ablakméret* mezővel ellátott szegmenssel engedélyezni lehet az adatok küldését.

A megbízhatóság érdekében van még egy *ellenőrző összeg* is a fejrészben. Ez ellenőrzi a fejrész, az adat és a 6.25. ábrán látható pszeudofejrész épségét. Ennek számításakor a TCP *ellenőrző összeg* mezeje 0 értéket kap, és az adatmező egy további bájttal bővül, ha a hossza eddig páratlan szám volt. Az ellenőrző összeg számítása egyszerűen a 16 bites szavak 1-es komplemens összeadásával történik, majd az összeg 1-es komplemensét vesszük. Ennek következményeként amikor a vevő az egész szegmensre – beleértve az *ellenőrző összeg* mezőt is – elvégzi ugyan ezt a számítást, az eredménynek 0-nak kell lennie.

A pszeudofejrész a forrás- és célhosztok 32 bites IP címét, a TCP protokoll azonosítóját (6), és a TCP szegmens (fejrésszel együtt mért) hosszát tartalmazza. Azzal, hogy a pszeudofejrész is részt vesz az ellenőrző összeg képzésében, a tévesen továbbított csomagok könnyebben detektálhatók, de ez a protokollhierarchiát is sérti, mert az itt szereplő IP címek nem a TCP réteghez, hanem az IP-hez tartoznak.

Az *opciók* mezőt a szabályos fejrészben nem szereplő lehetőségek megvalósítására tervezték. A legfontosabb lehetőség lehetővé teszi a hosztoknak, hogy meghatározzák a legnagyobb általuk elfogadható TCP adat mező nagyságát. Nagy szegmensek használata hatékonyabb kicsik alkalmazásánál, hiszen a 20 bájtos fejrész több adathoz tartozik, viszont kis hosztok esetleg képtelenek nagyon nagy szegmensek kezelésére. Összeköttetés létesítésekor minden hoszt megadhatja a saját maximumát és tudomást

szerezhet a partnerje maximum értékéről. Ha egy hoszt nem használja ezt a lehetőséget, az alapértelmezés 536 bájtos adat mező. Minden Internetre csatlakozó hoszt el kell hogy fogadjon 536 + 20 = 556 bájt hosszúságú szegmenseket, viszont a két irányban nem kell azonosnak lenni a maximális hosszaknak.

Nagy sávszélességgel vagy nagy késleltetéssel, esetleg mindkettővel rendelkező vonalak számára gyakran problémát jelent a 64 KB ablakméret. Egy T3 vonalon (44,736 Mb/s) csak 12 ms ideig tart egy 64 kilobájtos tele ablak továbbítása. Ha a teljes átviteli késleltetés 50 ms (tipikus kontinensek közötti fényvezető szálakra), a küldő a várakozási idő 3/4 részében tétlenül várakozik a nyugtára. Egy műholdas összeköttetésen még ennél is rosszabb a helyzet. Nagyobb ablakméret használata esetén a küldő tovább pumpálhatná az adatot, de 16 bites *ablakméret* mezővel nem lehet akkora méretet kifejezni. Megoldásul az RFC 1323-ban az ablak *skálatényező (window scale)* opció használatát javasolták, amely lehetővé tenné a küldő és a fogadó számára, hogy megegyezzenek egy ablak skála tényezőben. Ez a szám mindkét oldalon lehetővé teszi az *ablakméret* mező legfeljebb 14 bites balra történő eltolását, így legfeljebb 2^{30} bájtos ablakok használatát. A legtöbb TCP implementáció ma már támogatja ezt a lehetőséget.

Egy másik, az RFC 1106-ban javasolt és most már széles körben megvalósított opció a szelektív ismétlés, melyet az *n* visszalépéses protokoll helyett alkalmaznak. Ha a vevő kap egy rossz szegmenst, majd nagyszámú jó érkezik, a normális TCP protokoll szerint működő küldő időzítése végül lejár, és minden nyugtázatlan szegmenst újraküld, még azokat is, amelyek rendben megérkeztek. Az RFC 1106-ban bevezettek NAK-okat, hogy lehetővé tegyék a vevőnek egy adott szegmens (vagy szegmensek) újraátviteli kérését. Miután ezek megérkeztek, az összes pufferelt adatot nyugtázza. Ezzel a mechanizmussal csökkenthető az újraküldött adatok mennyisége.

6.4.4. TCP összeköttetések kezelése

Az összeköttetések létesítésére a 6.2.2. részben tárgyalt háromutas kézfogás technikát alkalmazzák a TCP-ben. Az összeköttetés létesítéséhez az egyik fél, nevezzük szervernek, passzívan várakozik a bejövő kérésekre a LISTEN és ACCEPT primitívek végrehajtásával. Ehhez megjelölhet egy adott forrást vagy nem jelöl ki senkit.

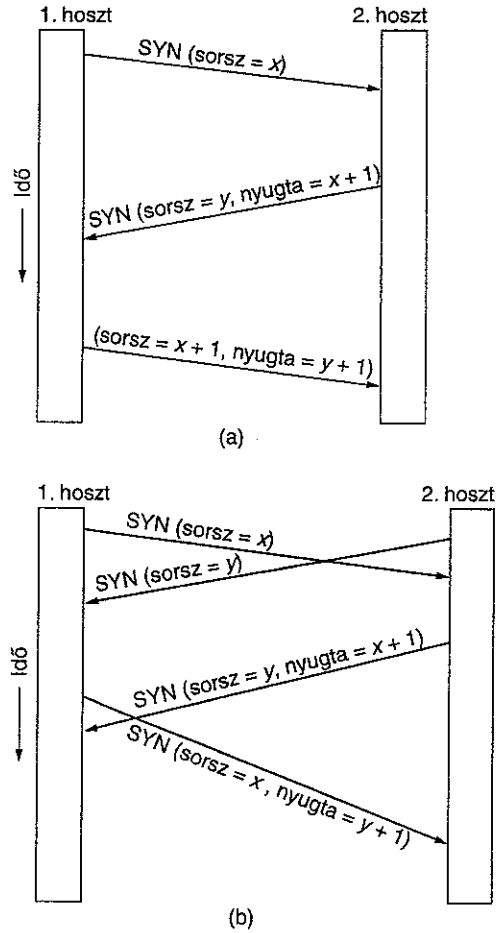
A másik oldal, melyet kliensnek hívunk, végrehajtja a CONNECT primitívet, melynek hívásakor rögzíti az IP címet, a használni kívánt port számát, az általa megengedett maximális TCP szegmens méretét és esetleg felhasználói adatot (pl. jelszót) is átad. A CONNECT primitív elküld egy TCP szegmenst $SYN = 1$ és $ACK = 0$ értékkel, majd választ vár.

Amikor a szegmens megérkezik a rendeltetési helyre, az ottani TCP entitás ellenőrzi, hogy létezik-e egy folyamat, amely a célport mezőben meghatározott porton végrehajtotta-e a LISTEN primitívet. Ha nem, $RST = 1$ válasszal elutasítja az összeköttetéskérést.

Ha valamilyen folyamat figyeli a megadott portot, akkor az megkapja a beérkező TCP szegmenst, és rajta múlik, hogy elfogadja-e vagy visszautasítja az összeköttetést. Ha elfogadja, egy nyugtázó szegmenst küld vissza. A 6.26.(a) ábrán láthatjuk az elküldött TCP szegmensek sorozatát normális esetben. Figyeljük meg, hogy a *SYN* szegmens egy bájtnak megfelelő sorszámot fogyaszt, így egyértelműen nyugtázható.

Abban az esetben, ha a két hoszt egyszerre próbál összeköttetést létesíteni ugyanazon két csatlakozó (socket) között, a 6.26.(b) ábrán látható eseménysorozat alakul ki. Ennek eredményeképpen csak egy összeköttetés jön létre kettő helyett, mivel az összeköttetéseket végpontjaik azonosítják. Ha a művelet során létrejövő első összeköttetés azonosítói (x, y), és a második is ilyen azonosítóval születik meg, csak egy (x, y) azonosítójú bejegyzés keletkezik a táblázatban.

Az összeköttetés kezdő sorszáma korábban már tárgyalt okokból kifolyólag nem 0. Órára alapozott módszert alkalmaznak, melyben az óra 4 μs-onként kettyen. A bizton-



6.26. ábra. *(a) TCP összeköttetés létesítés normális esetben. (b) Hívások ütközése*

ság növelése érdekében ha egy hoszt összeomlik, a maximális csomag élettartamig (120 s) nem indulhat újra. Így biztosítható, hogy a korábbi összeköttetésekből származó csomagok már nem bolyonganak szerte az Internetben, amikor újra működésbe lép.

Bár a TCP összeköttetések duplexek, hogy megérthessük az összeköttetések bontásának módját, legjobb két szimplex összeköttetésnek tekinteni azokat. Mindkét szimplex összeköttetés a másiktól függetlenül lebontható. Egy összeköttetés bontásához bármelyik fél küldhet egy *FIN* = 1 értékű TCP szegmenst, amivel jelzi, hogy nem szándékozik több adatot küldeni. Amint a *FIN* nyugtája megérkezik, az az irány lezárul. A másik irányban azonban ettől függetlenül korlátlanul folyhat adatátvitel. Amikor mindkét irányt lezárták, az összeköttetés befejeződött. Normális esetben egy összeköttetés bontásához négy TCP szegmens szükséges, egy *FIN* és egy *ACK* mindkét irányban. Az első *ACK* és a második *FIN* viszont elhelyezhető ugyanabban a szegmensben is, így összesen csak háromra van szükség.

Hasonlóan a telefonhívásokhoz, ahol mindkét ember egyszerre köszön el és teszi le a kagylót, itt is küldhet mindkét fél egy időben *FIN* szegmenst. Ezeket a szokásos módon nyugtázzák, majd az összeköttetés befejeződik. Valójában nincs lényeges eltérés az egyszerre vagy egymás után történő összeköttetés-bontások között.

A két-hadsereg probléma elkerülésére időzítőket használnak. Ha egy *FIN*-re nem érkezik válasz két csomag-élettartamnyi idő alatt, a *FIN* küldője bontja az összeköttetést. A másik fél végül észreveszi, hogy már senki sem figyel rá, az általa indított időzítő is lejár. Bár ez a megoldás nem tökéletes, az tény, hogy tökéletes megoldás elméletben sem létezhet, ezért ennek elégnek kell lenni. A gyakorlatban ritkán merülnek föl problémák.

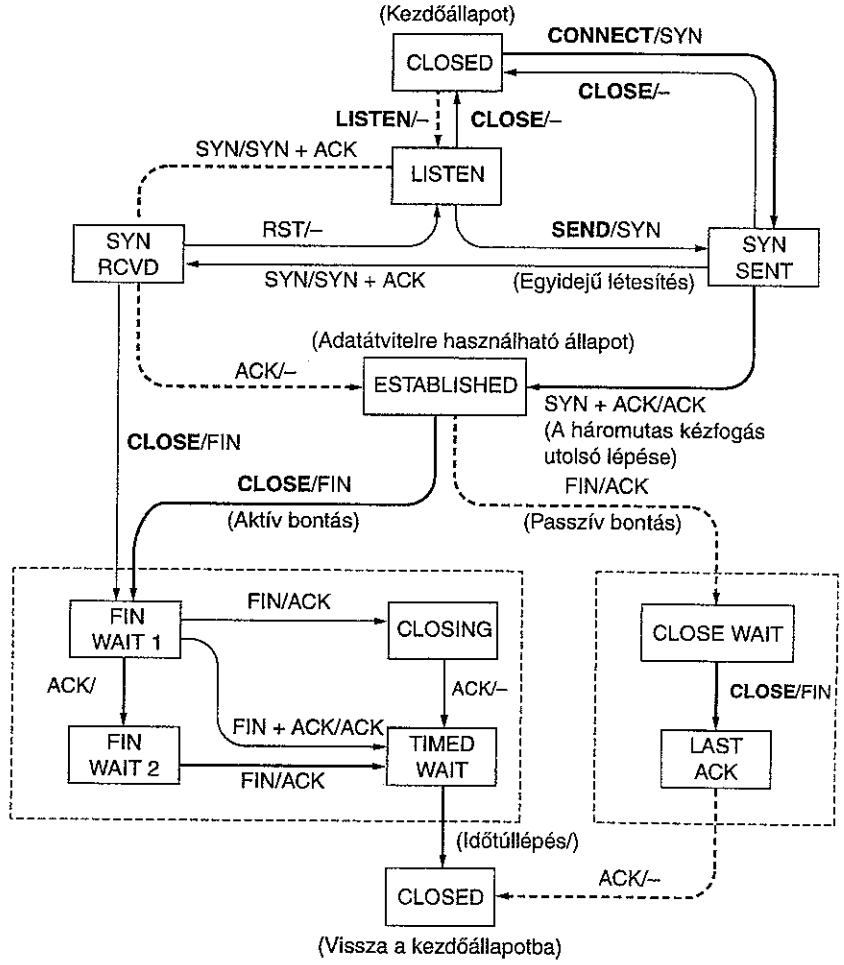
Összeköttetések létesítését és bontását véges automatával is modellezhetjük. A gép 11 állapotát a 6.27. ábrán láthatjuk. Minden állapotban az események bizonyos halmaza érvényes. Ha érvényes esemény történik, lejátszódik valamilyen tevékenység, más esemény hatására hibajelzés keletkezik.

Állapot	Leírás
CLOSED	Nincs aktív vagy függő összeköttetés
LISTEN	A szerver egy hívás beérkezésére vár
SYN RCVD	Összeköttetés-kérés érkezett, *ACK*-ra vár
SYN SENT	Az alkalmazás összeköttetés-létesítést kezdeményezett
ESTABLISHED	Normális adatátviteli állapot
FIN WAIT 1	Az alkalmazás bejelentette, hogy végzett teendőivel
FIN WAIT 2	A másik fél beleegyezett az összeköttetés bontásába
TIMED WAIT	Vár, amíg az összes csomag ki nem hal
CLOSING	Mindkét fél egyszerre próbálta bontani az összeköttetést
CLOSE WAIT	A másik fél bontást kezdeményezett
LAST ACK	Vár, amíg az összes csomag ki nem hal

6.27. ábra. *A TCP összeköttetéseket kezelő véges állapotú gép állapotai*

Minden összeköttetés *CLOSED* állapotból indul. Akkor hagyja el ezt az állapotot, amikor passzív módon (LISTEN) vagy aktív módon (CONNECT) összeköttetést próbál létesíteni. Ha a másik fél az ellenkező primitívet hajtja végre, az összeköttetés felépül és *ESTABLISHED* állapotot vesz fel. Bármelyik fél kezdeményezheti az összeköttetés bontását. Amikor ez lezajlott, az állapot ismét *CLOSED* lesz.

Magát a véges állapotú gépet a 6.28. ábrán láthatjuk. Vastag vonalak mutatják egy aktív kliens összeköttetés-létesítési folyamatát egy passzív szerverrel, a folytonos vo-



6.28. ábra. *A TCP összeköttetést kezelő véges állapotú gép. A vastag folytonos vonal a kliens normális működését, a vastag szaggatott vonal a szerver működését jelzi. A vékony vonalak váratlan események*

nalak a kliensre, a szaggatottak a szerverre vonatkoznak. A vékony vonalak váratlan eseményeket jeleznek. A 6.28. ábrán az összes nyílhoz tartozik egy *esemény/tevékenység* pár. Az esemény lehet egy felhasználó által végrehajtott rendszerhívás (CONNECT, LISTEN, SEND vagy CLOSE), egy szegmens érkezése (*SYN*, *FIN*, *ACK* vagy *RST*), vagy egy esetben a kétszeres maximális csomagélettartamra beállított időzítő lejárása. A tevékenység lehet vagy egy vezérlő szegmens (*SYN*, *FIN* vagy *RST*) elküldése, vagy semmi (ezt „–" jelöli). A megjegyzések zárójelben láthatók.

A diagramot úgy érthetjük meg legkönnyebben, ha először a kliens útját követjük végig (vastag folytonos vonal), majd a szerverét (vastag szaggatott vonal). Amikor a kliens gépen egy alkalmazás CONNECT kérést ad ki, a helyi TCP entitás létrehoz egy összeköttetés rekordot, amely *SYN SENT* állapotban van, és elküld egy *SYN* szegmenst. Jegyezzük meg, hogy több alkalmazás egyidejűleg több összeköttetést létesíthet, használhat, ezért az állapotok egyes összeköttetésekre vonatkoznak, és az összeköttetéshez tartozó rekordban vannak feljegyezve. Amikor a *SYN* + *ACK* megérkezik, a TCP elküldi a háromutas kézfogás utolsó *ACK* szegmensét és az összeköttetés *ESTABLISHED* állapotba kerül. Ezután lehet adatot küldeni és fogadni.

Amikor egy alkalmazás befejezte tevékenységét, végrehajtja a CLOSE primitívet, melynek hatására a helyi TCP entitás egy *FIN* szegmenst küld, majd az ehhez tartozó nyugtára (*ACK*) vár (szaggatott téglalap *aktív lebontás* megjegyzéssel). Amikor megérkezik az *ACK*, az új állapot *FIN WAIT 2* lesz, és az összeköttetés egyik irányban befejeződik. Amikor a másik fél is bont, egy *FIN* szegmens érkezik, amire a helyi entitás nyugtát küld. Ekkorra mindkét fél lebontotta az összeköttetést, de a TCP még maximális csomagélettartam ideig várakozik, így még akkor is garantálható, hogy az összeköttetés összes csomagja kihal, ha esetleg egy nyugta elveszett volna. Amikor az időzítő lejár, a TCP törli az összeköttetés bejegyzését.

Most vizsgáljuk meg az összeköttetés kezelését a szerver szemszögéből. A szerver LISTEN hívást ad ki és csöndben figyeli, hogy ki bukkan föl. Amikor beérkezik egy *SYN* szegmens, nyugtázza, és *SYN RCVD* állapotba lép. Amint a szerver *SYN* szegmensére is nyugta érkezik, a háromutas kézfogás protokoll véget ér és a szerver *ESTABLISHED* állapotba kerül. Megkezdődhet az adatátvitel.

Ha a kliens végzett tennivalóival, CLOSE hívást ad ki, melynek hatására *FIN* szegmens érkezik a szerverhez (szaggatott vonalas téglalap *passzív lebontás* felirattal). A szerver ezután megszakítást (signal) kezdeményez. Amikor a szerver is végrehajtja a CLOSE primitívet, a TCP entitás *FIN* szegmenst küld a kliensnek. Amint a kliens nyugtája megjelenik, a szerver bontja az összeköttetést és törli a hozzá tartozó bejegyzést.

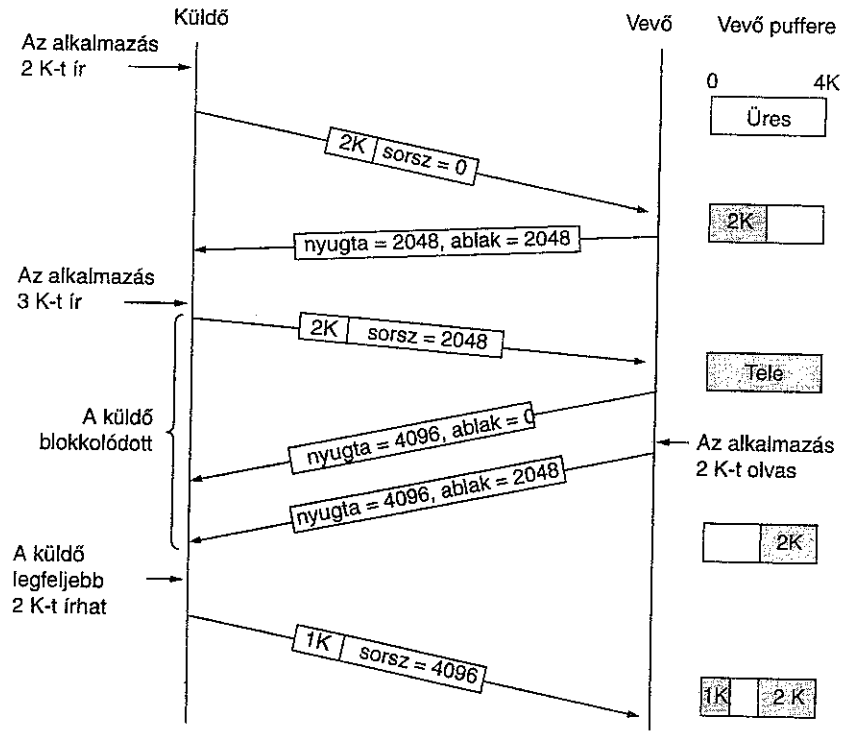
6.4.5. TCP átviteli politika

Az ablakkezelés a TCP-ben nem kötődik olyan szorosan a nyugtákhoz mint a legtöbb adatkapcsolati protokollban. Tegyük fel például, hogy a vevő 4096 bájtos pufferrel rendelkezik (6.29. ábra). Ha a küldő egy 2048 bájtos szegmenst küld el, ami rendben meg is érkezik, a vevő nyugtázza a vételt. Mivel azonban most csak 2048 bájt szabad pufferterülete van (amíg az alkalmazás ki nem olvas belőle valamennyi adatot) bejelenti, hogy a következő bájttól kezdve 2048 bájtos ablakot használ.

Most a küldő újabb 2048 bájtot továbbít, amit a vevő nyugtáz, továbbá közli, hogy az ablakméret 0 bájt. A küldőnek le kell állnia, amíg a fogadó hoszton futó alkalmazói folyamat el nem távolít valamennyi adatot a pufferből, amikor is a TCP egy nagyobb ablak használatát jelentheti be.

Amikor az ablakméret 0 bájt, a küldő normális esetben nem küldhet szegmenst, azonban van két kivétel. Először is a sürgős adatot továbbíthatja, hogy lehetővé tegye például a távoli gépen futó folyamat megszakítását. Másodszor, a küldő elküldhet egy egybájtos szegmenst, melyben arra kéri a vevőt, hogy közölje vele a következő várt bájt sorszámát és az ablakméretet. A TCP szabvány explicit módon biztosítja ezt a lehetőséget, hogy elkerülje a holtpontot, amennyiben egy ablakméretet közlő szegmens elveszne.

A küldő nem köteles azonnal továbbítani az alkalmazástól kapott adatot. A vevő sem köteles azonnal nyugtázni a beérkezett szegmenseket. Például a 6.29. ábrát tekintve, amikor beérkezett az első 2 KB adat, a TCP, tudva hogy 4 KB-os ablak áll rendelkezésére, teljesen korrekt módon járna el, ha az adatot újabb 2 KB beérkezéséig a pufferben tárolná, hogy 4 KB rakománnyal tudja továbbítani a szegmenst. Ezt a szabadságot a teljesítőképesség fokozásában lehet kamatoztatni.

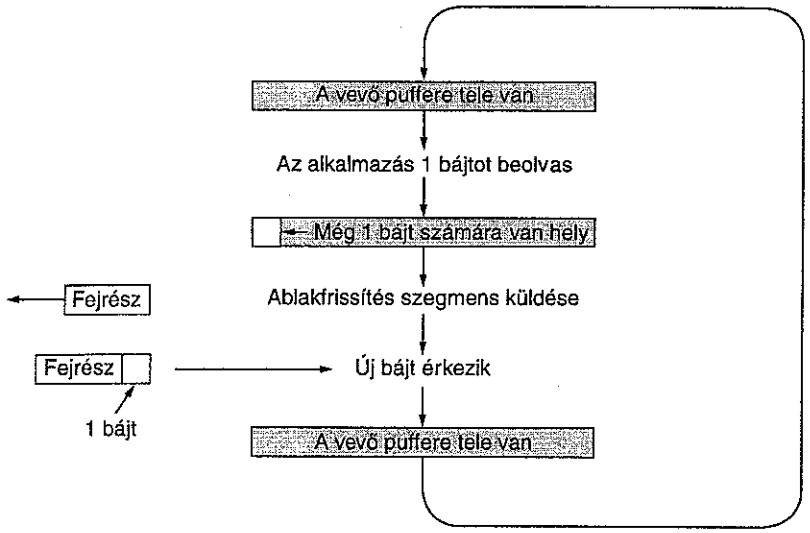


6.29. ábra. *A TCP ablakkezelése*

Vegyünk egy TELNET összeköttetést egy interaktív szövegszerkesztővel, ami minden billentyűleütésre reagál. Legrosszabb esetben amikor megérkezik egy karakter a küldő TCP entitáshoz, az létrehoz egy 21 bájtos szegmenst, amit átad az IP-nek, hogy 41 bájtos IP datagramként továbbítsa. A vevő oldali TCP azonnal visszaküld egy 40 bájtos nyugtát (20 bájtnyi TCP fejrész és újabb 20 bájtnyi IP fejrész). Később, mikor a szövegszerkesztő beolvasta a kapott bájtot, a TCP egy bájttal jobbra mozdítja az ablakot, és ezt közli a küldővel is. Ez a csomag szintén 40 bájtos. Végül, mikor a szövegszerkesztő feldolgozta a karaktert, egy 41 bájtos csomagban értesíti a feladót. Összesen négy szegmens továbbítása, azaz a sávszélesség 162 bájtja szükséges minden egyes begépelt karakterhez. Amikor a sávszélesség az igényeltnél kisebb, nem célszerű így gazdálkodni.

A fönti helyzet optimalizálására több TCP implementációban alkalmazzák a következő megközelítést. Késleltessük a nyugták és ablakméret információk elküldését 500 ms-ig azt remélve, hogy egy visszaküldendő adatcsomagba ágyazva ingyen elküldhetjük őket. Ha feltételezzük, hogy a szövegszerkesztő fél másodpercen belül küld visszajelzést, csak egyetlen 41 bájtos csomagot kell elküldeni a távoli felhasználónak. Az elküldött csomagok száma és a fölhasznált sávszélesség így a felére csökken.

Bár ez a szabály csökkenti a hálózat vevő által okozott terhelését, a küldő az egyetlen adatbájtot tartalmazó 41 bájtos csomagok küldözgetésével még mindig pazarlóan gazdálkodik. Az ennek csökkentésére kidolgozott módszer a **Nagle-féle algoritmus** (Nagle, 1984) néven ismert. Nagle ötlete egyszerű: amikor a küldőhöz bájtonként érkezik az adat, csak az elsőt továbbítja, a többit addig puffereli, amíg az elküldött bájt nyugtája meg nem érkezik. Ezután a pufferben tárolt összes karaktert egyetlen TCP szegmensben elküldi, és ismét újrakezdi a pufferelést, amíg az összes nyugta meg nem



6.30. ábra. *A buta ablak jelenség*

érkezett. Ha a felhasználó gyorsan gépel, és lassú a hálózat, minden szegmensben számos karakter utazhat jelentősen csökkentve a felhasznált sávszélességet. Az algoritmus ezenkívül lehetővé teszi egy újabb csomag küldését, ha fél ablakra való vagy a maximális szegmensméretet kiadó adat összegyűlt.

A Nagle-féle algoritmus széles körben elterjedt a TCP implementációkban, de némely esetben szerencsésebb kikapcsolni. Például amikor egy X-Window alkalmazás fut az Internet felett, az egérmozgásokat el kell küldeni a távoli számítógépnek. Ha összegyűjtenénk őket és löketszerűen továbbítanánk, az egérmutató ugrálva mozogna, ami bosszantaná a felhasználókat.

Egy másik probléma, ami le tudja rontani a TCP teljesítőképességét, a **buta ablak jelenség (silly window syndrome)** (Clark, 1982). Ez a probléma akkor merül föl, amikor a küldő TCP entitás nagy blokkokban kapja az adatokat, de a fogadó oldalon futó interaktív alkalmazás bájtonként olvassa be. A probléma könnyebb megértésére vegyük szemügyre a 6.30. ábrát. Kezdetben a vevő TCP puffere tele van, és a küldő ezzel tisztában van (tehát az ablakmérete 0). Ezután az interaktív alkalmazás beolvas egy bájtot a TCP adatfolyamról. Megörül ennek a fogadó TCP entitás, elküldi az új ablakméretet a küldőnek, mondván, hogy minden rendben, egy bájtot küldhet.

A küldő teljesíti ezt a kívánságot is, egy bájtot elküld. A puffer ismét tele lesz, így a vevő nyugtázza a bájt érkezését, de egyidejűleg közli, hogy az ablak mérete 0. Ez a viselkedés így mehet örökké.

Clark megoldása szerint nem szabad megengedni a vevőnek, hogy 1 bájt változásra ablakméret frissítést küldjön. Ehelyett várakoztatni kell addig, amíg elegendő hely szabaddá nem válik, és inkább azt kell a küldővel közölni. Pontosabban a fogadó nem küldhet addig ablakméret információt, amíg az összeköttetés létesítésekor bejelentett maximális szegmensméretet nem tudja kezelni, vagy félig ki nem ürült a puffer. A két korlát közül a kisebbet kell elérnie a szabad hely méretének.

Ezen kívül a küldő is segíthet azzal, hogy nem küld apró szegmenseket. Ehelyett, addig el kell halasztani a továbbítást, amíg elég hely össze nem gyűlt az ablakban, hogy egy teljes szegmenst elküldhessen, vagy legalább olyan hosszút, mint a vevő ablakméretének fele (amit az eddig beérkezett ablakméret információk alapján becsülhet meg).

A Nagle-féle algoritmus és Clark megoldása a buta ablak problémára kiegészítik egymást. Nagle olyan problémát próbált megoldani, melyben a küldő alkalmazás bájtonként adta át az adatokat a TCP-nek. A Clark által megoldott problémában a vevő alkalmazás kérte bájtonként az adatokat a TCP-től. Mindkét megoldás helyes és képesek együtt működni. Az a cél, hogy a küldő ne adjon apró szegmenseket, és a vevő se kérjen kicsiket.

A fogadó TCP entitás tovább növelheti a teljesítőképességet, ha csak nagyobb egységekben frissíti az ablakot. A küldő TCP entitáshoz hasonlóan a fogadónak is van lehetősége az adatok pufferelésére, így az alkalmazás egy READ hívását addig blokkolhatja, amíg egy nagyobb adatblokkot nem tud átadni. Ezzel csökken a TCP hívások száma, ezzel együtt a túlterhelés (overhead) is. Ez természetesen a válaszidőt is megnöveli, de az állomány átviteléhez hasonló nem interaktív alkalmazások esetében a hatékonyság ellensúlyozhatja az egyes kérések megnövekedett válaszidejét.

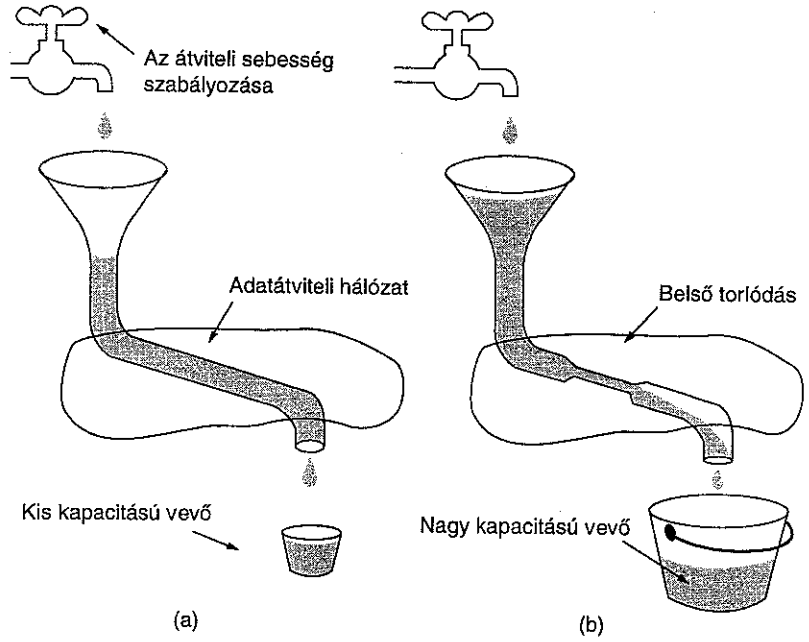
Egy másik feladat a vevő számára a rossz sorrendben érkező szegmensek kezelése.

A vevőtől függ, hogy azokat megtartja vagy eldobja. Nyugta természetesen csak akkor küldhető, ha a nyugtázott bájtig terjedő összes adat megérkezett. Ha a vevő a 0, 1, 2, 4, 5, 6 és 7 szegmenseket kapja meg, mindent nyugtázhat a 2. szegmens utolsó bájtjáig (azt is beleértve). Amikor a küldő időzítése lejár, újraküldi a 3. szegmenst. Ha a vevő megtartotta a 4–7. szegmenseket, a 3. szegmens vétele után a 7. szegmens utolsó bájtjáig mindet nyugtázhatja.

6.4.6. A TCP torlódásvédelme

Amikor egy hálózat terhelése nagyobb, mint amekkorát képes kezelni, torlódás keletkezik. Az Internet sem kivétel. Ebben a részben az utóbbi évtizedben kifejlesztett torlódáskezelő algoritmusokat fogunk tárgyalni. Bár a hálózati réteg szintén próbálkozik a torlódások kezelésével, a munka nehezét a TCP végzi, mert a torlódás elkerülésének igazi megoldása az adatátvitel sebességének csökkentése.

Elméletileg a torlódás egy fizikából kölcsönvett törvénnyel is kezelhető: ez a csomagmegmaradás törvénye. Az alapötlet az, hogy addig nem indítunk egy új csomagot a hálózatban, amíg egy régi el nem hagyja azt (tehát célba ér). A TCP az ablakméret dinamikus változtatásával próbálja ez a célt elérni.



6.31. ábra. *(a) Gyors hálózat táplál kis kapacitású vevőt. (b) Lassú hálózat táplál nagy kapacitású vevőt*

A torlódás kezelésének első lépése a torlódás detektálása. A régi időkben ez nehéz feladat volt. Ha egy időzítő lejárt egy elveszett csomag miatt, ennek oka egyaránt lehetett zaj az átviteli vonalon (1), vagy hogy egy túlterhelt router eldobta azt (2). Ezek között nehéz volt különbséget tenni.

Mostanában az átviteli hibákból eredő csomagvesztés viszonylag ritka, mert a legtöbb nagytávolságú trönk üvegszálas (bár a vezeték nélküli átvitel külön történet). Ennek következménye, hogy a legtöbb időtúllépés az Interneten torlódás eredménye. Az összes Interneten használt TCP algoritmus feltételezi, hogy időtúllépés torlódás miatt következik be, és hasonlóan figyeli az időzítőket a baj előjelei után kutatva, mint ahogy a bányászok nézik a kanárimadaraikat.

Mielőtt rátérnénk arra, hogy a TCP hogyan reagál a torlódásra, először beszéljünk arról, hogy mit is jelent elkerülni a torlódást. Amikor létrejön az összeköttetés, egy megfelelő ablakméretet kell választani. A vevő saját puffermérete alapján határozhatja meg az ablakméretet. Ha a küldő elfogadja ezt az ablakméretet, a vevőoldali puffer túlcsordulása nem okoz problémát, azonban a hálózatban fellépő torlódások így is bajt okozhatnak.

A 6.31. ábrán a probléma hidraulikai illusztrációját láthatjuk. A 6.31.(a) ábrán egy vastag cső torkollik kis kapacitású vevőbe. Ameddig a küldő nem ad több vizet, mint amennyi a vödörbe belefér, egy csöpp víz se megy kárba. A 6.31.(b) ábrán nem a vödör mérete jelenti a szűk keresztmetszetet, hanem a hálózat belső szállítókapacitása. Ha túl gyorsan érkezik sok víz, emelkedni fog a csőben a vízszint és a folyadék egy része kárba fog veszni (ebben az esetben a tölcsér szélén csordul túl).

Az Internet megoldása azon az észrevételen alapul, hogy két potenciális probléma létezik: – a hálózat kapacitása és a vevő kapacitása – melyeket külön kell kezelni. Ennek érdekében minden adó két ablakot használ: az egyik ablakot a vevő szabályozza, a másik pedig a **torlódási ablak (congestion window).** Mindkettő az adó által elküldhető bájtok számát mutatja. A ténylegesen továbbítható bájtok száma a két ablak értékének minimuma, tehát az effektív ablak a küldő és a fogadó által jónak tartott érték minimumát tartalmazza. Ha a vevő 8 kilobájtot kér, de az adó tisztában van vele, hogy 4 kilobájtnál hosszabb löketek eltömítik a hálózatot, 4 kilobájt adatot fog küldeni. Másrészt, ha a vevő 8 kilobájtot kér, emellett a küldő tudja, hogy 32 kilobájtos löketek is zavartalanul átvihetők, a teljes 8 kilobájtos blokkot el fogja küldeni.

Amikor egy összeköttetés létrejön, a küldő a torlódási ablak kezdőértékét az összeköttetésben használt legnagyobb szegmensméretre állítja be. Ezután elküld egy maximális szegmenst. Ha a szegmensre nyugta érkezik, mielőtt az időzítő lejárna, egy szegmensméretnyi bájttal növeli a torlódási ablak méretét, ami így a maximális szegmensméret kétszerese lesz, és két szegmenst küld el. Amint mindkettőre megérkezett a nyugta, a torlódási ablakot ismét maximális szegmensmérettel növeli. Ha a torlódási ablak n szegmens méretű, és az n számú nyugta rendben megérkezik, a torlódási ablakot n szegmens méretének megfelelő számú bájttal növeli. Végül is minden sikeresen nyugtázott adatlöket hatására a torlódási ablak megduplázódik.

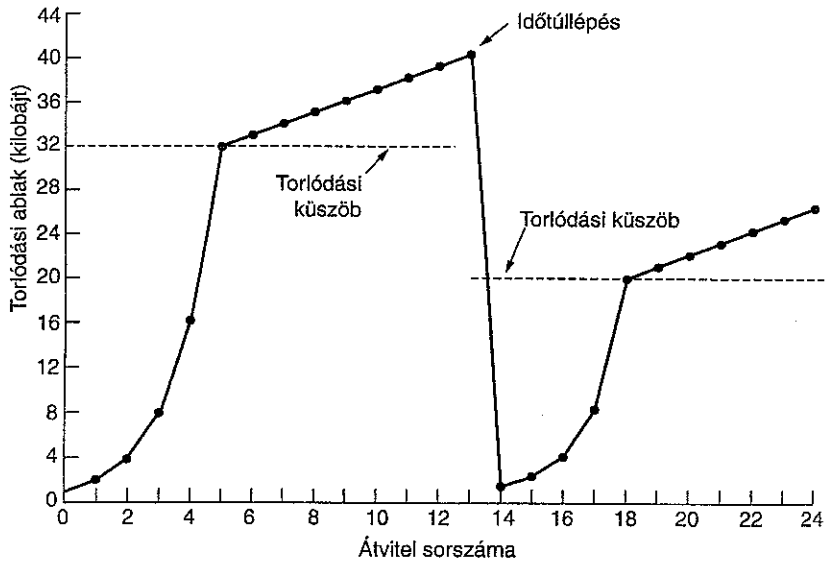
A torlódási ablak, amíg időtúllépés nem lép föl vagy el nem éri a vevő ablakméretét, exponenciálisan növekszik. Ha például 1024, 2048 és 4096 bájtos löketek könnyedén átvihetők, de egy 8192 bájtos löket továbbítása során időtúllépés következik be, a torlódási ablak méretét 4096 bájtra kell állítani, hogy a torlódást elkerülhessük. Amíg a torlódási ablak 4096 bájt hosszú marad, ennél hosszabb löketet nem továbbí-

tunk függetlenül attól, hogy a vevő mekkorát engedélyez. Ezt a technikát **lassú kezdést biztosító algoritmusnak** (Jacobson, 1988) nevezzük, amelyik azonban egyáltalán nem lassú. Az algoritmus exponenciális működésű. Minden TCP implementációnak támogatnia kell.

Vegyük most szemügyre az Internet torlódásvédelmi algoritmusát. Ez a vételi- és a torlódási ablakon kívül egy harmadik, **torlódási küszöbnek (threshold)** nevezett paramétert is használ, amelynek kezdőértéke 64 kilobájt. Amikor időtúllépés következik be, a torlódási küszöböt az aktuális torlódási ablak méretének felére állítjuk be, és a torlódási ablakot a maximális szegmensméretre állítjuk vissza. Ezek után meghatározzuk a hálózat teljesítőképességét a lassú kezdet protokoll segítségével, amelyben itt egy apró módosítás történt. Az exponenciális növekedés véget ér, amikor az ablakméret eléri a torlódási küszöböt. Innentől kezdve minden sikeres adatátvitel lineárisan növeli a torlódási ablakot (löketenként egy maximális szegmens méretével) ahelyett, hogy szegmensenként növelné eggyel. Végül is ez az algoritmus úgy tippeli, hogy valószínűleg jó közelítés lesz, ha megfelezi a torlódási ablakot, és onnan fokozatosan lépked felfelé.

A torlódásvédelmi algoritmus működésének illusztrálására tekintsük a 6.32. ábrát. A maximális szegmensméret itt 1024 bájt. Kezdetben a torlódási ablak 64 kilobájt volt, de időtúllépés történt, ezért a torlódási küszöböt 32 kilobájtra állítottuk, a 0. átvitelhez használt torlódási ablakot pedig 1 kilobájtra. Ez innentől kezdve exponenciálisan növekszik, amíg el nem éri a torlódási küszöböt (32 kilobájt). Onnantól kezdve lineárisan nő tovább.

A 13. átvitel nem szerencsés (ezt sejthettük volna), időtúllépés történik. A küszöböt az aktuális ablakméret felére állítjuk (mostanra 40 kilobájt lett, így a fele 20 kilobájt),



6.32. ábra. *Példa az Internet torlódásvédelmi algoritmusának működésére*

és a lassú kezdet protokollt újra elindítjuk. Amikor a 14. átviteltől kezdve sorra érkeznek a nyugták, az első négy mind megkétszerezi a torlódási ablakot, viszont azután a növekedés ismét lineáris lesz.

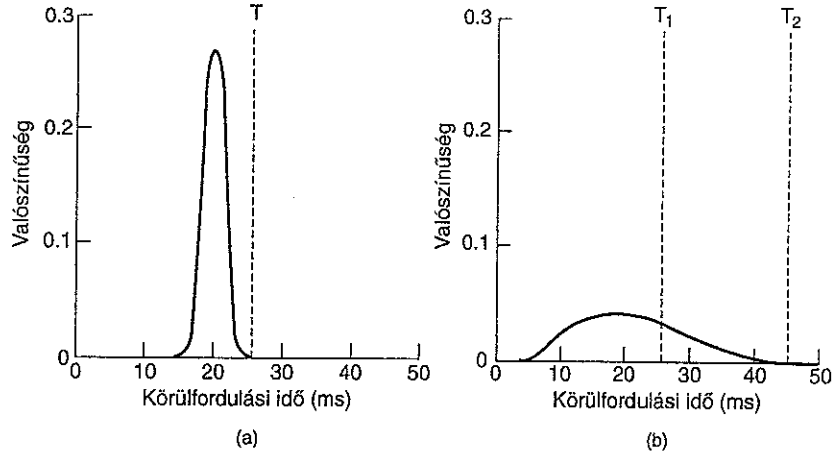
Ha nem történik több időtúllépés, a torlódási ablak addig növekszik, amíg el nem éri a vevő ablakméretét. Ezen a ponton abbahagyja a növekedést és állandó méretű marad amíg időtúllépés nem következik be vagy a vevő ablakmérete meg nem változik. Emellett azt az eseményt, amikor egy ICMP SOURCE QUENCH (lassítást kérő) csomag fut be, amelyet a TCP megkap, pontosan úgy kezeli, mintha időtúllépés történt volna.

A torlódásvédelmi mechanizmus fejlesztése tovább folytatódik. Például Brakmo és szerzőtársai (1994) jelentették be, hogy 40–70%-os javulást értek el a TCP áteresztőképességében az órák pontosabb kezelésével, a torlódások időtúllépés előtti jelzésével és a lassú kezdet protokoll ezen korai figyelmeztető mechanizmussal való bővítésével.

6.4.7. TCP időzítő kezelése

A TCP (elvileg) több időzítőt használ feladata elvégzéséhez. Ezek közül legfontosabb az **ismétlési időzítő (retransmission timer).** Egy szegmens elküldésekor az ismétlési időzítőt is elindítja. Ha a szegmensre az időzítő lejárása előtt nyugta érkezik, az időzítő leáll. Ha viszont az időzítő még a nyugta beérkezése előtt lejár, a szegmenst újraküldi a TCP (és az időzítőt is újraindítja). Fölmerül a kérdés: milyen hosszú ideig fusson az időzítő?

Ez a probléma sokkal bonyolultabb az Internet szállítási rétegében, mint a 3. fejezetben tárgyalt általános adatkapcsolati protokollok esetében. Az utóbbi esetben a várható késleltetés sokkal jobban megjósolható (azaz kicsi a szórásnégyzete), tehát az időzítőt úgy be lehet állítani, hogy kicsivel a nyugta beérkezésének várt ideje után járjon le. Ezt



6.33. ábra. *(a) A nyugtabeérkezési idők sűrűség függvénye az adatkapcsolati rétegben. (b) A nyugtabeérkezési idők sűrűségfüggvénye a TCP-ben*

láthatjuk a 6.33.(a) ábrán. Mivel az adatkapcsolati rétegben ritkán késnek a nyugták, a nyugta hiánya a szóban forgó időpontban általában a keret vagy nyugta elvesztését jelzi.

A TCP ettől gyökeresen eltérő környezettel szembesül. A TCP nyugták késleltetési idejének sűrűségfüggvénye sokkal inkább a 6.33.(b) ábrára hasonlít, mint a 6.33.(a) ábrára. A rendeltetési helyig terjedő körülfordulási időt nehéz megállapítani. Még ha ismert is, az időzítés időtartamáról is nehéz dönteni. Ha az időtartamot túl rövidre állítjuk be, mondjuk a 6.33.(b) ábra T_1 értékére, felesleges újraküldések történnek, az Internetet haszontalan csomagok terhelik. Ha túl hosszúra állítjuk (T_2), a teljesítőképesség egy csomag elveszésekor a hosszú újraküldési késleltetés miatt csökken. Ezen kívül a nyugta érkezési idejének átlaga és szórásnégyzete is jelentősen változhat pár másodperc alatt, ha torlódás lép fel vagy szűnik meg.

A megoldás az, ha erősen dinamikus algoritmust használunk, ami a hálózat teljesítőképességének folyamatos mérése alapján állandóan újra beállítja az időintervallumot. A TCP-ben általánosan használt algoritmus Jacobson (1988) nevéhez fűződik, és a következőképpen működik. A TCP minden összeköttetés részére fenntart egy *RTT*-nek nevezett változót, ami a szóban forgó rendeltetési helyig terjedő körülfordulási idő legjobb jelenlegi becsült értéke. Egy szegmens elküldésekor egy időzítőt is elindít a TCP, hogy megmérje, mennyi idő alatt ér vissza a nyugta, és ha túl sokáig késik, újraküldhesse a csomagot. Ha a nyugta az időzítő lejárása előtt visszaér, a TCP megméri, hogy mennyi ideig tartott (M). Ezután az

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

képlet szerint frissíti az *RTT* értékét, ahol α egy átlagoló tényező, azt határozza meg, hogy mekkora súlyt kapjon a régi érték. Tipikusan $\alpha = 7/8$.

Még *RTT* jó értékének tudatában sem triviális egy megfelelő ismétlési késleltetés kiválasztása. A TCP általában βRTT-t használ, viszont a trükk β megválasztásában van. Korai implementációkban β mindig 2 volt, de a tapasztalat azt mutatta, hogy a konstans érték rugalmatlanul viselkedik, nem tudja követni a változásokat, ha a szórásnégyzet megnőtt.

1988-ban javasolta Jacobson, hogy legyen β nagyjából a nyugtabeérkezési idő sűrűségfüggvényének szórásával arányos, tehát nagy szórásnégyzet nagy β-t eredményez és fordítva. Lényegében a *szórás átlagos szórással* történő olcsó közelítését javasolta. Algoritmusa egy másik csúszóátlagolással előállított változót is igényel, a D-vel jelölt szórást. Mindig, amikor egy nyugta beérkezik, a TCP kiszámolja a várt és megfigyelt értékek $|RTT - M|$ különbségét. Ennek egy csúszóátlagolással számított értékét tárolja D:

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

ahol α lehet ugyanaz vagy más, mint amit *RTT* számításához használtunk. Habár D nem egyezik pontosan a szórással, mégis elég jó. Ráadásul Jacobson eljárást adott csupán egész összeadás, kivonás, eltolás felhasználásával történő kiszámítására. A legtöbb TCP implementáció jelenleg ezt az algoritmust használja, az időzítés hosszát az alábbi összefüggés adja:

$$\text{Időzítés} = RTT + 4*D$$

A 4 szorzó választása valamennyire tetszőleges, de két jelentős előnnyel jár. Először is a néggyel történő szorzás megvalósítható egyetlen eltolással. Másodszor, lecsökkenti a fölösleges időtúllépések és újraküldések számát, mert a csomagok kevesebb mint egy százaléka érkezik a szórás négyszeresénél nagyobb késéssel. (Lényegében Jacobson először a 2 használatát javasolta, de későbbi tanulmányok szerint a 4 jobb teljesítőképességet eredményez.)

Az *RTT* dinamikus becslésekor felmerül egy probléma: mi a teendő, ha egy szegmens időzítése lejár, és újraküldik? Amikor beérkezik a nyugta, nem tudható, hogy az első átvitelre vonatkozik vagy az újabbra. Egy rossz tipp jelentősen megzavarhatja az *RTT* becslését. Phil Karn nehéz körülmények között fedezte fel ezt a problémát. Ő lelkes rádióamatőr, aki a TCP/IP csomagok amatőr rádióval történő átvitelével foglalkozik, ami egy hírhedten megbízhatatlan médium (egy jó napon a csomagok fele is átjuthat). Javaslata egyszerű: ne frissítsük az *RTT* értékét újraküldött szegmensek esetén, hanem az időzítés hosszát minden kudarc esetén duplázzuk meg, amíg a szegmens végül át nem jut. Ezt a javítást **Karn-féle algoritmusnak** nevezik. A legtöbb TCP implementációban alkalmazzák.

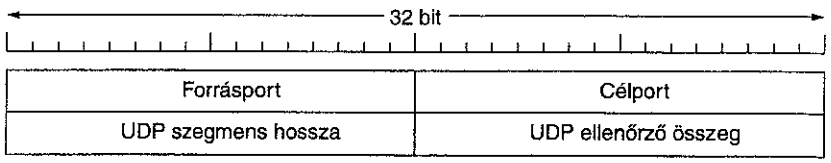
A TCP nem csak az ismétlési időzítőt használja. Egy másik időzítő a **folytatódó időzítő (persistence timer).** Ezt az alábbi holtpont elkerülésére tervezték. A vevő küld egy nyugtát 0 ablakmérettel, amivel a küldőt várakozásra kéri. Később a vevő frissíti az ablakot, de a frissítést hordozó csomag elvész. Most mind a küldő és a fogadó arra vár, hogy a másik tegyen valamit. Amikor a folytatódó időzítő lejár, a küldő egy kérést küld a vevőnek, amire válaszul megkapja az ablakméretet. Ha ez még mindig 0, a folytatódó időzítőt újraindítja és az egész folyamat megismétlődik, különben, ha nagyobb 0-nál, megkezdheti az adatátvitelt.

A harmadik időzítő, amelyet néhány implementáció használ, az **életben tartó időzítő (keepalive timer).** Amikor egy összeköttetés már régóta tétlen, az életben tartó időzítő lejár, és ennek hatására a TCP ellenőrzi, hogy partnere még mindig működik-e. Ha a távoli entitás nem válaszol, az összeköttetés befejeződik. Ez a szolgáltatás ellentmondásos, mert növeli a túlterhelést, és egy átmeneti hálózatszakadás hatására befejezhet egy amúgy még működő összeköttetést.

Az utolsó időzítő, amit minden TCP összeköttetésben alkalmaznak, az *TIMED WAIT* állapotban az összeköttetés bontásakor használt időzítő. Ez a maximális csomagélettartam kétszereséig jár, hogy biztosítsa az összeköttetés lebontása után az összeköttetés összes korábban generált csomagjának kihalását.

6.4.8. UDP

Az Internet protokollcsomag tartalmaz összeköttetés nélküli szállítási protokollt is. Ez az **UDP (User Datagram Protocol).** Az UDP beágyazott nyers IP datagramok küldését teszi lehetővé a felhasználók számára összeköttetés létesítése nélkül. Sok kliens-szerver alkalmazás, amely egyetlen kérdésre egyetlen választ küld, UDP-t használ



6.34. ábra. *Az UDP fejrész*

ahelyett, hogy összeköttetés létesítésével és bontásával bajlódna. Az UDP leírását az RFC 768-ban találhatjuk.

Az UDP szegmens 8 bájtos fejrészből és az azt követő adat mezőből áll. A fejrészt a 6.34. ábrán láthatjuk. A két port ugyan azt a célt szolgálja, mint a TCP-ben: a forrás- és a célgépen belül azonosítja a végpontokat. Az *UDP szegmens hossza* tartalmazza a fejrész és az adat együttes hosszát. Az *UDP ellenőrző összeg* magába foglalja a 6.25. ábrán látható pszeudofejrészt, az UDP fejrészt és az UDP adatot, szükség esetén páros hosszúságúra kiegészítve. Használata opcionális, helyette 0 áll, ha nincs kiszámolva (egy valódi számítással előálló 0 az 1-es komplemens ábrázolás miatt csupa 1-ként jelenik meg). Kikapcsolni értelmetlen, hacsak az adatok minősége nem számít (pl. digitalizált beszéd esetében).

6.4.9. Vezeték nélküli TCP és UDP

Elméletileg a szállítási protokolloknak függetlennek kéne lenniük az alattuk fekvő hálózati réteg technológiájától. Lényegében a TCP-nek nem kéne azzal törődnie, hogy az IP fényvezető szálon vagy rádión keresztül működik-e. A gyakorlatban viszont számít, mert a legtöbb TCP implementációt gondosan optimalizálták olyan feltételezéseket használva, amelyek érvényesek vezetékes hálózatokra, de vezeték nélküli hálózatokon kudarcot vallanak. A vezeték nélküli átvitel tulajdonságainak figyelmen kívül hagyása logikailag helyes TCP implementációt eredményezhet, viszont a teljesítőképessége szörnyen kicsi lesz.

A legfőbb problémát a torlódásvédelmi algoritmus jelenti. Manapság szinte minden TCP implementáció az feltételezi, hogy az időtúllépéseket torlódások okozzák, nem a csomagok elvesztése. Ebből következően, amikor az időzítő lejár, a TCP lassít és kisebb sebességgel ad (lásd Jacobson-féle lassú kezdet algoritmus). Ezen megközelítés mögött meghúzódó gondolat a hálózat terhelésének csökkentése, és így a torlódás csökkentése.

Sajnos a vezeték nélküli átviteli vonalak erősen megbízhatatlanok. Folyton csomagokat veszítenek. Az elveszített csomagok kezelésének helyes megközelítése az, hogy újraküldjük azokat, mégpedig olyan gyorsan, ahogy csak lehet. A lassítás viszont csak ront a helyzeten. Ha mondjuk az összes csomag 20%-a elvész, és az adó 100 csomag/másodperc sebességgel forgalmaz, az átbocsátóképesség 80 csomag/s. Ha a küldő lelassít 50 csomag/s sebességre, az átbocsátóképesség 40 csomag/másodperc értékre esik vissza.

Lényegében ha egy csomag a vezetékes hálózaton elveszik, a küldőnek lassítania kell. Ha a csomag vezeték nélküli hálózaton vész el, az adónak intenzívebben kellene adnia. Ha a küldő nem tudja, hogy milyen hálózattal van dolga, nehéz meghozni a helyes döntést.

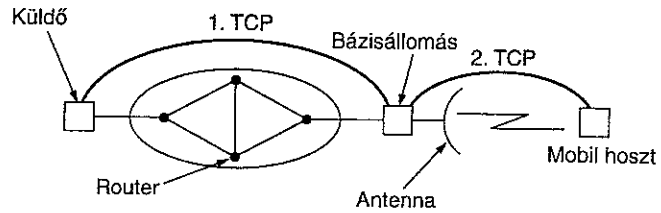
Gyakran az adó és a vevő közötti útvonal inhomogén. Lehet, hogy az első 1000 km vezetékes hálózat fölött fut, viszont az utolsó 1 km vezeték nélküli. Most még nehezebb időtúllépés esetén helyesen dönteni, mert számít, hogy hol lépett föl a probléma. Bakne és Badrinath javaslata (1995) a **közvetett TCP** használata. Osszuk a TCP összeköttetést két külön összeköttetésre, mint azt a 6.35. ábrán láthatjuk. Az első összeköttetés a küldőtől a bázisállomásig tart, a második a bázisállomástól a vevőig. A bázisállomás egyszerűen átmásolja a csomagokat az összeköttetések között mindkét irányba.

Ez a módszer azzal az előnnyel jár, hogy így mindkét összeköttetés homogén. Az első összeköttetésen bekövetkező időtúllépések lelassítják a küldőt, ugyanakkor a második összeköttetésen fellépők felgyorsítják. Más paramétereket is külön lehet beállítani a két összeköttetésen. A módszer hátránya, hogy felrúgja a TCP szemantikát. Mivel az összeköttetés mindkét része teljes TCP összeköttetés, a bázisállomás a szokásos módon nyugtáz minden TCP szegmenst. Ezért viszont az, ha a küldőhöz beérkezik egy nyugta, még nem jelenti azt, hogy a vevő megkapta a szegmenst, csak annyit jelent, hogy a bázisállomáshoz eljutott.

Egy másfajta megoldás, ami Balakrishnan és munkatársai (1995) nevéhez fűződik, nem töri meg a TCP szemantikát. Működésének alapja több kisebb változtatás a bázisállomás hálózati rétegének kódjában. A módosítások közül az egyik egy fürkésző ügynök beépítése, amely figyeli és gyűjti a mobil állomás felé tartó TCP szegmenseket, és a visszaérkező nyugtákat. Amikor a fürkésző ügynök észrevesz egy mobil állomás felé tartó TCP szegmenst, viszont a (viszonylag rövid) időzítése alatt nem érkezik onnan nyugta, egyszerűen újraküldi a szegmenst anélkül, hogy ezt közölné a forrással. Szintén újraküld, ha a mobil állomásról kettőzött nyugták érkeznek, ami egyértelműen annak a jele, hogy a mobil hoszt valamit összekevert. A kettőzött nyugtákat eldobja, nehogy a forrás torlódás jeleként félreértelmezze őket.

Ennek az átlátszóságnak az a hátránya, hogy amennyiben a vezeték nélküli vonal nagyon veszteséges, a forrás időzítése lejárhat miközben nyugtára várakozik, és elindíthatja a torlódásvédelmi algoritmust. A közvetett TCP esetében a torlódásvédelmi algoritmus soha nem indul el, kivéve ha ténylegesen torlódás van a hálózat vezetékes részében.

Balakrishnan publikációja a mobil hosztnál elveszett szegmensek problémájára is



6.35. ábra. *Egy TCP összeköttetés két külön összeköttetésre bontása*

tartalmaz megoldást. Amikor a bázisállomás szünetet fedez fel a beérkező sorszámokban, egy TCP opció felhasználásával szelektív ismétlést kér a hiányzó bájtokra. Ezzel a két módosítással a vezeték nélküli vonal mindkét irányban megbízhatóbbá tehető anélkül, hogy a forrás tudna róla, vagy a TCP szemantikája megváltozna.

Bár az UDP-nek nincsenek olyan problémái, mint a TCP-nek, a vezeték nélküli kommunikáció ott is támaszt nehézségeket. A fő probléma az, hogy az UDP-t használó programok nagyfokú megbízhatóságra számítanak. Tudják, hogy nem kapnak semmilyen garanciát, de így is szinte teljesen tökéletes szolgálatot várnak. A vezeték nélküli környezet messze lesz a tökéletességtől. Azoknak a programoknak, amelyek csak elfogadható költséggel képesek kezelni az elveszett UDP üzeneteket, a teljesítőképessége katasztrofális lesz, ha hirtelen egy olyan környezetből, ahol az üzenetek nagyon ritkán tűnnek el, olyanba kerülnek, ahol az üzenetek minduntalan elvesznek.

A vezeték nélküli kommunikáció más területre is hatással van, nem csak a teljesítőképességre. Például hogyan találhat meg egy mobil állomás egy helyi nyomtatót, amihez csatlakozhat, ahelyett hogy a saját otthoni nyomtatóját használná? Valamennyire ehhez kapcsolódik, hogy hogyan lehet megszerezni a lokális cella WWW oldalát még akkor is, ha ismeretlen a neve. Ezen kívül a WWW oldalak tervezői egyre inkább feltételezik, hogy nagy sávszélesség áll rendelkezésre. Egy minden oldalon elhelyezett nagy logo hátrányossá válik és bosszantja a felhasználót, ha minden alkalommal, amikor egy oldalra hivatkozik, 30 másodpercig tart az átvitel egy 9600 b/s sebességű modemen.

6.5. Az ATM AAL rétegének protokolljai

Nem teljesen tiszta, hogy az ATM rendelkezik-e szállítási réteggel, vagy sem. Egyrészt az ATM rétegben megvannak a hálózati réteg funkciói, és tetején van egy újabb réteg (AAL), ami az AAL-t egyfajta szállítási réteggé teszi. Néhány szakértő osztja ezt a nézetet (pl. De Prycker, 1993). Az itt használt egyik protokoll (AAL 5) funkcionálisan hasonló az UDP-hez, ami vitathatatlanul szállítási protokoll.

Másrészt egyetlen AAL protokoll sem biztosít egy megbízható végtől végig terjedő összeköttetést, mint azt a TCP teszi (bár néhány nagyon apró változtatással képesek lennének erre). Emellett a legtöbb alkalmazásban az AAL tetejére egy másik szállítási réteg épül. További szőrszálhasogatás helyett ebben az alfejezetben az AAL rétegről és annak protokolljairól fogunk beszélni anélkül, hogy valódi szállítási rétegnek tekintenénk.

Az ATM hálózatok AAL rétege gyökeresen eltér a TCP-től, főleg azért, mert a tervezőket elsősorban hang- és videofolyamok továbbítása érdekelte. Ott a gyors továbbítás sokkal fontosabb a helyes átvitelnél. Emlékezzünk vissza, hogy az ATM réteg csak 53 bájtos cellákat bocsát ki egymás után. Nincs se hibakezelés, se forgalomszabályozás vagy egyéb vezérlés. Ebből következik, hogy nem igazán illeszkedik az alkalmazások többségének igényeihez.

Ennek a szakadéknak áthidalására az ITU az I.363 ajánlásában definiált egy végtől végig terjedő réteget az ATM réteg tetején. Ez az úgynevezett **AAL (ATM Adaptation Layer – ATM adaptációs réteg)** réteg nehéz körülmények között született meg, történetében sok hiba, újratervezés és befejezetlen munka szerepel. Az alábbi részekben ezt a réteget és tervezését fogjuk megvizsgálni.

Az AAL célja hasznos szolgálatokat nyújtani az alkalmazói programok számára és eltakarni előlük az adat cellákká darabolását a forrásnál és visszaalakítását a rendeltetési helyen. Amikor az ITU definiálni kezdte az AAL-t, ráébredt, hogy a különböző alkalmazások igényei is különbözőek, ezért a szolgálatokat három független tulajdonsággal jellemezte:

1. Valós idejű szolgálat, vagy nem valós idejű szolgálat.

2. Állandó bitsebességű szolgálat vagy változó bitsebességű szolgálat.

3. Összeköttetés alapú vagy összeköttetés nélküli szolgálat.

Elméletileg a három tulajdonság összesen nyolc különböző szolgálat definiálását teszi lehetővé, mint azt a 6.36. ábrán láthatjuk. Az ITU úgy gondolta, hogy ezek közül négynek lenne gyakorlati haszna, és A, B, C, illetve D osztálynak nevezte őket, a többire nincs támogatás. Az ATM 4.0 változatától kezdve a 6.36. ábra valamennyire fölöslegessé vált, ezért itt inkább háttér-információként mutatjuk be, hogy könnyebben megérthessük, miért úgy tervezték az AAL protokollokat, ahogy tervezték. A szolgálati osztályok helyett ma inkább az 5. fejezetben tanulmányozott forgalmi osztályok (ABR, CBR, NRT-VBR, RT-VBR és UBR) között teszünk különbséget.

Ezen négy szolgálati osztály kezeléséhez az ITU négy protokollt definiált, ezek rendre AAL 1 – AAL 4. Később azonban felfedezték, hogy a C és D osztályok technikai követelményei annyira hasonlóak, hogy jónak látták az AAL 3-at és AAL 4-et az AAL 3/4-ben egyesíteni. Ezután a számítógépipar, ami átaludta a váltást, ráébredt, hogy egyáltalán nem jó egyik sem. Úgy oldotta meg a problémát, hogy gyorsan összedobott még egy protokollt, ez lett az AAL 5. Mind a négyet röviden át fogjuk tekinteni. Ugyancsak megvizsgálunk egy, az ATM rendszerekben használt érdekes vezérlő protokollt.

	A		B	C			D
Időzítés	Valós idejű	Nem valós idejű	Valós idejű	Nem valós idejű	Valós idejű	Nem valós idejű	Valós idejű	Nem valós idejű
Bitsebesség	Állandó		Változó		Állandó		Változó	
Mód	Összeköttetés alapú				Összeköttetés nélküli			

6.36. ábra. *Az AAL által támogatott eredeti szolgálati osztályok (ma már fölösleges)*

6.5.1. Az ATM adaptációs réteg felépítése

Az AAL réteg két fő részre osztható, melyek közül az egyik újabb részekből épül fel. Ezt láthatjuk a 6.37. ábrán.

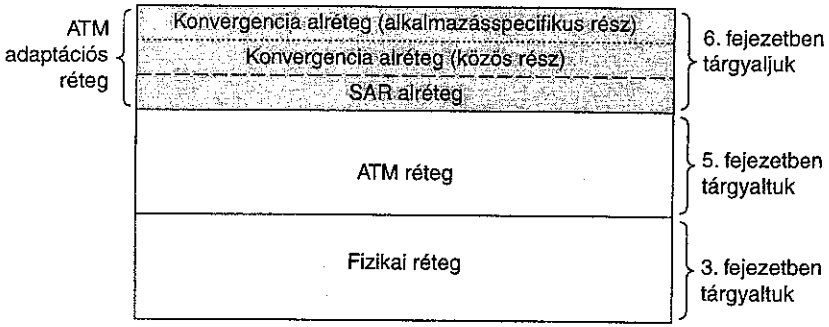
Az AAL réteg fölső részét **konvergencia alrétegnek (convergence sublayer)** nevezik. Feladata interfészt nyújtani az alkalmazás felé. Egy (adott AAL protokoll esetén) minden alkalmazásnál közös részből és egy alkalmazásspecifikus részből tevődik össze. Mindkét rész funkciói protokollfüggők, de tartalmazhatnak üzenetkeretezést és hibajelzést.

Ezen felül a forrásnál a konvergencia alréteg felel az alkalmazás felől jövő bitfolyamok vagy tetszőleges hosszúságú üzenetek fogadásáért, és azok továbbításhoz szükséges 44–48 bájtos egységekké tördeléséért. A rendeltetési helyen ez az alréteg a cellákból előállítja az eredeti üzeneteket. Az üzenethatárokat – ha voltak – megőrzi. Másképpen szólva, ha a forrás négy 512 bájtos üzenetet küld, ezek négy 512 bájtos üzenetként érkeznek meg, nem egy 2048 bájtos darabban. Adatfolyamok esetén nincsenek üzenethatárok, így azokat nem őrzi meg.

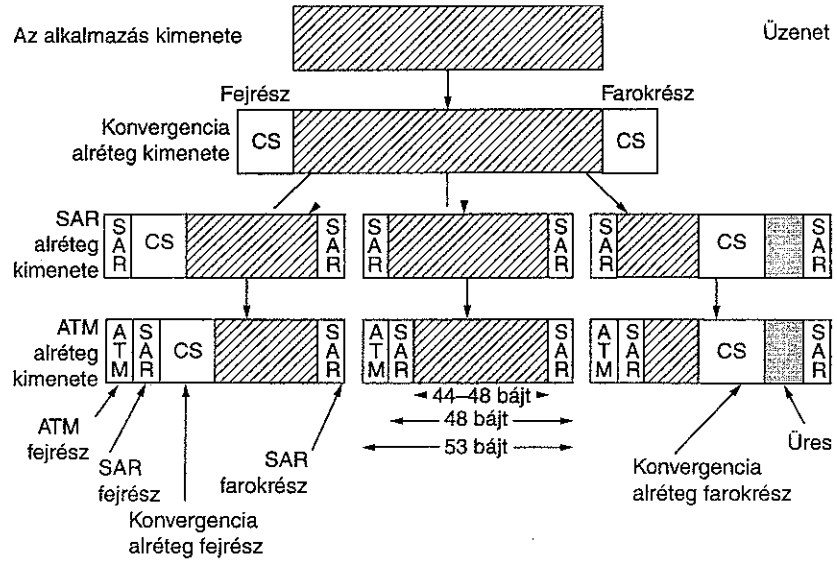
Az AAL alsó része a **SAR (Segmentation And Reassembly)** alréteg. Ez további fejrészeket és farokrészeket adhat az adategységekhez, melyeket a konvergencia alrétegtől kap, hogy adat mezővé alakítsa őket. Ezek a blokkok az ATM réteghez kerülnek továbbításra. A rendeltetési helyükön a SAR alréteg a cellákat visszaalakítja üzenetekké. A SAR alréteg alapvetően cellákkal foglalkozik, míg a konvergencia alréteg üzenetekkel.

A 6.38. ábrán láthatjuk a konvergencia- és SAR alrétegek általános működését. Amikor az AAL-hez egy üzenet érkezik az alkalmazástól, a konvergencia alréteg ahhoz fejrészt és/vagy farokrészt fűzhet. Ezt az üzenetet aztán 44–48 bájtos egységekre tördeli, melyeket átad a SAR alrétegnek. Ez minden darabhoz hozzáfűzheti saját fej-, illetve farokrészét, majd ezeket átadja az ATM rétegnek, hogy az független cellákként továbbítsa őket. Vegyük észre, hogy az ábra a legáltalánosabb esetet mutatja be, hiszen némelyik AAL protokollnak nincs fejrésze és/vagy farokrésze.

A SAR alréteg szintén rendelkezik egypár további funkcióval néhány (de nem minden) szolgálati osztály esetében. Például olykor hibajelzést és nyalábolást végez. A



6.37. ábra. *Az AAL réteg és alrétegei az ATM modellben*



6.38. ábra. *Egy ATM hálózatban az üzenethez adható fej- és farokrészek*

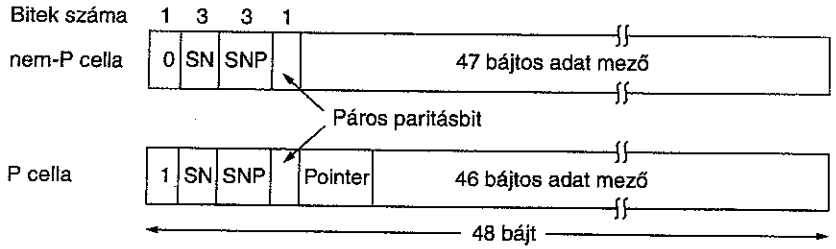
SAR alréteg minden szolgálati osztályban jelen van, viszont az adott protokolltól függően több vagy kevesebb munkát végez.

Az alkalmazás és az AAL réteg közti kommunikáció az első fejezetben tárgyalt szabványos OSI *kérés* és *bejelentés* primitívekkel folyik. Az alrétegek közti kommunikáció más primitívekkel történik.

6.5.2. AAL 1

Az AAL 1 protokoll szolgál A osztályú forgalom továbbítására, ami valós idejű, állandó bitsebességű összeköttetés alapú forgalom, mint például a tömörítetlen hang és video. A biteket állandó sebességgel adja az alkalmazás, és a távoli végen is ugyanezzel az állandó sebességgel és minimális késleltetéssel, dzsitterrel, overheaddel kell kézbesíteni. A bemenet egy üzenethatárok nélküli bitfolyam. Ehhez a forgalomhoz nem alkalmaznak megáll-és-vár vagy hozzá hasonló hibajelzési protokollt, mert az időzítések és újraküldések által okozott késleltetés megengedhetetlen. Az elveszett cellákat azonban jelenti az alkalmazásnak, ami aztán a maga módján intézkedhet (ha akar).

Az AAL 1 konvergencia alréteget és SAR alréteget használ. A konvergencia alréteg detektálja az elveszett és téves cellákat. (A téves cella a virtuális út vagy -áramkör azonosítójának észrevétlen hibájából eredően rossz rendeltetési helyre továbbított cella.) Ezenkívül egyenletesebbé teszi a bejövő forgalmat, hogy biztosítsa a cellák állandó sebességgel történő kézbesítését. Végül a konvergencia alréteg a bejövő üzeneteket vagy folyamot 46 vagy 47 bájtos egységekre tördeli, melyeket továbbítás cél-



6.39. ábra. *Az AAL 1 cella felépítése*

jából átad a SAR-nak. A túlsó oldalon ezekből helyreállítja az eredeti bemenetet. Az AAL 1 alrétegnek nincs saját protokoll fejrésze.

Ezzel ellentétben az AAL 1 SAR alréteg rendelkezik saját protokollal. Cellaformátumait a 6.39. ábrán láthatjuk. Mindkét cellafajta egy 1 bájtos fejrésszel kezdődik, ami tartalmaz egy 3 bites *SN* cellasorszámot (hogy a hiányzó vagy téves cellákat detektálhassa). Ezt a mezőt a 3 bites *SNP* cellasorszám-védelem (azaz ellenőrző összeg) követi. Ez az *SN* mezőre 1 bithiba javítását és 2 detektálását biztosítja. Az $x^3 + x + 1$ polinommal megvalósított CRC-t (Cyclic Redundancy Check) használ. Egy, a fejrész bájtra képzett páros paritásbit tovább csökkenti egy hibás sorszám észrevétlen becsúszásának valószínűségét. Az AAL 1 cellákat nem kell teljesen kitölteni 47 bájtnyi adattal. Például, ha 125 μs-ként 1 bájttal érkező digitalizált beszéddel töltenénk tele a cellákat, akkor 5,875 ezredmásodpercig kéne gyűjteni a mintákat. Ha átvitel előtt megengedhetetlen ekkora késleltetés, részben megtöltött cellákat lehet elküldeni. Ebben az esetben a cella valódi adatbájtjainak száma minden cellára ugyanaz az előre eldöntött érték.

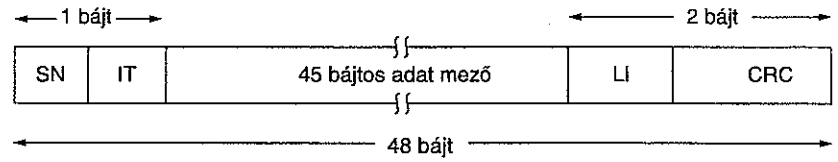
A *P* cellákat használják, ha az üzenethatárokat meg kell őrizni. A *Pointer* mező adja meg a következő üzenet kezdetének eltolását. Csak páros sorszámú cellák lehetnek *P* cellák, ezért a mutató 0–92 tartományból veheti értékét, és a saját, vagy az azt követő cella adat mezejébe mutathat. Vegyük észre, hogy ez a módszer tetszőleges számú bájtból álló üzenetet megenged, az üzenetek folytonosan jöhetnek, nem kell cellahatárokhoz igazodniuk.

A *Pointer* mező legnagyobb helyi értékű bitje le van foglalva későbbi felhasználásra. Az összes páratlan sorszámú cella első fejrészbitje óraszinkronizációra használt adatfolyamot alkot.

6.5.3. AAL 2

Az AAL 1-et egyszerű összeköttetés alapú valós idejű adatfolyamok hibajelzés nélküli (kivéve a hiányzó vagy téves cellákat, amiket jelez) átvitelére tervezték. Nyers tömörítetlen hang- vagy videoforrásra, vagy más adatfolyamra, ahol nem gond, ha néhány bit elkeveredik, az AAL 1 elegendő.

Tömörített hang- vagy videoforrás sebessége időben erősen változó lehet. Például sok tömörítési algoritmus periodikusan elküld egy teljes képkockát, majd csak azt a



6.40. ábra. *Az AAL 2 cella felépítése*

különbséget, amely az ezután következő keretek és az utoljára átvitt teljes keret között van. Amikor a kamera áll, és semmi sem mozog, a képkockák eltérései kicsik, viszont amikor a kamera gyors mozgást követ, az eltérések nagyok lesznek. Emellett az üzenethatárokat is meg kell őrizni, hogy a következő teljes képkocka kezdete fölismerhető legyen még elveszett cellák vagy rossz adat esetén is. Ezen feltételek teljesítéséhez trükkösebb protokoll szükséges. Az AAL 2-t ilyen célra tervezték.

Az AAL 1-hez hasonlóan a konvergencia alrétegnek itt sincs saját protokollja, viszont a SAR-nak van. A 6.40. ábrán láthatjuk a SAR cella felépítését. Egy 1 bájtos fejrésszel és egy 2 bájtos farokrésszel rendelkezik, így cellánként legfeljebb 45 adatbájtnak van helye.

Az *SN* (sorszám) mező a cellák számozására szolgál, hogy a hiányzó vagy téves cellákat detektálni lehessen. Az *IT* (információ típusa) jelzi, hogy a cella üzenet kezdetét, közepét vagy végét tartalmazza-e. Az *LI* (hossz) mező tartalmazza az adat mező bájtban mért hosszát (kisebb is lehet 45 bájtnál). Végül a *CRC* mező az egész cellára számított ellenőrző összeg, így a hibák detektálhatók.

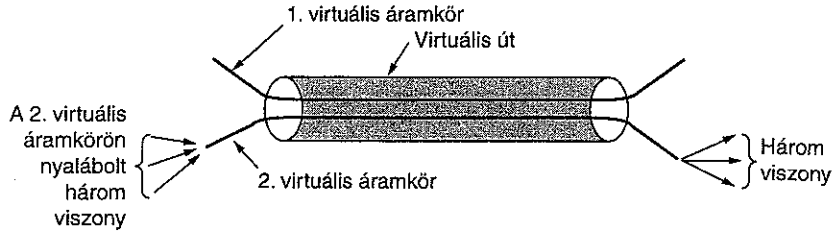
Furcsának tűnhet, hogy a mezők méreteit nem tartalmazza a szabvány. Egy bennfentes szerint a szabványosítási folyamat legvégén a bizottság ráébredt, hogy annyi problémája van az AAL 2-nek, hogy nem is érdemes használni. Sajnos már túl késő volt a szabványosítás leállításához, be kellett tartaniuk egy határidőt. Egy utolsó erőfeszítéssel a bizottság eltávolította a mezőhosszokat, tehát a hivatalos szabványt időben ki tudták adni, de így valójában senki nem tudja használni. Ilyen az élet a szabványosítás világában.

6.5.4. AAL 3/4

Eredetileg az ITU különböző protokollokat definiált a C és D osztályokra, tehát az összeköttetés alapú, illetve összeköttetés nélküli, cellavesztésre és hibára érzékeny, de nem időfüggő adatátvitelre. Ezután az ITU felfedezte, hogy nincs igazán szükség két protokollra, ezért egyetlen protokollban egyesítette őket, ez lett az AAL 3/4.

Az AAL 3/4 két üzemmódban dolgozhat: folyam vagy üzenet módban. Üzenet módban minden alkalmazásból kiadott AAL 3/4 hívás egy üzenetet küld el a hálózatban. Az üzenetet eredeti formájában kézbesíti, tehát az üzenethatárok megmaradnak. Folyam üzemmódban a határokat nem őrzi meg. Az alábbiakban az üzenet módra fogunk összpontosítani. Mindkét üzemmódban rendelkezésre áll megbízható és nem megbízható (ahol nincs garancia) átvitel.

A többi protokollok közül egyikben sincs meg az AAL 3/4 nyalábolási lehetősége.



6.41. ábra. *Több viszony nyalábolása egy virtuális áramkörre*

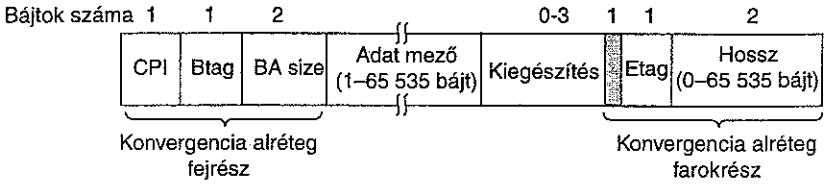
Az AAL 3/4 ezen tulajdonsága lehetővé teszi egy hoszt több viszonyának (pl. terminálbejelentkezések) közös virtuális áramkörön történő átvitelét. Ezeket a rendeltetési helyen megfelelően szétválasztja. Ezt láthatjuk a 6.41. ábrán.

Azért célszerű ezt a lehetőséget megvalósítani, mert a szolgáltatók gyakran minden összeköttetés felépítéséért és az összeköttetés nyitvatartási idejéért is díjat számítanak föl. Ha két hosztnak egyidejűleg sok viszonya él, drágább lenne mindegyikhez külön virtuális áramkört rendelni, mint az összeset egyetlen virtuális áramkörön nyalábolni. Ha egy virtuális áramkör elegendő sávszélességgel rendelkezik a feladat elvégzéséhez, nincs szükség egynél több áramkörre. Az összes, egy virtuális áramkört használó viszony ugyanazt a szolgálatminőséget kapja, hiszen azt virtuális áramkörönként határozzák meg.

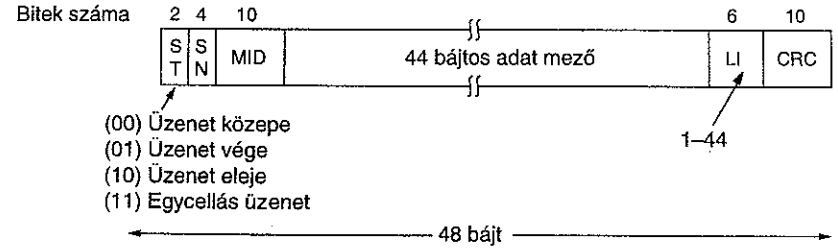
Ez a tulajdonság volt az igazi oka annak, hogy eredetileg külön AAL 3 és AAL 4 formátumok voltak: Amerikában akartak nyalábolást, Európában nem. Ezért mindkét csoport visszavonult, és elkészítette a saját szabványát. Végül az európaiak úgy döntöttek, hogy a fejrészben 10 bit megtakarítás nem ér annyit, hogy ne tudjon az Egyesült Államok és Európa kommunikálni. Ugyanennyi pénzért körömszakadtukig ragaszkodhattak volna elveikhez, és most három helyett négy inkompatíbilis AAL szabványunk lenne (melyek közül egy használhatatlan).

Az AAL 1- és AAL 2-től eltérően az AAL 3/4 rendelkezik mind konvergencia alréteg, mind SAR alréteg protokollal. Az alkalmazástól legfeljebb 65 535 bájtos üzenetek érkeznek a konvergencia alréteghez. Ezeket először négy bájt többszörösére egészíti ki, majd fej- és farokrésszel látja el. Ezt láthatjuk a 6.42. ábrán.

A *CPI* (Common Part Indicator) tartalmazza az üzenet típusát és az átszámolási egységet a *BA size,* illetve *Length* mezők számára. A *Btag* és *Etag* mezők szerepe az üzenetek keretezése. A két bájtnak egyformának kell lenni, értéküket minden üzenet elküldésekor eggyel növelni kell. Ezzel a mechanizmussal ismerhetők fel az elveszett



6.42. ábra. *AAL 3/4 konvergencia alrétegének üzenet formátuma*



6.43. ábra. *Az AAL 3/4 cella felépítése*

vagy téves cellák. A *BA size* mezőt pufferfoglalásra használják. Tájékoztatja a vevőt arról, hogy mekkora pufferterületet foglaljon le előre az üzenet számára. A *Length* mező tartalmazza az adat mező hosszát. Üzenet módban egyeznie kell a *BA size* értékével, folyam üzemmódban eltérő is lehet. A farokrész tartalmaz még egy használaton kívüli bájtot.

Miután a konvergencia alréteg előállította és az üzenethez fűzte a fej- és farokrészt (6.42. ábra), átadja az üzenetet a SAR alrétegnek, ami 44 bájtos szeletekre darabolja. Jegyezzük meg, hogy a nyalábolás megvalósításához a konvergencia alréteg egyszerre több különböző üzenetet előállíthat és tárolhat, és tetszőleges sorrendben először az egyikből, majd a másikból adhat át 44 bájtos darabokat a SAR alrétegnek.

A SAR alréteg minden 44 bájtos szeletet egy, a 6.43. ábrán látható felépítésű cella adat mezejében helyezi el. Ezeket a cellákat továbbítja a rendeltetési helyükre, ahol újra összeállítja, majd az ellenőrző összeggel ellenőrzi, és ha szükséges, elvégzi a megfelelő tevékenységet.

Az AAL 3/4 cella a következő mezőkből áll: az *ST* (szegmenstípus) mező az üzenetek keretezésére szolgál. Jelzi, hogy a cella egy üzenet kezdetét, közepét vagy a végét hordozza, avagy kicsi (azaz egycellás) üzenetet tartalmaz. Ezt követi egy négybites sorszám, *SN*, ami hiányzó vagy téves cellák detektálására szolgál. A *MID* (nyalábolásazonosító) tartja számon, hogy melyik cella melyik viszonyhoz tartozik. Emlékezzünk vissza, hogy a konvergencia alrétegnek különböző üzenetei lehetnek, melyeket egyszerre pufferel, viszont különböző viszonyokhoz tartozhatnak, és olyan sorrendben küldi ezek darabjait, ahogy akarja. Az *i.* viszonyhoz tartozó üzenetek minden darabjának *MID* értéke *i*, tehát az üzenetek helyesen összerakhatók a rendeltetési helyükön. A farokrész tartalmazza az adat mező hosszát és az ellenőrző összeget.

Vegyük észre, hogy az AAL 3/4 két szinten hoz be protokoll overheadet: minden üzenethez 8 bájtot ad és minden cellához további négyet. Mindent egybevetve ez egy nehézkes mechanizmus, különösen kis üzeneteket tekintve.

6.5.5. AAL 5

Az AAL 1–AAL 3/4 protokollokat főleg a távközlési ipar résztvevői tervezték és az ITU szabványosította a számítógépipar jelentősebb beleszólása nélkül. Amikor a számítógépipar végre fölébredt, és kezdett ráébredni a 6.43. ábra következményeire, pá-

nikhangulat alakult ki. A kétszintű protokoll bonyolultsága és csekély hatékonysága a meglepően rövid (csak 10 bites) ellenőrző összeggel párosulva néhány kutatót új protokoll kitalálására ösztönzött. Ezt **SEAL-nek (Simple Efficient Adaptation Layer – egyszerű hatékony adaptációs réteg)** nevezték utalva arra, hogy mit gondoltak a tervezők a régi protokollokról. Rövid tárgyalás után az ATM Forum elfogadta a SEAL-t, és AAL 5-nek keresztelte el. Az AAL 5-ről és arról, hogy miben különbözik az AAL 3/4-től, további információt Suzuki művében (1994) olvashatunk.

Az AAL 5 többfajta szolgálatot kínál az alkalmazásoknak. Az egyik lehetőség a megbízható szolgálat (tehát garantált kézbesítés és forgalomszabályozás az ütközések elkerülése érdekében). Egy másik választás a megbízhatatlan szolgálat lehet (nincs garancia a célba érésre), két lehetőséggel: a hibás ellenőrző összegű cellákat a szolgálat eldobja, vagy (rosszként megjelölve) átadja az alkalmazásnak. Mind kétpontos, mind adatszórásos működés lehetséges, de ez utóbbi nem biztosítja a garantált kézbesítést.

Az AAL 3/4-hez hasonlóan az AAL 5 egyaránt támogatja az üzenet és a folyam üzemmódot. Üzenet módban az alkalmazás 1–65 535 bájt méretű datagramot adhat át az AAL rétegnek akár garantált, akár „minden tőlem telhetőt megteszek" alapon történő továbbításra. Amikor a konvergencia alréteg megkapja, kibővíti az üzenetet, és farokrészt ad hozzá. Ezt láthatjuk a 6.44. ábrán. A kiegészítés mértékét (0–47 bájt) úgy választják meg, hogy az egész üzenet hossza, beleértve a farokrészt is, 48 bájt többszöröse legyen. Az AAL 5-ben nincs konvergencia alréteg fejrész, csak egy 8 bájtos farokrész.

Az *UU (User to User)* mezőt maga az AAL nem használja. Ehelyett magasabban fekvő rétegek használhatják saját céljaikra, pl. sorszámozásra vagy nyalábolásra. A szóban forgó fölső réteg lehet a konvergencia alréteg szolgálatfüggő része is. A *Length* mező adja meg a kiegészítés nélküli valódi rakomány hosszát bájtban. A 0 értékkel az aktuális üzenetet lehet menet közben érvényteleníteni. A *CRC* mező a szokásos 32 bites ellenőrző összeg, amelyet az üzenetből, a kiegészítésből és a farokrészből (ahol a *CRC* mezőt nullázzák) számítanak. A farokrész egy nyolcbites mezeje későbbi felhasználásra van fenntartva.

Az üzenetet ezután átadja a SAR alrétegnek, amely nem fűz hozzá újabb fej- vagy farokrészt. Ehelyett 48 bájtos darabokra tördeli és ezeket átadja az ATM rétegnek továbbításra. Ezenkívül utasítja az ATM réteget, hogy az utolsó cella *PTI* mezejében billentsen be egy bitet, hogy az üzenethatárok megmaradjanak. Itt vitatkozni lehetne arról, hogy ez a protokollrétegek helytelen keverése, mert az AAL rétegnek nem volna szabad az ATM réteg fejrészének bitjeit használni. Ez megsérti a protokolltervezés legalapvetőbb szabályát és azt jelzi, hogy a rétegek definiálását talán máshogy kellett volna megtenni.

Az AAL 5 legfőbb előnye az AAL 3/4-gyel szemben a sokkal nagyobb hatékonyság. Bár az AAL 3/4 üzenetenként csak 4 bájtot szúr be, szintén hozzáad 4 bájtot min-

6.44. ábra. *Az AAL 5 konvergencia alréteg üzenetének felépítése*

den cellához is, így lecsökkenti a rakomány méretét 44 bájtra, ami hosszú üzenetek esetén 8 százalékos veszteség. Az AAL 5 kicsit nagyobb farokrészt ad az üzenetekhez, de nem hoz be minden cellába overheadet. A cellákból hiányzó sorszámot ellensúlyozza a hosszabb ellenőrző összeg, ami sorszámok használata nélkül detektálni tudja az elveszett, téves vagy hiányzó cellákat.

Az Internet közösségében várható, hogy az ATM hálózatok felé irányuló interfész normális kialakítása az IP csomagok AAL 5 adat mezőjében történő szállítása lesz. Több értekezés olvasható ezzel a megközelítéssel kapcsolatosan az RFC 1483 és RFC 1577-ben.

6.5.6. Az AAL protokollok összehasonlítása

Megbocsátunk az Olvasónak, ha úgy érzi, hogy a különböző AAL protokollok fölöslegesen hasonlónak tűnnek és átgondolatlannak tekinti őket. Az elkülönített konvergencia és SAR alrétegek haszna szintén megkérdőjelezhető, különösen mivel az AAL 5 esetében semmi sincs a SAR alrétegben. Egy kicsit továbbfejlesztett ATM réteg megfelelő módon biztosíthatta volna a sorszámozást, nyalábolást és keretezést.

A 6.45. ábrán összefoglaljuk a különböző AAL protokollok egyes eltéréseit. Ezek a hatékonyságra, hibakezelésre, nyalábolásra és az alrétegek kapcsolatára vonatkoznak.

Az AAL azt az általános benyomása kelti, hogy túl sok fajtája van túl kicsi különbséggel és érezhetők rajta a félig elvégzett munka jelei. A négy eredeti szolgálatosztályt, A, B, C és D-t végeredményben elhagyták. Az AAL 1 valószínűleg fölösleges; az AAL 2 befejezetlen; az AAL 3 és AAL 4 soha nem látta meg a napvilágot; az AAL 3/4 rossz hatásfokú, és túl rövid ellenőrző összeggel rendelkezik.

Az AAL 5-é a jövő, de még itt is van mit fejleszteni. Az AAL 5 üzeneteknek rendelkezniük kéne sorszámokkal és egy bittel, ami megkülönbözteti az adat- és a vezér-

Tétel	AAL 1	AAL 2	AAL 3/4	AAL 5
Szolgálatosztály	A	B	C/D	C/D
Nyalábolás	Nincs	Nincs	Van	Nincs
Üzenethatárok	Nincs	Nincs	Btag/Etag	PTI egy bitje
Pufferfoglalás	Nincs	Nincs	Van	Nincs
Felhasználó bájtok száma	0	0	0	1
CS kiegészítés	0	0	32 bites szóra	0–47 bájt
CS protokoll overhead (bájt)	0	0	8	8
CS ellenőrző összeg	Nincs	Nincs	Nincs	32 bit
SAR adat mező (bájt)	46–47	45	44	48
SAR protokoll overhead (bájt)	1–2	3	4	0
SAR ellenőrző összeg	Nincs	Nincs	10 bit	Nincs

6.45. ábra. *A különböző AAL protokollok néhány eltérése (CS: Convergence Sublayer)*

lőüzeneteket, hogy megbízható szállítási protokollnak lehessen használni. Mostani állapotában a megbízható szállításhoz még egy további overheadet jelentő szállítási réteget kéne a tetejére ültetni, pedig elkerülhető lett volna. Ha az egész ATM bizottság beadta volna munkáját nagyházifeladatként, a professzor valószínűleg visszadobta volna, hogy javítsák ki és akkor adják be ismét, amikor elkészült. További kritika található az ATM-ről Sterbenz és munkatársai művében (1995).

6.5.7. SSCOP – szolgálatspecifikus összeköttetés alapú protokoll

Annak ellenére, hogy ennyi eltérő AAL protokoll van, egyik sem nyújt egyszerű megbízható végtől végig terjedő szállítási összeköttetést. Azon alkalmazások részére, melyek ezt igénylik, létezik még egy AAL protokoll: **SSCOP (Service Specific Connection Oriented Protocol).** Az SSCOP-t azonban csak vezérlési célokra használják, adatátvitelre nem.

Az SSCOP felhasználói üzeneteket küldenek, amelyek egy 24 bites sorszámmal vannak ellátva. Az üzenetek legfeljebb 64 kilobájt hosszúak lehetnek, az átvitel során nem tördelődnek. Garantáltan helyes sorrendben érkeznek meg. Más megbízható szállítási protokolloktól eltérően a hiányzó üzeneteket mindig szelektív ismétléssel küldik újra és nem n visszalépéses protokollal.

Az SSCOP alapvetően egy dinamikus csúszóablakos protokoll. Minden összeköttetésre a vevő karbantart egy olyan üzenetsorszámokat tartalmazó ablakot, amelyeket venni képes, és egy bit-térképet, amin a már beérkezetteket tartja nyilván. Az ablak mérete a protokoll működése folyamán változhat.

Ami szokatlan az SSCOP működésében az a nyugták kezelési módja: nincs ráültetett nyugta. Ehelyett a küldő periodikusan megkéri a vevőt, hogy küldje vissza az ablak állapotát leíró bit-térképet. Az eredmény alapján a küldő az elfogadott üzeneteket eldobja, és frissíti a saját ablakát. Az SSCOP-ról részletesen olvashatunk Henderson (1995) művében.

6.6. Teljesítőképesség

A teljesítőképességgel kapcsolatos kérdések nagyon fontosak a számítógép-hálózatok esetében. Amikor számítógépek százai vagy ezrei vannak összekapcsolva, mindennaposak az előre nem látható következményekkel járó bonyolult kölcsönhatások. Gyakran ez az összetettség gyenge teljesítőképességhez vezet, aminek senki sem tudja az okát. A következő alfejezetben sok hálózati teljesítőképességgel kapcsolatos kérdést megvizsgálunk, hogy láthassuk, a fennálló problémákat és hogy mit lehet tenni ellenük.

Sajnos a hálózat teljesítőképességének megértése inkább művészet, mint tudomány. E mögött kevés gyakorlatban is valóban alkalmazható elmélet van. A legtöbb, amit tehetünk, hogy tapasztalatból nyert ökölszabályokat és életből vett példákat mutatunk. Szándékosan időzítettük ennek a témának a tárgyalását a TCP és ATM szállítási rétegek utánra, hogy itt mutathassunk rá azokra a pontokra, ahol a dolgok jól és azokra a pontokra, ahol rosszul mennek.

Nem a szállítási réteg az egyetlen hely, ahol teljesítőképességgel kapcsolatos kérdések felmerülnek. Már láthattunk néhányat a hálózati rétegben az előző fejezetben. A hálózati réteg mégis afelé halad, hogy leginkább forgalomirányítással és torlódásvédelemmel foglalkozzon. Az általánosabb, rendszerorientált feladatok egyre inkább a szállítással kerülnek kapcsolatba, tehát ez a fejezet megfelelő hely ezek tanulmányozására.

A következő öt részben a hálózat teljesítőképességét öt oldalról vizsgáljuk meg:

1. A teljesítőképesség problémái.

2. A hálózat teljesítőképességének mérése.

3. Rendszertervezés a teljesítőképesség növelésére.

4. Gyors TPDU feldolgozás.

5. A jövő nagy teljesítményű hálózati protokolljai.

Mellesleg nevet kell találnunk a szállítási entitások által váltott egységek számára. A TCP *szegmens* elnevezése könnyen összezavarhat, ebben a környezetben a TCP világon kívül sehol nem használják. A megfelelő ATM kifejezések: *CS-PDU, SAR-PDU* és *CPCS-PDU* ATM specifikusak. A *csomag* egyértelműen a hálózati rétegre utal, az *üzenet* az alkalmazási réteghez tartozik. Egy rendes kifejezés híján a szállítási entitások által váltott egységeket továbbra is *TPDU*-nak nevezzük. Ha egyszerre utalunk TPDU-ra és csomagra, a csomagot fogjuk használni mint gyűjtőfogalmat, mint pl. a következő mondatban: „A processzornak elég gyorsnak kell lenni ahhoz, hogy a beérkező csomagokat valós időben feldolgozhassa." Ezzel egyaránt utalunk a hálózati rétegbeli csomagra és a benne elhelyezkedő TPDU-ra is.

6.6.1. A számítógép hálózatok teljesítőképesség-problémái

Néhány teljesítőképességgel kapcsolatos problémát, mint pl. a torlódást, átmeneti erőforrás-túlterhelés okoz. Ha hirtelen nagyobb forgalom alakul ki egy routernél, mint amekkorát az kezelni képes, torlódás alakul ki és a teljesítőképesség romlik. A torlódást részletesen tanulmányoztuk az előző fejezetben.

A teljesítőképesség akkor is csökken, ha strukturális erőforrás-kiegyensúlyozatlanság áll fönn. Például, ha egy gigabites távközlési vonal csatlakozik egy kiskapacitású PC-hez, a gyenge processzor képtelen lesz elég gyorsan feldolgozni a beérkező csomagokat, tehát egy részük elvész. Ezeket a csomagokat később újraküldik, ami növeli a késleltetést, pazarolja a sávszélességet, általában véve rontja a teljesítőképességet.

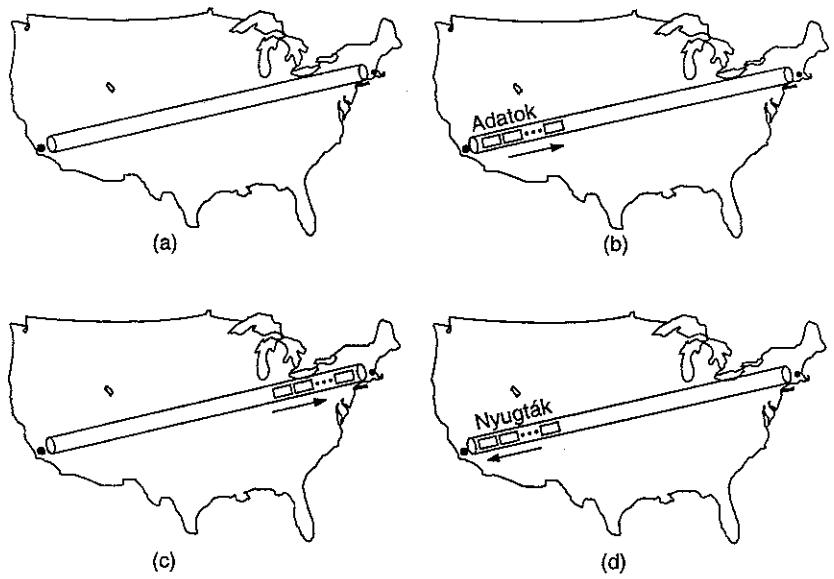
A túlterhelést egyidejű események is kiválthatják. Például, ha egy TPDU rossz paramétert tartalmaz (pl. a célport vagy célfolyamat azonosítóját), sok esetben a figyel-

mes vevő egy hibajelzést küld vissza. Most képzeljük el, hogy mi történne, ha a rossz TPDU üzenetszórással 10 000 géphez jutott el: mindegyik visszaküldhet egy hibajelzést. Az ebből kialakuló **üzenetszórás vihar (broadcast storm)** romba döntheti a hálózatot. Az UDP ebben a betegségben szenvedett, amíg a protokollt olyan értelemben meg nem változtatták, hogy a hosztok ne küldjenek hibajelzést üzenetszórásos címre érkezett TPDU-kra.

Egy másik példa egyidejű események által okozott túlterhelésre, amely egy áramszünet után következhet be. Amikor helyreáll az energiaszolgáltatás, az összes gép egyszerre nekilát a ROM-ban tárolt bootprogram végrehajtásához. Egy tipikus újraindulási menetrend megkövetelheti, hogy a hoszt először egy (RARP) szervert keressen fel, hogy megtudja valódi azonosítóját, majd egy állományszolgáltatóról letöltse az operációs rendszerét. Ha gépek százai egyszerre tesznek így, a szerver valószínűleg összeomlik a terhelés alatt.

Még ha egyidejű események nem is okoznak túlterhelést, és elegendő erőforrás áll rendelkezésre, a rendszer rossz beállítása is oka lehet a gyenge teljesítőképességnek. Például, ha egy gépnek bőven van szabad processzorkapacitása és memóriája, viszont kevés memóriát foglaltak le pufferterület céljára, ráfutások fognak előfordulni és TPDU-kat kell eldobnia. Hasonlóan, ha az ütemező nem ad elég nagy prioritást a beérkező TPDU-k feldolgozására, egy részük el fog veszni.

Egy másik beállításokkal kapcsolatos kérdés az időzítések megfelelő értékének meghatározása. Amikor egy TPDU-t elküldenek, rendszerint elindítanak egy időzítőt,



6.46. ábra. *Egy megabit továbbítása San Diegóból Bostonba. (a) t = 0-kor. (b) 500 μs múlva. (c) 20 ms elteltével. (d) 40 ms után*

ami a TPDU elvesztését figyeli. A túl rövid időtartam fölösleges újraküldésekhez vezet, terheli a vonalakat. Ha túl hosszúra állítják, egy TPDU elvesztését túlságosan hosszú késleltetések követik. További példák a beállítható paraméterekre: mennyi ideig kell adatra várni, hogy ráültetett nyugtát lehessen küldeni különálló nyugta helyett, vagy mennyi az újraküldések száma, mielőtt a küldő az ismétlést föladná.

A gigabites hálózatok új teljesítőképességgel kapcsolatos problémákat hoznak magukkal. Tekintsük például azt az esetet, amikor adatot küldünk San Diegóból Bostonba, és a vevőnek 64 kilobájtos puffere van. Tegyük fel, hogy a vonal sebessége 1 Gb/s és az üvegszálban a fénysebességből adódó késleltetés egy irányban 20 ms. Kezdetben $t = 0$-ban a csővezeték üres, ezt láthatjuk a 6.46.(a) ábrán. Alig 500 μs-mal később a 6.46.(b) ábrának megfelelően az összes TPDU úton van az üvegszálon. Az első TPDU most valahol Brawley magasságában járhat, még mindig mélyen Dél-Californiában. A küldőnek azonban meg kell állnia, amíg ablakfrissítést nem kap.

20 ms elteltével az első TPDU eléri Bostont – mint azt a 6.46.(c) ábrán láthatjuk – és a vevő nyugtázza. Végül 40 ms-mal a kezdés után az első nyugta visszaér a küldőhöz, ami elküldheti a második löketet. Mivel az átviteli vonalat csak fél ezredmásodpercig használták a 40-ből, a hatékonyság körülbelül 1,25 százalék. Ez a helyzet tipikusan akkor fordul elő, ha régi protokollokat használnak gigabites vonalak fölött.

Egy hasznos mennyiséget érdemes megjegyezni, ha számítógép-hálózatok teljesítőképességének vizsgálatával foglalkozunk. Ez a **sávszélesség-késleltetés szorzat (bandwidth-delay product),** ami úgy áll elő, hogy megszorozzuk a b/s-ban mért sávszélességet a másodpercben mért körülfordulási idővel. A szorzat a küldőtől a vevőig és vissza terjedő csővezeték bitben mért kapacitása.

Például a 6.46. ábrán a sávszélesség-késleltetés szorzat 40 millió bit. Másképpen szólva a küldőnek 40 millió bites löketet kéne folyamatosan teljes sebességgel adnia, amíg az első nyugta vissza nem érkezik. Ennyi sok bit szükséges a csővezeték teletöltéséhez (mindkét irányban). Ezért jelent egy félmillió bites löket csak 1,25 százalékos hatékonyságot: a csővezeték kapacitásának csak 1,25 százaléka.

Azt a tanulságot vonhatjuk le, hogy jó teljesítőképesség eléréséhez a vevő ablakának legalább akkorának kell lennie, mint a sávszélesség-késleltetés szorzat, lehetőleg még egy kicsit nagyobbnak, hiszen a vevő esetleg nem tud azonnal válaszolni. Egy kontinenseket összekötő gigabites vonalon minden összeköttetés számára legalább 5 megabájt szükséges.

Ha a hatékonyság már egy megabit átvitelekor is szörnyű, képzeljük el, hogy milyen lesz, ha pár száz bájtot küldünk egy távoli eljáráshíváshoz. Hacsak nem találunk valami más feladatot a vonalnak, amíg a kliens válaszra vár, a gigabites vonal nem jobb egy megabites vonalnál, csak drágább.

Egy másik teljesítőképességgel kapcsolatos probléma, a dzsitter (csomagok érkezési idejének változása) időkritikus alkalmazásoknál – mint pl. hang- és videoátvitel – lép föl. Nem elegendő, ha az átviteli idők átlaga kicsi, a szórásának is kicsinek kell lennie. Komoly tervezői erőfeszítést igényel rövid átlagos átviteli idő és kis szórás egyidejű megvalósítása.

6.6.2. A hálózat teljesítőképességének mérése

Amikor egy hálózat gyengén teljesít, a felhasználók gyakran panaszkodnak a működtetőknek és fejlesztést követelnek. A teljesítőképesség javításához az operátoroknak először is meg kell állapítaniuk, hogy mi is történik pontosan. Hogy megtudják, mi történik valójában, méréseket kell végezniük. Ebben a részben a hálózatok teljesítőképességének mérését fogjuk bemutatni. Az alábbiakban leírtak Mogul munkáján (1993) alapulnak. A mérési folyamat teljesebb tárgyalását Jain (1991), Villamizan és Song (1995) műveiben találhatjuk.

A hálózat teljesítőképességének javítása a következő lépések ismételt végrehajtásával lehetséges:

1. Megmérjük a fontosabb hálózati paramétereket és teljesítőképességet.

2. Megpróbáljuk megérteni, hogy mi is történik.

3. Egy paramétert megváltoztatunk.

Ezeket a lépéseket addig ismételjük, amíg a teljesítőképesség már elég jó nem lett, vagy már mindent kifacsartunk a hálózatból.

A méréseket (mind fizikailag és a protokoll készlet minden szintjén) sokféleképpen és sok helyen lehet végezni. A legalapvetőbb mérés az időmérés, amikor valamilyen tevékenység kezdetén egy időzítőt indítunk, amellyel megmérjük, hogy az adott tevékenység mennyi időt vesz igénybe. Például annak ismerete, hogy mennyi időt igényel egy TPDU nyugtázása, kulcsfontosságú. Más mérések számlálók felhasználásával lehetségesek, ezek valamely esemény gyakoriságát rögzítik (pl. elvesztett TPDU-k száma). Végül gyakran fontos tudni valaminek a mennyiségét, például egy bizonyos időintervallum alatt feldolgozott bájtok számát.

A hálózat teljesítőképességének és paramétereinek mérése számos potenciális buktatót rejt. Az alábbiakban néhányat fölsorolunk. Minden hálózati teljesítőképesség mérésére tett szisztematikus próbálkozás során gondosan el kell kerülni ezeket.

Győződjünk meg, hogy elég nagy-e a minták száma

Ne egyetlen TPDU elküldésének idejét mérjük, hanem ismételjük meg a mérést mondjuk egymilliószor és vegyük az átlagot. Nagyszámú minta vizsgálata a mért átlag és szórás bizonytalanságát is csökkenti. Ezt a bizonytalanságot statisztikai képletekkel határozhatjuk meg.

Bizonyosodjunk meg arról, hogy reprezentatív mintákat használunk

Ideális esetben az egymillió mérést különböző napszakokban és a hét napjain meg kéne ismételni, hogy láthassuk a különböző rendszerterheléseknek a mért mennyiségre gyakorolt hatását. A torlódáson végzett mérések például kevés haszonnal járnak, ha a mérés pillanatában nincs is torlódás. Néha az eredmények első ránézésre valószínűtlennek tűnhetnek, mint pl. súlyos torlódás 10, 11, 1 és 2 órakor, viszont nincs torlódás délben (amikor az összes felhasználó ebédelni ment).

Bánjunk óvatosan a durva felbontású órával

A számítógépes órák úgy működnek, hogy szabályos időközönként egy számláló értékét eggyel növelik. Például egy ezredmásodperces időzítő ezredmásodpercenként egyet ad a számláló értékéhez. Ilyen időzítő felhasználásával nem lehetetlen 1 ms-nál rövidebb eseményt mérni, csak gondosságot igényel.

Hogy például egy TPDU elküldéséhez szükséges időt megmérjük, kétszer kell leolvasni a rendszerórát (mondjuk ezredmásodpercekben) amikor belépünk a szállítási réteg kódjába és amikor elhagyjuk azt. Ha a TPDU tényleges elküldési ideje 300 μs, a két olvasás különbsége 0 vagy 1 lesz, ami egyaránt rossz. Ha azonban a teljes mérést egymilliószor megismételjük, és az összes mérési eredményt összeadjuk és egymillióval elosztjuk, az átlagos idő 1 μs-nél is pontosabb lesz.

Győződjünk meg arról, hogy mérés közben nem következik be váratlan esemény

Ha azon a napon végzünk egy egyetemi rendszeren méréseket, amikor egy nagyobb laborfeladatot kell beadni, más eredményeket kaphatunk, mint ha a mérést a következő napon végeznénk. Hasonlóan, ha néhány kutató úgy döntött, hogy videokonferenciát tart a mérés alatt levő hálózaton, hibás eredményeket kaphatunk. A legjobb tétlen rendszeren végezni a méréseket és a teljes terhelést saját kezűleg generálni, azonban még ennek a megközelítésnek is akadnak buktatói. Amíg úgy gondolnánk, hogy senki sem fogja használni a hálózatot éjjel háromkor, éppen ekkor foghat hozzá egy önműködő másolatkészítő program az összes merevlemez videoszalagra másolásához. Továbbá erős forgalmat generálhatnak a csodálatos World Wide Web oldalainkat nézegető más időzónában élő felhasználók.

A gyorsítótár működése romba döntheti a mérést

Az állományátvitel idejének mérése nyilvánvalóan a következő módon történhet: egy nagy állományt megnyitunk, az egészet beolvassuk, majd lezárjuk és megnézzük, hogy ez mennyi ideig tartott. Az egész mérést sokszor megismételjük, hogy egy jó

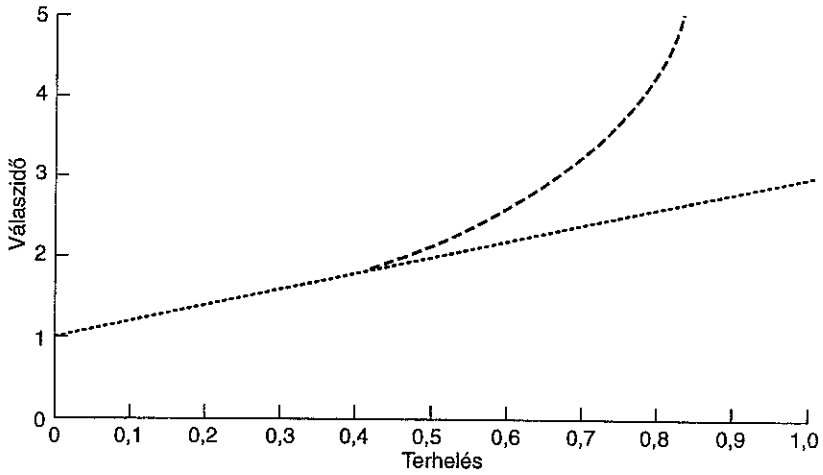
átlagot kapjunk. Akkor van baj, ha a rendszer gyorsítótárba (cache) rakja az állományt, ezért csak az első mérés alatt folyik tényleges hálózati forgalom. A maradék csak a helyi gyorsítótár olvasása. Az ilyen mérésekből származó eredmények lényegében értéktelenek (hacsak nem a gyorsítótár teljesítőképességére vagyunk kíváncsiak).

Gyakran megkerülhetjük a cache használatát, ha túlcsordulást okozunk benne. Például, ha a gyorsítótár kapacitása 10 megabájt, egy ciklus minden menetben megnyithat, beolvashat és lezárhat két 10 megabájtos állományt, hogy megpróbálja a cache találatok számát 0-ra szorítani. Még ekkor is érdemes vigyázni, hacsak teljesen biztosan nem értjük a cache algoritmus működését.

A pufferelésnek hasonló hatása lehet. Egy népszerű TCP/IP teljesítőképességet vizsgáló segédprogram arról volt ismert, hogy az UDP teljesítőképességére a fizikai vonal által megengedettnél jóval nagyobb értéket közölt. Hogy történhetett ez meg? Egy UDP hívás normálisan akkor adja vissza a vezérlést, amikor az üzenetet elfogadta a mag, és beillesztette a továbbítási sorba. Ha elegendő pufferterület van, 1000 UDP hívás nem jár együtt az összes adat tényleges továbbításával. A legtöbb üzenet esetleg még mindig a magban várakozik, de a segédprogram úgy gondolja, hogy már mindet elküldték.

Értsük meg azt, amit mérünk

Amikor egy távoli állomány beolvasásának idejét mérjük, a mérések függnek a hálózattól, mindkét gép operációs rendszerétől, az esetünkben használt illesztőkártyáktól, azok meghajtóprogramjától és egyéb tényezőktől. Ha gondosan dolgozunk, végül megkapjuk az állományátvitel idejét a használt konfigurációra vonatkozóan. Ha a célunk ennek a bizonyos konfigurációnak a beállítása, ezek a mérések tökéletesek.



6.47. ábra. *A válaszidő a terhelés függvényében*

Ha azonban három különböző rendszeren végzünk hasonló méréseket, hogy eldönthessük, melyik hálózati illesztőkártyát vásároljuk meg, az eredményeinket teljesen meghamisíthatja, ha az egyik meghajtóprogram csapnivaló, és a kártya teljesítőképességének csak 10 százalékát használja.

Vigyázzunk az eredmények extrapolálásával

Tegyük fel, hogy valamit szimulált hálózati terhelésnél mérünk. A terhelés 0 (tétlen) értéktől 0,4-re (a kapacitás 40 százaléka) nő, ezt jelzik a 6.47. ábrán látható pontok és a rájuk fektetett folytonos egyenes. Nehéz ilyenkor ellenállni a kísértésnek, hogy lineárisan extrapoláljuk, amit a pontozott vonal mutat. A sorban állás behoz azonban egy $1/(1-\rho)$ tényezőt is, ahol a ρ a terhelés, ezért az igazi eredmény inkább a szaggatott görbére emlékeztet.

6.6.3. Rendszertervezés a teljesítőképesség növelésére

Mérések végzése és javítgatás jelentős mértékben növelheti a teljesítőképességet, de nem helyettesítheti az elsődleges fontossággal bíró gondos tervezést. Egy gyengén megtervezett hálózatot csak úgy-ahogy lehet javítani, azon túl az alapoktól kell újrakezdeni.

Ebben a részben néhány ökölszabályt mutatunk be, melyek sok hálózaton végzett munka tapasztalatain alapulnak. Ezek a szabályok rendszertervezéssel is kapcsolatosak, nem csak hálózatok tervezésével, hiszen a szoftver és az operációs rendszer sokszor fontosabb a routereknél és a csatolókártyáknál. Ezen ötletek nagy része hosszú évek óta a hálózattervezők közös ismerete, és szájhagyomány útján terjed generációról generációra. Először Mogul (1993) rögzítette őket; a mi megközelítésünk nagyjából párhuzamos az övével. Egy másik ide kapcsolódó forrás Metcalfe (1993) műve.

Első szabály: a processzor sebessége fontosabb a hálózat sebességénél

Sok tapasztalat szerint szinte minden hálózatban az operációs rendszer és a protokoll overheadje határozza meg a vonalon töltött idő döntő részét. Például, elméletileg egy Etherneten eltöltött legrövidebb RPC végrehajtási idő 102 μs, ami megfelel egy minimális (64 bájtos) kérésnek és az azt követő minimális (szintén 64 bájtos) válasznak. A gyakorlatban jelentős eredménynek számít (Van Renesse és munkatársai, 1988), ha az RPC válaszidejét sikerül 1500 μs alá szorítani. Vegyük észre, hogy az 1500 μs az elméleti minimumnál 15-ször rosszabb. Szinte az egész overhead a szoftverben, a szoftver miatt van.

Hasonlóan az 1 Gb/s sebességű átvitel legnagyobb problémája a bitek elég gyors átvitele felhasználói pufferből az üvegszálra, és a vevő processzorral olyan sebességű