Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai Tanszék

Robusztus, torlódásszabályozás nélküli transzport protokoll tervezése és fejlesztése

Szakdolgozat

Készítette
Solymos Szilárd

Konzulens
Dr. Molnár Sándor
egyetemi docens

2011. december 9.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Solymos Szilárd*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2011. december 9.

Solymos Szilárd
hallgató

Kivonat

Az Internet születésétől fogva az elmúlt 35 évben főként a TCP (Transmission Control Protocol) által használt torlódásszabályozás volt az alkalmazott szabályozó mechanizmus. Az Internet folyamatos változásának következtében újabbnál újabb TCP verziókat fejlesztettek ki. Ezen TCP verziók a hagyományos torlódásvezérlő algoritmus megváltoztatásával képesek kezelni a hálózati erőforrások bizonyos esetekben fellépő alacsony kihasználtságát. Így a jelenleg használt TCP verziók képesek hatékony megoldást nyújtani néhány meghatározott hálózati környezet esetén, azonban nem képesek univerzális, optimális megoldást nyújtani a folyamatosan változó, heterogén környezet okozta kihívásokra. Úgy tűnik, hogy kevés remény van arra, hogy a TCP által használt zárt hurkú torlódásszabályozás a jövőben képes lesz univerzális megoldást nyújtani ezekben az esetekben.

Egy GENI (Global Environment for Network Innovations) által javasolt alternatív megoldás a jövő Internetére nézve az lehet, hogy egyáltalán nem alkalmazunk torlódásszabályozást, és a hálózatban minden entitás esetén maximális sebességgel történik az adatküldés. Ilyen módon lehetségessé válik a hálózati erőforrások teljes kihasználása. Az ekkor fellépő főleg csomós csomagvesztést hatékony hibajavító kódolás segítségével kezeljük, és ilyen módon történik az elküldött adatok helyreállítása a vevőnél. Ebben a dolgozatban egy ezen az elven működő szállítási rétegbeli protokollt mutatok be. A protokoll implementációja Linux kernelben történt, ezért röviden tárgyalom a Linux kernel hálózati alrendszerének felépítését, és alapvető működését, majd ennél jóval részletesebben ismertetem a felhasznált koncepciót, és annak előnyeit, valamint hátrányait, és ezt a koncepciót alkalmazó új protokoll tulajdonságait is. Láthatjuk a protokoll működésének folyamatát, különböző fázisait, valamint az alkalmazott kódolásokat, és ennek konkrét megvalósítását is. Végül bemutatom a protokoll működésének befolyásolására alkalmas paramétereket is. Minden esetben táblázatok, ábrák szemléltetik az egyes folyamatokat, és ezekhez kapcsolódó magyarázatok által ismertetem a működést.

A protokoll működésének leírása után láthatjuk az elvégzett teszthálózati mérések során felhasznált programokat, a hálózati topológiákat, és az adott konkrét elrendezés esetén alkalmazott beállításokat. A mérések során az új protokoll teljesítményét a TCP protokoll Linux kernelben található jelentősebb verziói által hasonló forgalmi körülmények között elért teljesítménnyel hasonlítom össze, és ismertetem, majd értékelem ezeknek a méréseknek az eredményeit. Ennek során számos ábrával szemléltetem a protokoll által elért teljesítményt. Végül kitérek az új protokollal kapcsolatos jövőbeli tervekre, és fejlesztési lehetőségekre is.

Abstract

Congestion control is one of the most important control mechanisms mostly managed by the TCP (Transmission Control Protocol) from the beginning of the Internet over the last 35 years. This mechanism tries to avoid low utilization of network resources in cases, when links are overloaded by transmitting end hosts. Since the Internet had been changing continuously, new and new TCP versions have been developed. Current TCP variants provide more efficient solutions by modifying the existing congestion control algorithm. The new algorithms has been widely varied in order to fit the requirements in ever-changing network environments. From backbone to wireless networks, different conditions demand changes and improvements on the TCP, but these new TCP variants only work for some important network environments and all of them fail to act as universal mechanism considering heterogeneous and changing network conditions. It seems that there is little hope that TCP's closed-loop congestion control mechanism could result such a universal solution in the future.

Some ideas have been proposed and investigated where congestion control was not employed at all. One of these ideas was proposed by GENI (Global Environment for Network Innovations). This idea advocates an alternative solution for the future Internet, which can be to avoid congestion control and allow users to send at maximal rate, so every network resource would always be fully utilized. If end point access rates do not cause congestion, then this idea yields the most efficient solution. However, when hosts send data at their maximal rates, excessive packet loss is likely to occur, mostly in bursty manner. The emerging huge packet loss is handled by the use of efficient erasure coding, which makes it possible to recover the original message at the receiver. In this work I introduce a new transport layer protocol, which uses the new idea, so congestion control is avoided, and packet loss is corrected by efficient erasure codes. The implementation of this protocol can be found in a modified Linux kernel, so firstly I will provide a quick overview of the core elements of the Linux networking subsystem. After that I will show the new concept, its advantages, disadvantages, and the properties of the new protocol in more detail. You will see the different working phases of the protocol, the applied coding schemes, and their implementation. Finally I will explain the parameters, which can be used to change the behavior of the protocol in some important situations. In all cases I use figures, tables, and flowcharts to demonstrate the processes used by the protocol.

After that I will introduce the programs, network topologies, and the applied configurations used to measure the performance of the new protocol compared to some important

TCP variants, which can be found in the Linux kernel. Finally I discuss the future works on improving the performance of the new protocol.

1. fejezet

Bevezetés

Napjainkban az Interneten a TCP (Transmission Control Protocol) az egyik meghatározó szállítási rétegbeli protokoll. A TCP kapcsolat alapú, megbízható, sorrendhelyes adatátvitelt tesz lehetővé az alkalmazási réteg számára. Jelenleg az Internet szerves része a torlódásszabályozás, amely mechanizmus célja a hálózati erőforrások kihasználásának javítása, és a linkek, átviteli utak túlterhelésének megakadályozása. Ebben az esetben a végpontok különböző tényezők, mint például a csomagvesztés, a késleltetés, és egyéb szempontok alapján úgy választják meg az átviteli sebességüket, hogy a torlódás ilyen módon elkerülhető legyen. A TCP protokoll a torlódásszabályozás alkalmazásával képes a stabilitást, és az igazságos működést biztosítani, így alkalmazkodni az állandóan változó hálózati környezethez. A TCP-t széleskörűen alkalmazzák számos különböző hálózati környezet esetén, a gerinchálózatoktól, és optikai hálózatoktól kezdve a vezeték nélküli környezetig, amelyek számos kihívást jelentenek a hatékony működésre nézve. Annak érdekében, hogy a TCP ezeknek a kihívásoknak minél jobban megfeleljen, számos különböző verzióját fejlesztették ki, amelyek valamilyen szempontból hatékonyabb működést próbáltak elérni egy adott környezet esetén. A hálózat minél jobb kihasználása érdekében kifejlesztett verziók ugyan megoldást jelentettek néhány területen, de mégsem nyújtanak egy univerzális mechanizmust a heterogén, állandóan változó hálózati környezet esetén, nem képesek jól kihasználni a rendelkezésre álló erőforrásokat. További problémát jelent, hogy egy újabb kifejlesztett TCP változat esetén együttműködési problémák lépnek fel a többi, már létező változattal.

A tapasztalt problémák miatt olyan új megoldásokra, mechanizmusokra van szükség, amelyeket alkalmazva képesek vagyunk teljes mértékben kihasználni a rendelkezésre álló erőforrásokat, és közben univerzális megoldást nyújtani a számos különböző környezet esetén a torlódásszabályozásra. Az ehhez kapcsolódó kutatások során a GENI (Global Environment for Network Innovations) [1] egy olyan új megoldást javasolt, amely szerint ne alkalmazzunk egyáltalán torlódásszabályozást, hanem minden végpont maximális sebességgel küldheti a rendelkezésre álló adatokat. Ebben az esetben az okozott túlterhelés miatt csomagvesztések történnek. Az elküldött adatokat, információkat hibajavító kódolás alkalmazásával állíthatjuk helyre. Ennek a megoldásnak számos előnye van. A TCP protokollal szemben képes teljes mértékben kihasználni a hálózati erőforrásokat. Ezen kívül fontos az új módszer egyszerűsége, stabilitása, ugyanis a maximális sebességű adatküldés a hálózati

forgalmat sokkal inkább becsülhetővé teszi, mint a TCP, ahol a küldési sebesség nagymértékben ingadozhat. Végül fontos, hogy az új módszer támogatja az optikai hálózatokat, ahol fizikai okokból csak kis méretű bufferek alkalmazására van lehetőség.

A munkám során bekapcsolódtam a *HSN Laboratóriumban* végzett ez irányú kutatásokba, és a szakdolgozatomban egy olyan szállítási rétegbeli protokollt mutatok be, amely a fent leírt új módszert alkalmazza, tehát nem használ torlódásszabályozást, és a maximális sebességű küldés esetén fellépő csomagvesztést hatékony hibajavító kódolás alkalmazásával állítja helyre. A dolgozatban bemutatom a protokoll teljesítményét a különböző hálózati topológiákon elvégzett mérések során, ismertetem ezen mérések eredményeit, valamint összehasonlítom a TCP protokoll egyes verziói által elért teljesítménnyel az adott körülmények között.

A bevezetés után a jelenlegi nagysebességű hálózati környezetekben elterjedten használt, torlódásszabályozást alkalmazó szállítási rétegbeli protokollokat mutatom be, különös tekintettel a Linux operációs rendszer esetén használt TCP változatokra. Itt először a torlódásszabályozással kapcsolatos alapfogalmakat ismertetem, majd rátérek a TCP verziók bemutatására. Minden változat esetén láthatjuk majd, hogy milyen környezet esetén alkalmazzuk, milyen problémákra nyújt megoldást, valamint, hogy az adott változat hogyan módosítja az eredeti torlódásszabályozó algoritmust, milyen fázisokat használ a működés során, milyen tulajdonságokkal rendelkeznek ezek a fázisok. Végül, összefoglalom az alkalmazott algoritmus által elért előnyöket, és a felmerülő problémákat is. A következő részben a torlódásszabályozás nélküli koncepciót ismertetem, amely képes lehet javítani az erőforrások kihasználtságán, és a TCP esetén jelentkező problémákra is megoldást nyújthat. Majd, az ezen koncepció szerinti működést megvalósító új protokoll specifikációját adom meg. Ennek során először röviden bemutatom a protokoll által alkalmazott fázisokat, és bővebben ismertetem a megvalósított kódolás, illetve dekódolás folyamatát. Az utolsó részben az új protokollal elvégzett méréseket láthatjuk. Itt bemutatom a mérések során felhasznált programokat, a topológiákat, és konkrét megvalósításukat, a tesztelés során alkalmazott beállításokat, majd ismertetem az elvégzett mérések eredményeit, az azokból levont következtetéseket. Végül, a protokollal kapcsolatos jövőbeli terveket, fejlesztési lehetőségeket láthatjuk. A függelék tartalmazza a mérésekhez kapcsolódó pontos eredményeket is.

2. fejezet

Szállítási protokollok nagysebességű hálózati környezetben

Ebben a fejezetben áttekintem a különböző hálózati környezetek esetén alkalmazott szállítási rétegbeli protokollokat, TCP verziókat, amelyek az adott környezetek esetén képesek növelni a TCP teljesítményét. Kiemelt figyelmet kap a nagysebességű hálózati környezetben alkalmazott TCP verziók tárgyalása [2], és a Linux operációs rendszer esetén a torlódásszabályozásra alkalmazott megoldások bemutatása.

Annak érdekében, hogy a TCP a különböző környezetek esetén minél hatékonyabban legyen képes működni, a torlódásszabályozó algoritmus különböző változatai jelentek meg. Ezen torlódásszabályozó mechanizmusokat két nagy csoportba sorolhatjuk. Az első csoportba tartoznak a *csomagvesztés alapján* működő verziók. Ezen változatok a csomagvesztést használják fel a torlódásvezérléshez, ami egybites információt jelent. Ebbe a csoportba tartozó TCP változat például a TCP Reno [3], a HSTCP (High Speed TCP) [4], a Scalable TCP [5], a Linux kernel esetén alkalmazott BIC TCP (Binary Increase Congestion control) [6], valamint CUBIC TCP [7]. A jelenlegi Linux kernelek esetén az alapértelmezett verzió a CUBIC TCP. A másik csoportot a *késleltetés alapú* megközelítések alkotják, amelyek a torlódásszabályozáshoz a *körülfordulási időt* használják fel, amelynek rendszeres mérésével a küldési sebességet a késleltetés értékének megfelelően választják meg. A körülfordulási idő egy csomag elküldése, és a rá vonatkozó nyugta megérkezése között eltelt időt jelenti. Késleltetés alapú verzió például a FAST TCP [8], amely jól skálázható, és együttműködés szempontjából is kedvező tulajdonságokkal bír. Léteznek *hibrid változatok* is, amelyek mindkét megközelítést alkalmazva próbálják azok előnyeit egyesíteni. Ilyen például a Windows 7, Windows Vista, és Windows Server 2008 operációs rendszerekben megtalálható Compound TCP [9], amely a csomagvesztésen túl egy skálázható, késleltetés alapú komponenst is használ, és ezen két komponens együttesen határozza meg a csomagküldés sebességét. Ezen kívül külön osztályt képviselnek azok a protokollok, amelyek a torlódásszabályozást *explicit értesítések* segítségével oldják meg. Az ilyen módon működő protokollok hátránya, hogy módosításokat igényelnek a hálózati eszközökben annak érdekében, hogy azok támogassák az adott torlódásszabályozási módszert. Az explicit megoldást alkalmazó csoportba tartozó protokoll az XCP (eXplicit Control Protocol) [10]. Az XCP

protokollt alkalmazó routerek a működésük során értesítik a küldőket és a vevőket a hálózat állapotáról, az aktuális torlódásokról.

A következőkben néhány fontos alapfogalmat ismertetek, amelyekre a későbbiekben számos esetben fogok hivatkozni a TCP verziók működésének ismertetése során.

2.1. Alapfogalmak

A különböző TCP verziók esetén alkalmazott torlódásszabályozás első lépése a torlódás detektálása. A korai hálózatokban, ha egy időzítő lejárt egy elveszett csomag miatt, akkor annak oka egyaránt lehetett zaj az átviteli vonalon, vagy hogy egy router eldobta azt. A jelenlegi hálózatokban az átviteli hibákból eredő csomagvesztés viszonylag ritka, mert a legtöbb nagytávolságú trönk üvegszálas. Ennek az a következménye, hogy a legtöbb időtúllépést a hálózatban fellépő torlódás eredményezi. Az összes, TCP esetén alkalmazott algoritmus feltételezi, hogy az időtúllépés torlódás miatt következik be. Később látni fogjuk, hogy ez a megközelítés egyes esetekben (például vezeték nélküli átvitel esetén, ahol a csomagvesztés természetes velejárója a kommunikációnak) rontja a TCP teljesítőképességét, és ezen kívül egyéb problémák is fellépnek, amelyek kevésbé optimálissá teszik a TCP működését.

Az egyes TCP verziók több különböző időzítőt is használhatnak a feladatuk elvégzéséhez. Ezek közül a legfontosabb az *ismétlési időzítő* [11] (retransmission timer). Egy szegmens elküldésekor a TCP ezt az időzítőt elindítja. Ha a szegmensre az időzítő lejárta előtt nyugta érkezik, akkor az időzítő leáll. Ha viszont az időzítő még a nyugta beérkezése előtt lejár, akkor az adott szegmenst újraküldi a TCP (és az időzítőt is újraindítja). Ha az időzítő esetén használt időtartamot túl rövidre állítjuk, akkor felesleges újraküldések történnek, amelyek a hálózatot terhelik. Ha az időtartam túl hosszú, akkor pedig a teljesítőképesség csökken egy csomag elvesztésekor a hosszú újraküldési késleltetés miatt. Ahhoz, hogy meghatározhassuk, hogy mennyi ideig fusson az időzítő az RTT (Round-Trip Time) értékét használjuk fel.

Az RTT alatt a körülfordulási időt értjük, amely tehát egy csomag elküldése, és a rá vonatkozó nyugta megérkezése között eltelt időt jelenti. A nyugta érkezési idejének átlaga, és szórásnégyzete is jelentősen változhat néhány másodperc alatt, ha torlódás lép fel, vagy szűnik meg, ezért ennek becslésére a megoldás egy dinamikus algoritmus alkalmazása, amely a hálózat teljesítőképességének folyamatos mérése alapján állandóan újra beállítja az időintervallumot. A TCP esetén használt algoritmus Jacobson nevéhez fűződik, és a következőképpen működik. A TCP minden összeköttetés esetén nyilvántartja a rá vonatkozó RTT változót, amely a rendeltetési helyig terjedő körülfordulási idő legjobb jelenlegi becsült értéke. Egy szegmens elküldésekor a TCP egy időzítő segítségével méri, hogy mennyi idő alatt ér vissza a nyugta. A nyugta megérkezésekor a TCP megméri, hogy mennyi ideig tartott (M), ez után az RTT értékét a 2.1. képlet szerint frissíti:

$$RTT = \alpha \cdot RTT + (1 - \alpha) \cdot M \tag{2.1}$$

Itt α egy súlyozó tényező, amelynek tipikus értéke 7/8. Az időzítés esetén használt késlel-

tetés megválasztásakor a TCP korai implementációi általában a $\beta \cdot RTT$ értéket használták, ahol β mindig 2 értékű volt, amely rugalmatlanul viselkedett, ezért 1988-ban Jacobson javaslata alapján módosított algoritmus egy másik, csúszóátlagolással előállított változót is igényelt, amely a D-vel jelölt szórás. Amikor nyugta érkezik a TCP mindig kiszámítja a várt és a megfigyelt értékek $|RTT - M|$ különbségét, ennek egy csúszóátlagolással számított értékét tárolja D, a 2.2. képlet szerint:

$$D = \alpha \cdot D + (1 - \alpha) \cdot |RTT - M| \tag{2.2}$$

Itt α más érték is lehet, mint amit az RTT meghatározásakor használtunk. Végül is az időzítés (T) hosszát a 2.3. képlet adja meg:

$$T = RTT + 4 \cdot D \tag{2.3}$$

A 4-es szorzótényező választása valamennyire tetszőleges, de két jelentős előnnyel is jár. Először is, a néggyel történő szorzás egyetlen eltolással megvalósítható. Másodszor, lecsökkenti a fölösleges időtúllépések, és újraküldések számát.

A számítógép-hálózatok teljesítményanalízisekor egy fontos paraméter az úgynevezett *sávszélesség-késleltetés szorzat* (BDP: Bandwidth-Delay Product), amely a küldőtől a vevőig, és a vevőtől a küldőig terjedő csővezeték kapacitását adja meg bitekben, vagy bájtokban. Ezt a mennyiséget a 2.4. egyenlet szerint számíthatjuk ki:

$$BDP = B \cdot RTT \tag{2.4}$$

Itt B jelenti a sávszélességet, RTT pedig a körülfordulási időt. Például egy 1 Gbps sávszélességű, és 1 ms körülfordulási idővel rendelkező hálózat esetén a 2.5. egyenlet szerint számíthatjuk ki ezen értéket:

$$BDP = 10^9 \frac{b}{s} \cdot 10^{-3} s = 10^6 b = 125 kB \tag{2.5}$$

Fontos fogalom az MTU (Maximum Transmission Unit), amely meghatározza az adott közegen átvihető, legnagyobb csomag méretét. Itt a *csomag* a hálózati rétegbeli adategység elnevezése.

A szállítási rétegbeli protokollok adategységét *szegmensnek* nevezzük. Az előző fogalomhoz kapcsolódóan lényeges az MSS (Maximum Segment Size) definiciója is, amely azt a legnagyobb szegmensméretet jelenti, melyet tördelés nélkül lehet belerakni egy csomagba.

A következő fogalom a különböző protokollok jellemzésére használható *válaszfüggvény*. Egy protokoll válaszfüggvénye az átlagos átbocsátóképességet adja meg egy adott kapcsolat esetén a csomagvesztési valószínűség, és egyéb tényezők (például RTT, MSS) függvényében. A válaszfüggvény alapján következtethetünk az adott protokoll TCP kompatibilis tulajdonságára, RTT igazságosságára, konvergenciájára, és skálázhatóságára.

Végül, a különböző torlódásszabályozó algoritmusok a küldő oldalon egy *torlódási ablakot* használnak, amely meghatározza az egy RTT alatt továbbítható csomagok számát.

A fejezet további részében bővebben tárgyalom a fent bemutatott első két csoporthoz tartozó TCP verziók tulajdonságait, és az alkalmazott torlódásszabályozási algoritmust. Itt ismertetem többek között az alkalmazott módszerek előnyeit, és hátrányait is.

2.2. TCP Tahoe, és TCP Reno

Ezen két TCP verzió elnevezése a 4.3BSD operációs rendszerhez kötődik [12]. A TCP Tahoe az 1988-ban megjelenő 4.3BSD-Tahoe részeként, a TCP Reno pedig az 1990-es 4.3BSD-Reno részeként mutatkozott be. A Reno elnevezés a Nevada állambeli *Reno* városára, a Tahoe elnevezés pedig a szintén USA-ban található *Tahoe* elnevezésű tóra utal. A TCP Reno a TCP Tahoe továbbfejlesztett, hatékonyabb működésű változata. Ebben az alfejezetben először a két TCP verzió közös tulajdonságait ismertetem, majd kitértek a köztük levő különbségekre is.

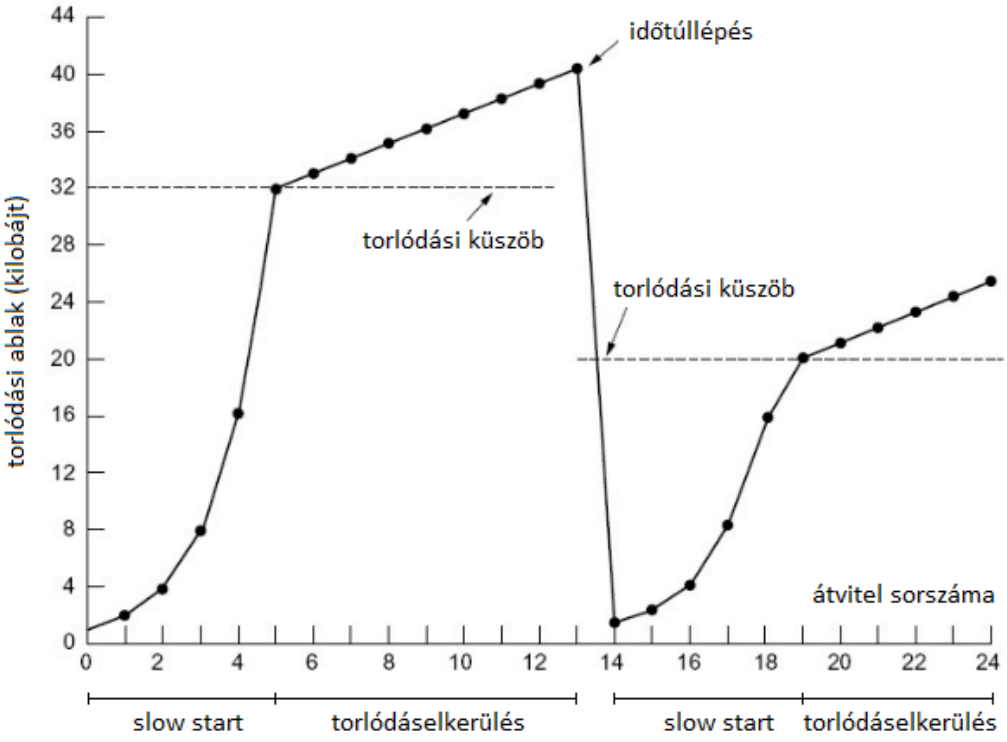
A hálózati torlódások kezelése esetén két potenciális probléma létezik. Az egyik a vevő kapacitása, a másik a hálózat kapacitása. A két problémát külön kell kezelni, ennek megfelelően minden küldő két ablakot használ. Az egyik ablakot a vevő szabályozza, a másik pedig a *torlódási ablak*. Mindkettő ablak az adó által elküldhető bájtok számát határozza meg. A ténylegesen továbbítható bájtok száma a két ablak értékének minimuma lesz, azaz a küldő és a fogadó által is jónak tartott érték minimumát használjuk.

Amikor az összeköttetés létrejön, a küldő a torlódási ablak kezdeti értékét az összeköttetésben használható legnagyobb szegmensméretre állítja be. Ez után elküld egy maximális méretű 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. 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 fel, vagy el nem éri a vevő ablakméretét, exponenciálisan növekszik. Ezt a technikát *Slow Start* algoritmusnak nevezzük, és minden TCP implementációnak támogatnia kell.

A torlódásvédelmi algoritmus egy harmadik, *torlódási küszöbnek* nevezett paramétert is használ, amelynek kezdőértéke 64 kilobájt. Amikor időtúllépés következik be, akkor a torlódási küszöböt az aktuális torlódási ablak méretének felére állítjuk be, és a torlódási ablakot a maximális szegmensméretre állítjuk vissza. Ez után újból a Slow Start algoritmust használjuk a hálózati teljesítőképességének meghatározására, de az exponenciális növekedés véget ér, ha 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. Az algoritmusnak ezt a fázisát *torlódáselkerülési fázisnak* nevezzük. A TCP protokoll előzőekben ismertetett torlódásvédelmi mechanizmusát összefoglaló néven *AIMD* (Additive Increase Multiplicative Decrease) mechanizmusnak nevezzük.

Az eddig leírt működés illusztrálására tekintsük a 2.1. ábrát. A maximális szegmensméret itt 1024 bájt. Kezdetben a torlódási ablak 64 kilobájt volt, de időtúllépés történt, ezért

a torlódási küszöböt 32 kilobájtra állítottuk be, a 0. átvitelhez használt torlódási ablakot pedig 1 kilobájtra. Innentől kezdve a növekedés exponenciális, amíg el nem érjük a torlódási küszöböt, ami 32 kilobájt értékű. Ettől kezdve a növekedés lineáris. A 13. átvitel esetén időtúllépés történik.



2.1. ábra. A TCP Tahoe és TCP Reno torlódásszabályozása

A küszöböt az aktuális ablakméret felére állítjuk (mostanra 40 kilobájt lett, így a fele 20 kilobájt), és a Slow Start algoritmust újraindítjuk. 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, akkor 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.

Abban az esetben, ha csomagvesztés történik, jó eséllyel előfordulhat, hogy duplikált nyugták keletkeznek. Ennek oka, hogy a TCP abban az esetben, ha valamely csomag nem a megfelelő sorrendben érkezik meg, duplikált nyugtát küld a forrásnak. Ebben az esetben a TCP Tahoe és TCP Reno viselkedése eltérő. A TCP Tahoe 3 duplikált nyugtát (4 azonos nyugta) ahhoz hasonlóan kezel, mintha időtúllépés következett volna be, tehát az elveszettnek ítélt csomagot az időzítő lejárta előtt újraküldi, majd a torlódási ablak méretét 1 MSS értékűre állítja be, és Slow Start fázisba lép. Az alkalmazott módszer a *Fast Retransmit* nevet viseli. A TCP Reno 3 duplikált nyugta esetén az elveszettnek ítélt csomagot szintén újraküldi az időzítő lejárta előtt, és a torlódási küszöböt az aktuális torlódási ablak értékének felére állítja be, de a torlódási ablakot az így megkapott torlódási küszöb értékére állítja be. A forrás ebben az esetben azért nem indul újra a Slow Start fázissal, mert a dup-

likált nyugták egyben azt is jelzik, hogy a vevőhöz továbbra is érkeznek csomagok, hiszen azok hatására generálja a vevő a duplikált nyugtákat. Az átviteli sebesség teljes lecsökkentése ezért indokolatlan lenne. A fent leírt utóbbi módszert *Fast Recovery* algoritmusnak nevezzük, amely a TCP Reno részeként jelent meg. A továbbiakban a hagyományos TCP, és a jelzők nélküli TCP alatt a TCP Renot fogom érteni.

2.3. High Speed TCP

A High Speed TCP a hagyományos TCP torlódásszabályozási algoritmusának egy olyan módosítása, amely lehetővé teszi nagyméretű torlódási ablakok esetén a hatékonyabb működést. Az algoritmust az RFC 3649 [4] írja le. Ebben az alfejezetben először röviden bemutatom, hogy mely problémák miatt volt szükség ezen új TCP verzió kidolgázásra, és mely követelményeket támasztották felé. Ez után tárgyalom a fontosabb módosításokat, és bemutatom, hogy ezeknek segítségével hogyan képes a HSTCP hatékonyabb működést biztosítani.

Állandósult állapotbeli viselkedés esetén, ha a csomagvesztési valószínűség p, akkor a hagyományos TCP esetén a torlódási ablak átlagos méretét szegmensekben számítva jó közelítéssel a 2.6. képlet adja meg [4]:

$$cwnd = \frac{1.2}{\sqrt{p}} \tag{2.6}$$

Ez jelentős korlátozásokat jelent valós hálózati környezetek esetén a torlódási ablakok esetére. Például 1500 bájt méretű csomagok, és 100 ms átlagos körülfordulási idővel rendelkező, hagyományos TCP algoritmust alkalmazó környezet esetén ahhoz, hogy elérjük egy 10 Gbps sebességű kapcsolat rendelkezésre álló állandósult állapotbeli átbocsátóképességét, átlagosan 83333 szegmensméretű torlódási ablakra van szükség, amely mellett minden $5 \cdot 10^9$ csomagból csak egyet veszíthetünk, ami megközelítőleg 100 perc alatt egyetlen csomagvesztést jelent. Ez átlagosan legfeljebb $2 \cdot 10^{-10}$ csomagvesztési valószínűséget jelent, amely az adott környezet esetén $2 \cdot 10^{-14}$ bithiba valószínűséggel jár. Ez egy olyan kis érték, amelyet még napjainkban sem lehetséges teljesíteni.

A HSTCP esetén a probléma megoldása a hagyományos TCP algoritmusa esetén használt paraméterek minimális módosítása. A HSTCP esetén a következő követelményeket támasztották:

- Magas átbocsátóképesség elérése anélkül, hogy nagyon alacsony csomageldobási valószínűségek garantálására volna szükség.
- Minél gyorsabban magas átbocsátóképesség elérése, különösen Slow Start fázis esetén.
- Magas átbocsátóképesség nagy késleltetések esetén akkor is, ha többszörös időtúllépés lépett fel, vagy egy kis torlódási ablakok által jellemzett periódus után.
- A routerek támogatására, és közreműködésére ne legyen szükség.
- A TCP vevők további visszajelzésére ne legyen szükség.

- TCP kompatibilitás közepes, vagy nagymértékű torlódás esetén.
- A teljesítmény legalább olyan jó legyen, mint amit a hagyományos TCP képes elérni közepes, vagy nagymértékű torlódás esetén.
- Elfogadható tranziens viselkedés a torlódási ablak változtatását, és a fair működés konvergencia idejét tekintve.

A hagyományos TCP esetén használt válaszfüggvényt úgy módosították a HSTCP esetén, hogy a következő három paramétert határozták meg:

- Low_Window: Annak érdekében, hogy biztosítsuk a TCP kompatibilitást, amikor a torlódási ablak nem haladja meg ezt az értéket, akkor a hagyományos TCP esetén használt válaszfüggvényt alkalmazzuk. Ha pedig meghaladjuk ezt az értéket, akkor a HSTCP módosított válaszfüggvényét használjuk.
- High_Window: Ez a paraméter meghatározza az elérni kívánt átlagos torlódási ablak mértékét.
- High_P: Az előző paraméter által meghatározott méretű átlagos torlódás ablak mellett High_P határozza meg a csomagvesztési valószínűséget.

Például a fenti 10 Gbps átbocsátóképesség eléréséhez a következő értékeket használhatjuk a paraméterek esetén.

- Low_Window: 38 MSS méretű szegmensre állítjuk be, amely 10^{-3} csomagvesztési valószínűséget jelent a TCP esetén.
- High_Window: 83000 értékűre állítjuk be, amely lehetővé teszi, hogy átlagosan 83000 szegmensméretű torlódási ablakot érjünk el.
- High_P: 10^{-7} értékűre állítjuk be. Ez a csomagvesztési valószínűség lehetővé teszi a HSTCP kapcsolat számára, hogy elérje a megfelelő átlagos torlódási ablakméretet.

A fentiek alapján a 2.7. képlet szerinti válaszfüggvényt használjuk abban az esetben, ha az átlagos torlódási ablak nagyobb, mint Low_Window:

$$W = \left(\frac{p}{Low\_P}\right)^{S} \cdot Low\_Window \tag{2.7}$$

Itt p jelenti az aktuális vesztési arányt, Low_P pedig az eredeti válaszfüggvény szerinti vesztési arányt. S értékét a 2.8. egyenlet alapján határozhatjuk meg [13]:

$$S = \frac{\log\left(High\_Window\right) - \log\left(Low\_Window\right)}{\log\left(High\_P\right) - \log\left(Low\_P\right)} \tag{2.8}$$

Itt minden esetben 10 a logaritmusok alapja, és az előzőleg tárgyalt értékeket használjuk a paraméterek esetén, így a 2.9. képlet szerinti válaszfüggvényt kapjuk meg:

$$W = \frac{0.12}{p^{0.385}} \tag{2.9}$$

Ez a válaszfüggvény tehát lehetővé teszi, hogy a fenti paramétereknek megfelelő működést elérjük.

A továbbiakban az előzőleg tárgyalt HSTCP válaszfüggvény alapján a torlódásszabályozás esetén alkalmazott paraméterek meghatározását ismertetem. Két fontos paramétert használunk, az egyik az *a(w)* növelési, a másik a *b(w)* csökkentési paraméter. Csomagvesztések esetén a csökkentési paraméter alapján a torlódási ablakot $w \cdot (1 - b(w))$ szegmensméretűre állítjuk be. Ha pedig nincsen torlódás, akkor a növelési paraméter határozza meg, hogy egy RTT alatt hány szegmensmérettel növekszik a torlódási ablak mérete. Tehát ez azt jelenti, hogy egy beérkezett nyugta esetén a 2.10. képlet szerint növekszik a torlódási ablak mérete:

$$w = w + \frac{a(w)}{w} \tag{2.10}$$

A hagyományos TCP w értékétől függetlenül az $a(w) = 1$, és a $b(w) = 1/2$ értéket használja. A HSTCP is ugyanezeket az értékeket használja abban az esetben, ha w nem nagyobb a *Low_ Window* paraméter értékénél. Ha w értéke megegyezik a *High_ Window* paraméter értékével, akkor pedig High_P csomagvesztési rátát állítottunk be, így a 2.11. képlet szerinti viszonynak kell fennállnia *a(w)* és *b(w)* értéke között $w = High\_Window$ esetén [13]:

$$a(w) = High\_Window^2 \cdot High\_P \cdot 2 \cdot \frac{b(w)}{2 - b(w)} \tag{2.11}$$

A csökkentési paraméter értékét $w = High\_Window$ esetén egy *High_ Decrease* elnevezésű paraméter határozza meg a fenti egyenletben, így az egyenlet megoldásával megkaphatjuk a növelési paraméter értékét is $w = High\_Window$ esetére. Tehát az előzőek alapján a $High\_P = 10^{-7}$, és a $High\_Window = 83000$ értékeket ismerjük. A $High\_Descrease = 0.1$ értéket alkalmazzuk, amely egy általánosan elfogadott érték, és amely így meghatározza a $b(83000) = 0.1$ értéket. Ekkor az egyenletből adódik, hogy $a(83000) = 72$. Ez azt jelenti, hogy torlódás esetén a csökkentés mértéke 10%. Ellenkező esetben pedig 72 szegmensmérettel növekszik a torlódási ablak egy RTT alatt. Ezek után a csökkentési paraméter értéke adott (1/2 értékű) $w = Low\_Window$, és az előbb meghatároztuk $w = High\_Window$ esetére is. A továbbiakban meghatározzuk a többi, $w > Low\_Window$ értékek esetére is, a 2.12. képlet szerint [13]:

$$b(w) = \frac{(High_{Decrease} - 0.5) \cdot (\log(w) - \log(W))}{\log(W_1) - \log(W)} + 0.5 \tag{2.12}$$

Itt $W = Low\_Window$, és $W_1 = High\_Window$. A növelési paramétert ekkor a következő képlet alapján határozhatjuk meg:

$$a(w) = w^2 \cdot p(w) \cdot 2 \cdot \frac{b(w)}{2 - b(w)} \tag{2.13}$$

Itt *p(w)* a csomagvesztési arányt jelenti w méretű torlódási ablak esetén, és értéke a 2.9.

képlet alapján a 2.14. képlet szerint számítható:

$$p(w) = \frac{0.078}{w^{1.2}} \tag{2.14}$$

Az így meghatározott értékeket már felhasználhatjuk a torlódási ablak méretének szabályozására. A növelés a 2.10. képlet alapján, a csökkentés pedig szintén az előző részben leírtak szerint történik. Látható, hogy ezen algoritmus szintén AIMD torlódásszabályozást valósít meg, de egy sokkal skálázhatóbb megoldást kaptunk, mint a hagyományos TCP esetén. Ez lehetővé teszi a HSTCP esetén a nagy kapacitású linkek jobb kihasználását.

2.4. BIC TCP

Ebben a részben a BIC TCP [6] esetén alkalmazott algoritmust ismertetem. Ez a TCP verzió a Linux kernel 2.6.8. verziójától kezdve a 2.6.18. verzióig alapértelmezett. A HSTCP esetén láthattuk azt, hogy képes jobb skálázhatóságot biztosítani, miközben a TCP kompatibilitás is fontos szempont volt a kifejlesztésénél. Ezeken kívül fontos tulajdonság az RTT igazságosság is, amelyet viszont nem képesek biztosítani az eddigi algoritmusok. Az RTT igazságtalanság (illetve igazságosság) arra utal, hogy az egymással versengő, különböző RTT értékkel rendelkező folyamok a sávszélességen igazságtalanul osztoznak meg. Az eddigi TCP verziók esetén fennáll ez a probléma, amelynek az oka, hogy torlódási ablak növelésének sebessége az ablak méretével nő. Éppen ez a tulajdonság az, amely a skálázhatóságot biztosította. A kisebb RTT értékkel rendelkező folyam gyakrabban növeli az ablakméretét, így jóval nagyobb sávszélességre tehet szert, mint a nagyobb RTT értékű folyam. A HSTCP és STCP (Scalable TCP) esetén például mind-mind jelentkezik ez a probléma. A következő példák esetén két folyam esetén láthatjuk a jelentkező RTT igazságtalanságot. Itt a két folyam átbocsátóképességének arányát a két RTT érték hányadosával fejeztük ki, és ez a hányados határozza meg az RTT igazságtalanság mértékét. Ha a két egymással versengő folyam létezik, amelyeknek egy adott sávszélességen kell osztozniuk, akkor a hagyományos TCP AIMD algoritmusa esetén az RTT igazságtalanság a 2.15. képlet szerint alakul:

$$\left(\frac{RTT_2}{RTT_1}\right)^2 \tag{2.15}$$

HSTCP alkalmazása esetén a pedig a 2.16. képlet szerint:

$$\left(\frac{RTT_2}{RTT_1}\right)^{5.56} \tag{2.16}$$

Végül, a Scalable TCP esetén ezt az arányt a 2.17. egyenlet adja meg:

$$\left(\frac{RTT_2}{RTT_1}\right)^{\infty} \tag{2.17}$$

Láthatjuk, hogy a legutóbbi esetben a kisebb RTT értékű folyam már teljesen kiszorítja

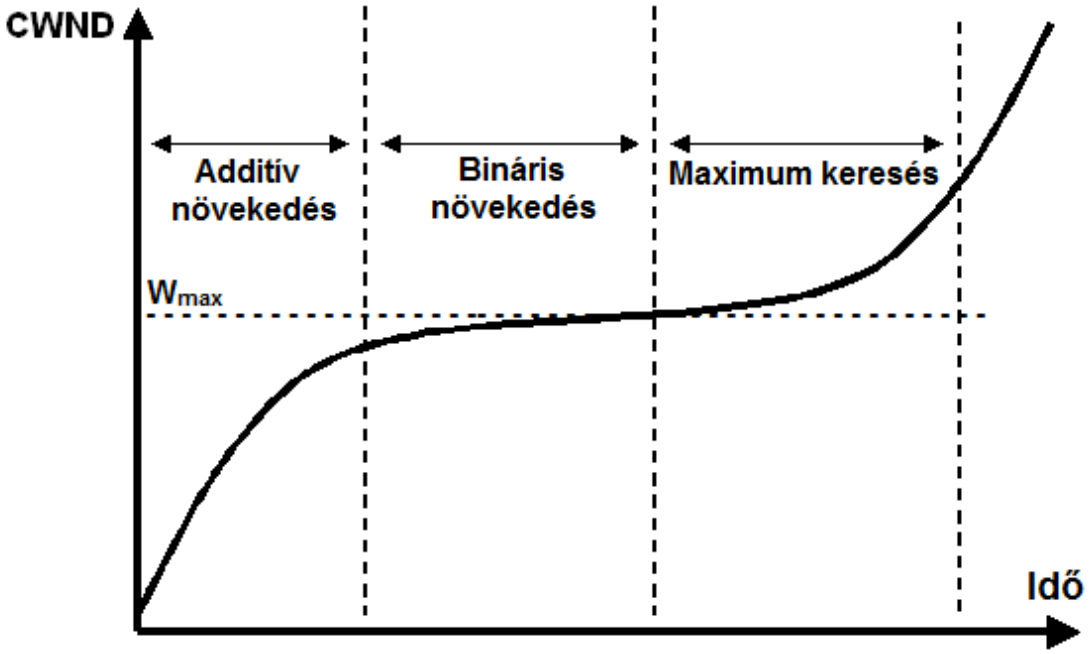
a nagyobb RTT értékkel rendelkezőt.

Az előbb tárgyalt RTT igazságtalanság a nagysebességű hálózatok esetén főként a *Drop-tail* routerek alkalmazásakor jelent problémát, és általában a nagy torlódási ablakkal rendelkező, egymással versengő folyamok esetén lehet jelentős. A Drop-tail eldobási stratégiát alkalmazó routerek abban az esetben, ha a tárolójuk megtelt, akkor az újonnan beérkező csomagokat dobják el.

A BIC TCP kidolgozásakor tehát a célok a következők voltak:

- Stabilitás: A BIC TCP képes kihasználni 10 Gbps sávszélességet $3.5 \cdot 10^{-8}$ csomagvesztési arány mellett. (hasonlóan a HSTCP-hez, amelynél ez az arány 10^{-7} volt)
- RTT igazságosság: Nagy ablakméretek esetén, és AIMD torlódásszabályozás esetén is, az RTT igazságtalanság arányos az RTT értékek hányadosával.
- TCP kompatibilis viselkedés, minden ablakméret esetén. Jelentős mértékű csomagvesztés esetén a kompatibilitás az STCP változatéhoz hasonló.
- Fair viselkedés és konvergencia: A HSTCP, és STCP algoritmusokhoz képest a BIC algoritmusa korrektebb módon osztja meg a sávszélességet, mind rövid, mind hosszútávon és ezt az igazságos állapotot gyorsabban képes elérni.

Ebben a részben a BIC TCP torlódásszabályozó algoritmusa által használt függvényt mutatom be. A protokoll (amelynek neve is erre utal) által használt algoritmus két részből tevődik össze, az első fázis a *bináris keresésű ablakméret növelés* (binary search increase), a második pedig az *additív ablakméret növelés* (additive increase). Az egyes fázisokat a 2.2. ábrán láthatjuk, amely megmutatja a torlódási ablak, *cwnd* változását az eltelt idő függvényében.



2.2. ábra. A BIC TCP torlódásszabályozása

2.4.1. Bináris keresésű ablakméret növelés

Ebben a fázisban a torlódásszabályozást egy olyan keresési problémaként tekintjük, ahol a rendszerből kapott visszacsatolás alapján (történt-e csomagvesztés, vagy sem) eldönthetjük azt, hogy a jelenlegi küldési sebesség meghaladja-e a hálózat kapacitását. Az algoritmus ehhez a kereséshez három paramétert használ. A keresés kezdőpontja a jelenlegi minimális ablakméret, W_{min}, és a maximális ablakméret W_{max}. Általában W_{max} a legutóbbi csomagvesztés előtti ablakméret, W_{min} pedig a legutóbbi csomagvesztés utáni ablakméret. Az algoritmus folyamatosan nyilvántartja ezen két érték közötti távolság felét, a *megcélzott ablakméretet*, amely így a két érték felezőpontjánál található meg. Az algoritmus a jelenlegi ablakméretet az így megkapott értékűre állítja be, ez után pedig a csomagvesztésre utaló visszajelzések alapján hoz döntést. Ha csomagvesztés történt, akkor az új W_{max} értéket a felezőpontra állítjuk, ha nem történt csomagvesztés, akkor pedig az új W_{min} értékét állítjuk be a felezőpontra. Ez a folyamat addig ismétlődik, amíg a W_{max} és a W_{min} értékek közötti különbség egy adott S_{min} paraméternél kisebb nem lesz. Ez az eljárás sokkal agresszívabb viselkedésű, amikor a cél ablakmérettől távol van a jelenlegi ablakméret, de ahogy ahhoz közeledünk, egyre kevésbé lesz az. Az algoritmus egyediségét az adja, hogy az ablaknövelő függvénye logaritmikus. Így a növekedés mértéke egyre csökken, ahogy az ablakméret megközelíti a hálózat telítettségét jelentő értékét. Más, skálázható protokollok a telítettségi ponton úgy növelik a küldési sebességet, hogy ekkor a legmagasabb a növekmény. Általában a csomagvesztések száma arányos a csomagvesztés előtti növekménnyel, ezért a BIC esetén alkalmazott eljárás képes a csomagvesztések csökkentésére. Ezen kívül fontos, hogy a bináris keresés legfontosabb előnyét az jelenti, hogy alkalmazásával egy konkáv válaszfüggvényt kapunk, amely jól kiegészíti a következő részben tárgyalt additív ablakméret növelést.

2.4.2. Additív ablakméret növelés

Annak érdekében, hogy biztosítsuk a gyorsabb konvergenciát, és az RTT igazságosságot, a bináris keresésű ablakméret növelést egy additív ablakméret növelési stratégiával kombináljuk. Amikor a jelenlegi minimum és a felezőpont közti különbség túl nagy, akkor az ablakméret hirtelen növelése a felezőpontig túl nagy terhelést jelenthet a hálózatra nézve. Amikor a bináris keresésű ablakméret növelés során a különbség a jelenlegi ablakméret és a cél ablakméret között nagyobb, mint egy előre meghatározott maximális érték, S_{max}, akkor ahelyett, hogy a következő RTT során azonnal a felezőpontra állítanánk az ablakméretet, inkább S_{max} értékkel növeljük azt, amíg a különbség kisebb nem lesz, mint S_{max}, amikor végül az ablakméretet a felezőpontra állíthatjuk be. Így abban az esetben, ha előzőleg egy nagyobb mértékű ablak csökkentés történt, akkor ez a stratégia kezdetben lineárisan növeli az ablakot, majd azután logaritmikusan. A bináris keresésű, és additív ablakméret növelés ezen kombinációját bináris (ablakméret) növelésnek nevezzük. Multiplikatív csökkentési stratégiával együtt a bináris növelés nagy ablakméretek esetén megközelíti a tisztán additív növelést. Ennek az oka, hogy egy nagyobb ablakméretet a multiplikatív csökkentés jobban lecsökkent, és ez hosszabb additív növelési periódust eredményez. Kis ablakmére-

tek esetén pedig a tisztán bináris keresésű ablakméret növeléshez közelít, amely rövidebb additív növelési periódust jelent.

2.4.3. Slow Start

Miután az aktuális ablakméret elérte a jelenlegi maximum, W_{max} értékét, az új maximális értéket nem ismerjük. Ekkor a bináris keresés során használt maximum egy nagy konstans által meghatározott értéket fog felvenni, jelenlegi ablakméret pedig a minimum lesz, tehát a cél felezőpontig nagyon nagy különbség lehet. Ekkor a bináris növelés szerint, amikor a különbség nagy, akkor S_{max} értékű lineáris növelés kellene, hogy következzen. Ezzel szemben most egy Slow Start stratégiát alkalmazunk, hogy meghatározzuk az új maximumot. Abban az esetben ha *cwnd* értékű a jelenlegi torlódási ablak, akkor azt minden RTT esetén a következőképpen növeljük: $cwnd{+}1, cwnd{+}2, cwnd{+}4, \ldots, cwnd{+}S_{max}$. Ez azért fontos, mert ezen a ponton a hálózat teljesítőképességét jelentő értékhez közel állhatunk, és nem ismerjük az új maximumot, így a rendelkezésre álló sávszélességet Slow Start segítségével próbáljuk meghatározni, amíg biztonságossá nem válik az ablakméret S_{max} értékkel való növelése. A Slow Start után újra bináris növelést alkalmazunk.

2.4.4. Gyors konvergencia

Megmutatható, hogy teljesen szinkronizált vesztési modell esetén a multiplikatív csökkentéssel kombinált bináris keresésű növelés igazságos sávszélesség megosztást eredményez. Feltételezzünk két olyan folyamatot, amelyek esetén az ablakméret eltérő, de az RTT érték azonos. Mivel a nagyobb ablakméretet nagyobb mértékben csökkentjük a multiplikatív tulajdonság miatt, ezért ekkor több idő szükséges ahhoz, hogy a nagyobb ablak elérje a célértékét. A bináris keresésű ablakméret növelés esetén $\log(d) - \log(S_{min})$ RTT szükséges ahhoz, hogy elérjük a maximális ablakméretet egy d mértékű ablakméret csökkentés után. Mivel az ablakméret logaritmikus lépésekben növekszik, ezért a kisebb, és a nagyobb ablak közel egyszerre éri el a maximumát (a kisebb ablak kicsit hamarabb éri el azt), ezért a kisebb ablakkal rendelkező folyam csak kevés sávszélességet képes elvenni a nagyobb folyamtól, mielőtt a következő ablakcsökkentés bekövetkezne. Azért, hogy orvosolják ezt a jelenséget, módosították a bináris keresésű ablakméret növelést a következő módon. Ablakcsökkentés után új maximumot, és új minimumot használunk, ezek az értékek legyenek max_win_i, és min_win_i az i. folyam esetén. Ha az új maximum kisebb, mint az előző, akkor az ablak csökkenő trendet mutat, tehát valószínűleg nagyobb, mint amekkora az igazságos megosztás esetén indokolt lenne. Ezért az új maximumot a cél ablakméretre állítjuk, tehát $max\_win_i = (max\_win_i - min\_win_i)/2$, majd pedig frissítjük a célértéket is. Ez után pedig bináris növelést alkalmazzuk. A leírt algoritmust *gyors konvergenciának* nevezzük, és a nagyobb ablakméretű folyam esetén az ablak növelésének lassításával lehetővé teszi azt, hogy a kisebb ablakméretű folyam felzárkózhasson.

Összegezve, a BIC TCP jó teljesítménye az ablakméret W_{max} érték körüli óvatos növeléséből, és az additív ablakméret növelés, valamint a maximum keresés során alkalmazott lineáris növelésnek köszönhető. Ebből következik, hogy az algoritmus képes a TCP kom-

patibilis működésre. Ezen kívül stabilis viselkedésű, és képes a hálózati erőforrások jobb kihasználására, mint más verziók. A lineáris növelés ezenkívül a BIC TCP fair működését is fejleszti, és lehetővé teszi az egymással versengő BIC folyamok között a sávszélesség igazságosabb megosztását.

2.5. CUBIC TCP

Az utolsó nagysebességű TCP változat, amelyet bővebben bemutatok, a CUBIC TCP [7], amely a BIC TCP továbbfejlesztett változata, és a Linux kernel 2.6.19 verziójától kezdve alapértelmezett. A CUBIC egyszerűsíti a BIC ablakvezérlését, javít a TCP kompatibilitáson, RTT igazságosságon, és eközben megtartja a BIC jó tulajdonságait, stabilitást, és skálázhatóságot biztosít. Az ablaknövelő függvényét egy köbös függvény adja meg a legutóbbi csomagvesztés óta eltelt idő függvényében, tehát a működés a valós időskálához kötött. Az algoritmus valós idejű természete lehetővé teszi, hogy az RTT értékétől független legyen az ablaknövelés mértéke, amely TCP kompatibilitást tesz lehetővé nagy, és kis RTT értékek esetén is, valamint RTT igazságosságot biztosít, mivel az RTT értékétől függetlenül történik az ablaknövelés. A következőkben bemutatom a CUBIC által használt ablaknövelő függvényt, és ismertetem annak tulajdonságait.

A BIC által alkalmazott függvény túl agresszív a TCP számára, főleg kis sebességű, vagy kis RTT értékkel rendelkező hálózatok esetén. Ezenkívül a BIC ablakvezérlő mechanizmusa által alkalmazott fázisok nagyobb komplexitást jelentenek a protokoll implementálása, és teljesítményének vizsgálata során. A CUBIC által alkalmazott függvény egyszerűsíti, és továbbfejleszti a BIC által alkalmazott eljárást. A függvényt a 2.18. képlettel írhatjuk le:

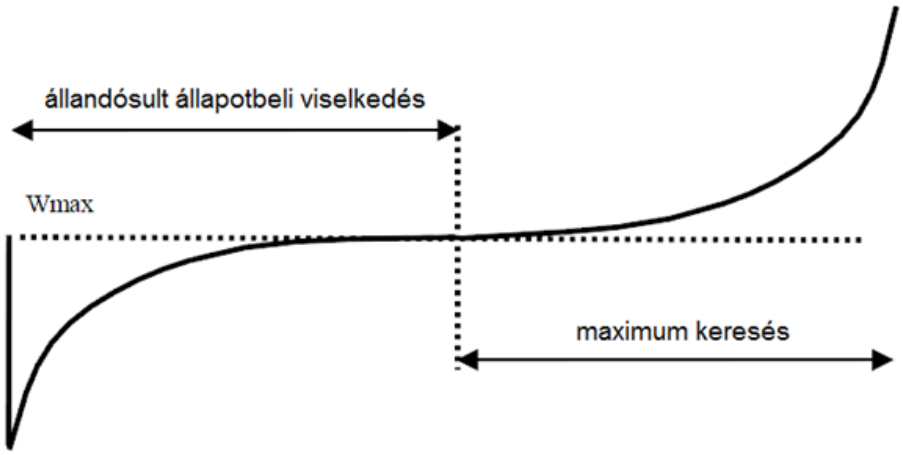
$$W_{cubic} = C \cdot (t - K)^3 + W_{max} \tag{2.18}$$

Itt C egy skálafaktor, t a legutóbbi csomagvesztés által okozott ablakcsökkentés óta eltelt idő, W_{max} pedig az ablakcsökkentés előtti ablakméret. Az egyenletben látható K paraméter értékét a 2.19. összefüggéssel határozhatjuk meg:

$$K = \sqrt[3]{W_{max} \cdot \frac{\beta}{C}} \tag{2.19}$$

Az összefüggés esetén β egy konstans multiplikatív csökkentési faktor, amelyet az ablak csökkentésére alkalmazunk, amikor csomagvesztés történik. Tehát az ablak mérete $W_{max} \cdot \beta$ értékre csökkent a legutóbbi csökkentés idején.

A 2.3. ábrán láthatjuk az ablaknövelő függvényt. A legtöbb TCP verzió konvex ablaknövelő függvényt alkalmaz, amikor egy csomagvesztés után az ablaknövelés mértéke egy állandóan növekvő mennyiség. Ezzel szemben a CUBIC az alkalmazott köbös függvény konkáv és konvex részeit is felhasználja az ablak növelésére.



2.3. ábra. A CUBIC TCP ablaknövelő függvénye

Egy csomagvesztés miatt bekövetkező ablakcsökkentés esetén a vesztés idején használt ablakméret lesz W_{max} értéke, és a fent leírtak szerint a torlódási ablak multiplikatív csökkentése történik. Majd a TCP esetén ismertetett Fast Retransmit, és Fast Recovery fázisokon keresztül torlódáselkerülési fázisba kerülünk. Az ablakméretet a 2.3. ábrán látható függvény konkáv részének megfelelően növeljük. Láthatjuk, hogy az ablakméret kezdetben nagyon gyorsan növekszik, de ahogy a W_{max} értékhez közelítünk egyre lassul ez a növekedés. Ennél az értéknél majdnem 0 az ablaknövelés mértéke. Az *állandósult állapotbeli viselkedés* esetén az ablakméret legtöbbször a W_{max} érték körül mozog. A W_{max} érték után azonban a *maximum keresés* során CUBIC megpróbálja kiaknázni a további, rendelkezésre álló sávszélességet. Ekkor kezdetben lassan történik az ablak növelése, de a W_{max} értéktől távolodva egyre gyorsabbá válik. A W_{max} érték közelében tapasztalható lassulásnak, és a W_{max} értéktől való távolodás során történő gyorsulásnak köszönhető a protokoll stabilis, és igazságos működése, és a hálózati erőforrások jobb kihasználása. Ezen kívül ez a viselkedés jó skálázhatóságot képes biztosítani nagy BDP értékű, és nagy sávszélességű hálózatok esetén.

Az igazságos működés, és stabilitás további fejlesztése céljából azt az értéket, amivel az ablakot növelnénk korlátozzuk, és másodpercenként maximum S_{max} mértékű növekedést engedünk meg. Emiatt abban az esetben, ha az ablak a W_{max} értéktől távol esik, akkor lineárisan történik a növelés, hasonlóan, mint a BIC algoritmusa esetén, ahol abban az esetben, ha az ablak növelése esetén a növekmény egy előre meghatározott konstansnál nagyobb, szintén additív ablaknövelés történik.

Az ablak valós időskálához kötött növelése nagymértékben képes javítani a protokoll TCP kompatibilitását. Amíg más, RTT értéktől függő protokollok ablaknövelő függvénye gyorsabban növekszik a kisebb RTT értékű hálózatok esetén, addig a CUBIC esetén a növelés RTT értéktől független módon történik. Ez a TCP esetén egyre agresszívabb működést jelent, ezzel szemben a CUBIC esetén változatlan működés történik. A kis RTT értékű hálózatok esetén a CUBIC által alkalmazott algoritmus így sokkal TCP kompatibilisebb, hiszen lassabban történik az ablak növelése, mint a TCP esetén. A *TCP mód* segítségével

biztosítható, hogy ennek ellenére mégis ugyanolyan mértékben történjen az ablak növelése.

Egy nyugta beérkezésekor torlódáselkerülés esetén az algoritmus meghatározza az ablaknövelés sebességét a következő RTT periódusra nézve a 2.18. képlet szerint. A torlódási ablak megcélzott méretét $W(t+RTT)$ értékre állítja be. A jelenlegi torlódási ablak legyen *cwnd* értékű. Ekkor a CUBIC ettől az értéktől függően három különböző módban képes működni. Ha *cwnd* kisebb, mint amit a TCP elérne a legutóbbi csomagvesztés után t idővel, akkor a később ismertetett TCP módot használja. Egyébként, ha a *cwnd* kisebb, mint W_{max}, akkor a függvény konkáv részét, ha pedig nagyobb, mint W_{max}, akkor pedig annak konvex részét használjuk.

Abban az esetben, ha egy csomagvesztés után TCP módba kerülünk, akkor emuláljuk a TCP algoritmusát. Mivel a CUBIC a β multiplikatív faktor által meghatározott módon csökkenti az ablakot egy csomagvesztés után, a TCP fair additív növekmény RTT-nként $3 \cdot \beta/(2-\beta)$. Ennek az oka, hogy AIMD algoritmus esetén az átlagos ablakméret a 2.20. képlet szerint alakul:

$$\frac{1}{RTT} \cdot \sqrt{\frac{\alpha}{2} \cdot \frac{2-\beta}{\beta} \cdot \frac{1}{p}} \tag{2.20}$$

Itt α az additív ablaknövekmény, p a csomagvesztési valószínűség. A TCP esetén $\alpha = 1$, és $\beta = 1/2$ értékű, tehát a megfelelő értéket a 2.21. képlettel számíthatjuk ki:

$$\frac{1}{RTT} \cdot \sqrt{\frac{3}{2} \cdot \frac{1}{p}} \tag{2.21}$$

A fenti két egyenlet alapján kifejezhető, hogy tetszőleges β érték esetén α meg kell egyezzen a $3 \cdot \beta/(2-\beta)$ értékkel. Ha a TCP α értékkel növeli egy RTT alatt az ablakméretet, akkor az eltelt idő függvényében a 2.22. képlet alapján kaphatjuk meg a TCP ablakméretét:

$$W_{tcp}(t) = W_{max} \cdot (1-\beta) + 3 \cdot \frac{\beta}{2-\beta} \cdot \frac{t}{RTT} \tag{2.22}$$

Ha az aktuális *cwnd* kisebb, mint $W_{tcp}(t)$, akkor a TCP módnak megfelelően $W_{tcp}(t)$ értékűre állítjuk be a *cwnd* értékét minden egyes beérkező nyugta esetén.

Az eddig leírtakból látható, hogy a CUBIC esetén fontos szempont volt a hagyományos TCP folyamokkal való igazságosság, és együttműködés, emellett a CUBIC algoritmusa jó stabilitást, és skálázhatóságot nyújt, és képes jól kihasználni a hálózati erőforrásokat nagy BDP érték esetén is, emellett ez a megoldás egyszerűsíti a BIC algoritmusát.

2.6. Egyéb TCP változatok

A Scalable TCP [5] egy olyan verzió, amely szintén a nagysebességű hálózati környezetek esetén javítja a teljesítményt. Itt a torlódási ablak méretének szabályozása nem AIMD, hanem *MIMD* (Multiplicative Increase Multiplicative Decrease) mechanizmus alapján történik. Ez a módszer azért előnyös, mert a TCP Reno esetén drasztikus csökkentésnek bizonyult a torlódási ablak felezése torlódás esetén. Hátránya viszont, hogy az algorit-

mus agresszív, nem képes az igazságos együttműködésre, kiéhezteti a hagyományos TCP folyamokat.

A késleltetés alapú szabályozás először a TCP Vegas [14] protokollban jelent meg. Ennek a nagysebességű környezethez továbbfejlesztett, módosított változata a FAST TCP [8]. Az alkalmazott algoritmus ebben az esetben tehát késleltetést használja fel a torlódásszabályozáshoz, amely egy folytonos mennyiség, így több információt képes nyújtani a hálózatról, mint amennyit a csomagvesztésből származó információ jelentene. A protokoll előnyei közé tartozik, hogy képes jó kihasználtságot elérni, és kedvező tulajdonságokkal rendelkezik az együttműködés szempontjából, illetve csak a küldő oldalon igényel módosításokat. Hátrányai közé tartozik, hogy nehézségeket okoz az algoritmus paramétereinek megfelelő beállítása.

A TCP Hybla [15] célja, hogy nagy RTT értékű (tipikusan műholdas, vagy egyéb nagy távolságú) hálózatok, kapcsolatok esetén javítsa a hagyományos TCP teljesítményét. A nagy RTT értékkel rendelkező kapcsolatok esetén olyan küldési sebességet próbál meg elérni, mint amilyet egy, az adott tulajdonságokkal rendelkező, viszonylag gyors (vezetékes) referencia TCP kapcsolat elérne. Ez a TCP verzió a Linux kernelben is elérhető a 2.6.13 verziótól kezdődően. Előny, hogy szintén csak a küldő oldalon igényel változtatásokat, és teljesen kompatibilis a hagyományos TCP algoritmust alkalmazó vevőkkel.

A TCP Illinois [16] főleg a nagy BDP értékkel rendelkező hálózatok esetére kifejlesztett hibrid TCP változat. Elsődleges információként a csomagvesztést használja, hogy meghatározza az ablakméret változtatásának irányát. Emellett a sorbanállási késleltetést is felhasználja, így határozza meg az ablakméret változtatásának mértékét. A hagyományos TCP torlódásszabályozása esetén a felhasznált α és β paraméterek itt már nem konstansok, hanem az átlagos sorbanállási késleltetés függvényében változnak. Ennél a TCP változatnál előny, hogy képes jobb átlagos átbocsátóképességet elérni, mint a hagyományos TCP, a hálózati erőforrásokat igazságosan használja, TCP kompatibilis, és csak a küldő oldalon igényel változtatásokat.

A YeAH TCP (Yet Another High-speed TCP) [17] egy olyan hibrid változat, amely számos TCP verzió teszteléséből szerzett tapasztalatok alapján született meg. Két különböző működési módja van. *Gyors üzemmódban* egy agresszív algoritmus szerint növeli a torlódási ablakot, itt az STCP eljárását alkalmazzák. *Lassú üzemmódban* pedig a TCP Renohoz hasonlóan működik. Az alkalmazott üzemmódot a szűk hálózati keresztmetszet esetén sorbanálló csomagok becsült száma alapján határozza meg, ez a becslés pedig a mért RTT értékek alapján történik. A protokoll elkészítése során fontos szempont volt, hogy képes legyen jól kihasználni a hálózati erőforrásokat, ne terhelje jobban a hálózatot mint a Reno, legyen TCP kompatibilis, RTT igazságos, valamint a csomagvesztések, és a kis méretű hálózati bufferek ne rontsák indokolatlanul jelentősen a teljesítményt.

Végül, az utolsó változat, amelyet megemlítek, a Compound TCP [18], amely szintén hibrid algoritmust alkalmaz a torlódásszabályozásra. Ezt a TCP változatot a Windows Vista, és a Windows Server 2008 részeként mutatták be. Az algoritmus a küldő torlódási ablakának agresszívabb növelésével próbál a TCP algoritmusánál jobb hatékonyságot elérni a nagy BDP értékkel rendelkező hálózatok esetén úgy, hogy igazságos működésű maradjon.

Két torlódási ablakot alkalmaz. Az egyik a megszokott *AIMD ablak*, a másik pedig egy olyan ablak, amely a sorbanállási késleltetésen alapul. Az aktuálisan használt csúszóablak mérete a két ablak összege lesz. Az AIMD ablakot a hagyományos TCP-hez hasonlóan növeli. A másik ablak kis késleltetés esetén nagyon gyorsan növekszik, hogy a hálózat kihasználtságát növelje. Ha nagyobb lesz a késleltetés, akkor ez az ablak folyamatosan csökken, így az AIMD ablak növekedését kompenzálja. Gyakorlatilag a *késleltetési ablakot* a várakozási sor becsült hosszával csökkentik. A cél az, hogy megközelítőleg konstans értéken tartsák a két ablak összegét, ami az algoritmus által becsült BDP értéket jelenti az adott útvonalon.

3. fejezet

Torlódásszabályozás nélküli transzport protokoll

A torlódásszabályozást alkalmazó protokollok esetén látható, hogy nem létezik egy optimális működést megvalósító protokoll, amely a gyorsan fejlődő Internet követelményeinek meg tudna felelni. Ebben a részben egy olyan új elvet ismertetek, amely szerint egyáltalán ne alkalmazzunk torlódásszabályozást, hanem hatékony hibajavító kódolások segítségével történjen a csomagvesztések helyreállítása, a sávszélesség igazságos megosztását pedig igazságos ütemezés segítségével biztosíthatjuk. Itt bemutatok egy olyan protokollt, amely ezen az elven működik, tehát nem alkalmaz torlódásszabályozást, helyette hibajavító kódolással történik a csomagvesztések helyreállítása. A fejezetben először ismertetem a torlódásszabályozás nélküli koncepciót, amelynek során kitérek a TCP protokoll esetén tapasztalt problémákra, és megmutatom, hogy az új ötlet hogyan lehet képes egy alternatívát nyújtani a jelenleg TCP protokollon alapuló Internet számára. Ez után röviden tárgyalom a Linux kernel hálózati alrendszerének felépítését, majd bemutatom az új protokollt, amelynek ismertetem a működési fázisait a jelenlegi Linux kernelben megtalálható implementációja alapján, itt láthatjuk az alkalmazott kódolást, és dekódolást is. Ennek során számos ábrával, és azokon keresztül történő magyarázatokkal mutatom be a részleteket. Végül ismertetem a protokoll lehetséges paramétereit, amelyek segítségével nagymértékben lehetséges a protokoll működésének szabályozása.

3.1. Alapelv

A jelenlegi Internet fő szabályozó mechanizmusa a torlódásszabályozás, amely a hálózati erőforrások alacsony kihasználtságát próbálja csökkenteni. Az előző részben láthattuk, hogy a különböző környezetek esetén fellépő problémák megoldására számos különböző TCP verzió jelent meg, amelyek az adott környezetben képesek voltak jobb teljesítményt elérni, mint a hagyományos TCP. Az is látható volt, hogy ezek a protokollok nem mindig képesek az igazságos együttműködésre, és a különböző TCP változatok nagyon eltérő teljesítményt nyújtanak. Képesek néhány területen hatékonyabb működést elérni, de nem képesek egy univerzális és optimális megoldást nyújtani a torlódásszabályozásra a jelenlegi heterogén,

és állandóan változó hálózati környezetek esetén. Az egyik legfontosabb probléma a TCP változatok esetén az, hogy ezek saját torlódásszabályozó algoritmusokat alkalmaznak, és a legtöbb esetben látható, hogy a felhasznált mechanizmusok nem képesek hatékonyan együttműködni.

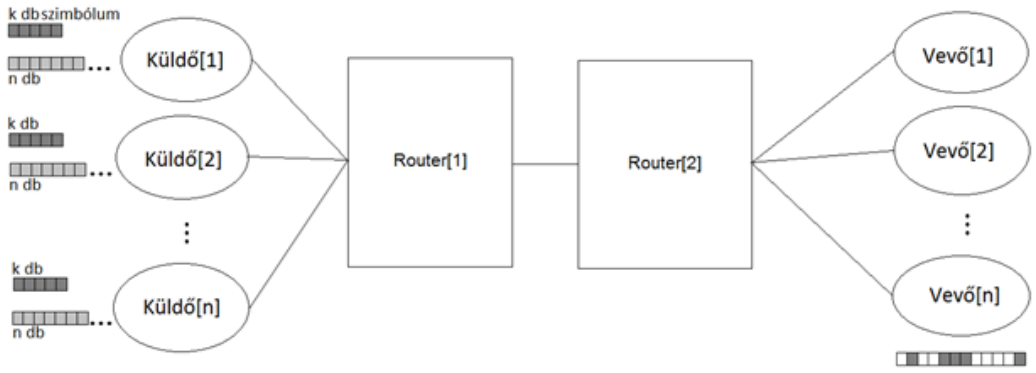
Annak érdekében, hogy megoldást találjunk ezekre a problémákra, jelenleg is számos kutatás történik. Az egyik ezzel kapcsolatos javaslat szerint, amely a GENI-től származik [1], a jövő Internetje esetén egyáltalán ne alkalmazzunk torlódásszabályozást. Az ötlet ígéretesnek tűnik, de eddig néhány, ezzel az elvvel kapcsolatos munkán kívül semmilyen realizációt, vagy további finomítást nem publikáltak. A következőkben röviden áttekintem az ehhez kapcsolódó munkákat. Raghavan és Snoeren tanulmányozta a torlódásszabályozás nélküli hálózat nyújtotta előnyöket, és bemutatott egy torlódásmentesítőt, mint egy lehetséges megoldás alapját [19]. Bonald és mások tanulmányozták a torlódásszabályozás nélküli hálózat viselkedését [20]. Az ő eredményük megmutatta, hogy nem igaz az elterjedt hiedelem, amely szerint a torlódásszabályozás nélkülözése a hálózat összeomlásához vezetne. López és mások a játékelméleten keresztül vizsgálták meg egy szökőkút kódoláson alapuló protokoll teljesítményét [21]. Megmutatták, hogy elérhető egy olyan Nash egyensúly, amelynél a hálózat teljesítménye hasonló, mintha minden végpont TCP-t alkalmazna. Úgy tűnik, hogy a ráta nélküli kódok jól alkalmazhatók hibajavító kódolásként. Például [22] bemutatja az élő videó közvetítés egy ráta nélküli kódokon alapuló forgatókönyvét, és tartalmazza a hozzá kapcsolódó kísérleti eredményeket is.

A továbbiakban azt tételezem fel, hogy egyáltalán nem alkalmazunk torlódásszabályozást. A felhasznált új javaslat szerint a hálózat minden végpontja maximális sebességgel küldhet adatokat, tehát amíg csak van rendelkezésre álló adat egy adott végpont esetén, addig olyan gyorsan történhet a küldés, amennyire csak lehetséges. Ha ez nem okoz torlódást, akkor ez a leghatékonyabb megoldás. Természetesen ha minden végpont maximális sebességgel küld adatokat, akkor nagymértékű, főleg csomós jellegű csomagvesztés keletkezhet az erőforrások túlterhelése miatt. A GENI javaslata alapján ezt a csomagvesztést hatékony hibajavító kódolás alkalmazásával ellensúlyozhatjuk. Az eddig leírt megoldásnak számos előnye van. Az egyik a hatékonysága, ugyanis ez a módszer minden hálózati erőforrást mindig teljesen kihasznál, és azonnal felhasználja a rendelkezésre álló új kapacitásokat is a hálózatban. A másik az egyszerűsége, ugyanis a csomagvesztések hatékony hibajavító kódolásokkal történő helyreállítása következtében a routerek esetén csökkenthető a bufferméret. Végül, fontos a módszer stabilitása is, mivel a maximális sebességű küldés alkalmazása sokkal előrejelezhetőbb forgalmat jelent. Ezzel szemben a TCP esetén látható volt, hogy a küldési sebesség nagymértékben ingadozhat, amely ezt megnehezíti. Az itt leírt előnyök különösen kedvezőek az optikai hálózatokra nézve, ahol csak kisméretű bufferek alkalmazására van lehetőség.

Az új koncepció esetén a legnagyobb kihívást az jelenti, hogy egy olyan mechanizmust adjunk, amely a működés során fellépő csomagvesztést úgy képes kijavítani, hogy közben skálázható kommunikációt tesz lehetővé. Az egyik lehetséges megközelítés a *ráta nélküli* kódolás alkalmazása. A hagyományos blokk kódolásokkal szemben, ahol egy k hosszúságú információt egy n hosszúságú kódszóba transzformálunk, a ráta nélküli, vagy más néven

szökőkút kódolás egy végtelen hosszúságú kódolt szimbólumokból álló folyamot állít elő az eredetileg k hosszú üzenetből. Ebből látható, hogy egy ráta nélküli kódoló rátája, amelyet a k/n értékkel fejezhetünk ki, a 0 értékhez tart, ha n a végtelenhez tart. Az univerzális ráta nélküli kódolások első gyakorlati megvalósításai a *Luby Transform* (röviden *LT*) [23] kódok voltak, amelyek rendelkeznek azzal az előnnyel, hogy megközelítőleg optimálisak minden törléses csatorna esetén, és nagyon hatékonyak, ahogy az adatmennyiség növekszik. A szökőkút kódolások következő realizációját a *Raptor kódok* jelentették [24]. A Raptor kódok az LT kódok olyan továbbfejlesztései, amelyek lineáris idejű kódolást, és dekódolást tesznek lehetővé. A Raptor kódolás alkalmazásával egy adott, k szimbólumból álló üzenet, és bármely valós $\varepsilon > 0$ paraméter esetén egy olyan végtelen hosszúságú, kódolt szimbólumokból álló folyamot állíthatunk elő, amelynek bármely $\lceil (1+\varepsilon) \cdot k \rceil$ méretű részéből nagy valószínűséggel visszaállíthatjuk az eredeti k szimbólumot. Ebből következik, hogy minden sikeresen átvitt szimbólum felhasználható a dekódolás során. Ebben az esetben a kódolás komplexitása $\mathcal{O}\big(\log(1/\varepsilon)\big)$, a dekódolás komplexitása pedig $\mathcal{O}\big(k \cdot \log(1/\varepsilon)\big)$. Előnyt jelent, hogy a kódolás, és dekódolás során is csak olyan egyszerű műveletek elvégzésére van szükség, mint a szimbólumok másolása, vagy a néhány szimbólumra alkalmazott *kizáró vagy* művelet.

A 3.1. ábra egy szökőkút kódolást alkalmazó hálózati elrendezést mutat be. A küldő folyamatok a küldendő adatokat kódolják Raptor kódolással, majd maximális sebességgel küldik. Ezt a sebességet csak két dolog korlátozhatja, amelyből az egyik a küldő alkalmazás, a másik pedig a link kapacitása.



3.1. ábra. Hálózati architektúra n küldő-vevő pár esetén

A Raptor kódolás kiválóan beleillik az új koncepcióba, mivel ez a kódolás lehetővé teszi, hogy a beérkező kódolt folyam bármely $\lceil (1+\varepsilon) \cdot k \rceil$ méretű részéből nagy valószínűséggel visszaállíthassuk az eredeti k szimbólumot. Ez a tulajdonság rendkívül hibatűrő adatátvitelt tesz lehetővé, még abban az esetben is, ha a csomagvesztés mértéke dinamikusan változik, és így nagymértékben csomósodik. Ha ekkor a dekódolás sikertelen lenne, akkor a vevő megpróbálhat újabb kódolt szimbólumokat gyűjteni, és újra próbálkozhat a dekódolással.

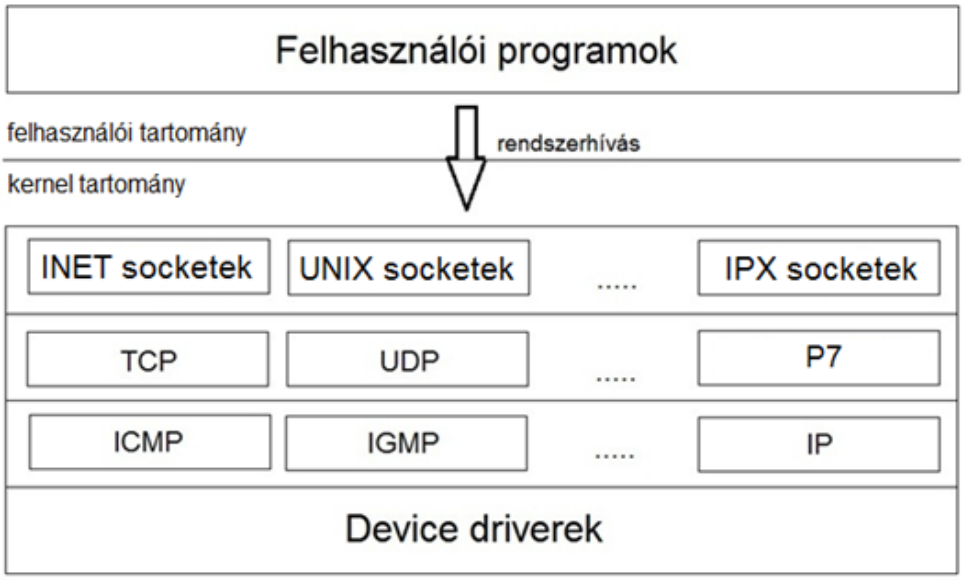
A maximális sebességű adatküldés felveti az alkalmazott mechanizmus igazságosságának kérdését. A versengő folyamok hozzáférése különböző sebességű lehet. Ez egy szűk hálózati

keresztmetszet esetén azt jelentheti, hogy a mohóbb folyamok kiéheztetik a kevésbé aktív folyamokat. Ahhoz, hogy megoldjuk a maximális sebességgel küldő folyamok esetén ezt a problémát, igazságos ütemezést tételezünk fel a routerek esetén. Egy igazságos ütemező implementálása normális esetben nehéz feladat lehet, de sokkal könnyebb azt megvalósítani egy ráta nélküli kódolást alkalmazó átvitel esetén, ahol a csomagvesztések elhanyagolhatók.

A következő részben a Linux kernel hálózati alrendszerének felépítését mutatom be, itt olyan alapfogalmakat fogok ismertetni, amelyek a fejezet későbbi részeit alapozzák meg.

3.2. A Linux kernel hálózati alrendszere

A protokoll implementációja C nyelven, Linux kernelben történt. A felhasznált kernel verziója 2.6.26-2. A megvalósítás, valamint a vizsgálatok és mérések elvégzése során a Debian Linux legutóbbi stabil változatát, a *Lennyt* használtam az általam módosított kernellel. A Linux kernel hálózati alrendszere réteges felépítésű [25]. Az egyes rétegeket a 3.2. ábra szemlélteti.



3.2. ábra. A hálózati alrendszer felépítése

A legalsó rétegben helyezkednek el az egyes hálózati eszközök driverei, amelyek a hardverhez való alacsonyszintű hozzáférésért felelősek. A felette elhelyezkedő rétegben a hálózati rétegbeli protokollok találhatóak, például az *IP* (Internet Protocol), az *ICMP* (Internet Control Message Protocol), az *IGMP* (Internet Group Management Protocol), és a *RIP* (Routing Information Protocol). A következő rétegben helyezkednek el a szállítási rétegbeli protokollok, például a *TCP*, és a *UDP* (User Datagram Protocol). Itt található az új protokoll is, a *P7*. Felette található a socket réteg, amely a *socketeket* tartalmazza. Ezen socketek egy protokoll független interfészt nyújtanak a felhasználói programok számára. A felhasználói programok a socket réteg által nyújtott interfészt használják. A socketek egy kommunikációs link egy végpontját reprezentálják. Két kommunikáló folyamat esetén mindkét folyamat számára rendelkezésre áll egy-egy socket, amelyek az adott oldal számára

azonosítják a linket. A Linux számos különböző socketet támogat, ezeket családokba sorolják. Ilyen a *UNIX*, *INET*, *AX25*, és *IPX* socket család. A kapcsolatorientált protokollokat, mint a TCP, vagy a P7 az INET család támogatja.

Az operációs rendszer által használt memóriát két különböző tartományra oszthatjuk fel, ezen tartományok a *kernel tartomány*, és a *felhasználói tartomány* [26]. A kernel tartományt a kernel használja, amelynek adatstruktúrái itt találhatóak meg. A kernel kódja mindig a processzor privilegizált üzemmódjában hajtódik végre. A felhasználói tartományt a felhasználói programok használják. Az ábrán látható, hogy a felhasználói programok a kernellel *rendszerhívások* segítségével kommunikálnak. A kernelben megtalálható az egyes rendszerhívások száma, és a hozzájuk tartozó függvény, amely meghívódik a kernelben az adott rendszerhívás hatására. Ezeket az információkat a *syscall tábla* tartalmazza. A felhasználói programok számára a rendszerhívások száma az *unistd.h* fájlban érhető el. A rendszerhívások számának segítségével képesek a felhasználói programok azonosítani az adott műveletet. Például, ha egy felhasználói program kapcsolódni szeretne egy távoli szerverhez, akkor a program a *connect()* függvényt hívja meg. Az adott függvénykönyvtárbeli connect() függvény a kernelhez egy rendszerhívás segítségével fordul. A kernel a syscall tábla segítségével képes meghatározni, hogy a kernelben ennek hatására a *sys_connect()* függvényt kell meghívni azokkal a paraméterekkel, amelyeket a felhasználói program átadott. Ez hasonlóan működik az adatküldés (*write()* függvény), az adatfogadás (*read()* függvény), a kapcsolat lezárása (*close()* függvény), és egyéb műveletek végrehajtása esetén is.

3.3. A P7 protokoll alapvető működése

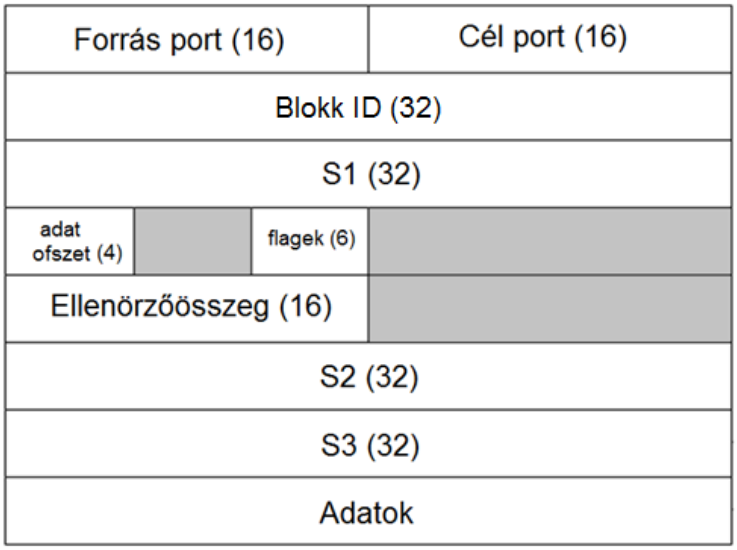
A P7 protokoll az előzőleg ismertetett, torlódásszabályozás nélküli koncepciót alkalmazza. Első lépésben szükség van egy kapcsolat felépítésére az adó és a vevő között. Ez a TCP protokoll kapcsolat felépítéséhez hasonlóan történik. Ha létrejön az összeköttetés, akkor lehetőség nyílik az adatok küldésére. A kernelben a kapcsolat felépítése során egy bizonyos paraméterrel szabályozható mennyiségű, és meghatározott méretű *tárolók* kerülnek lefoglalásra, amelyeket az adatok küldésére, és az adatok fogadására használhatunk fel. Az adatküldéshez a felhasználói programtól érkező, küldendő adatok a kernelben egy éppen szabad tárolóba kerülnek, majd az ilyen módon rendelkezésre álló adatokra alkalmazom a kódolást, amely az adatküldéssel párhuzamosan történik meg. A kódolást mindig egy meghatározott mennyiségű adatra alkalmazom, ezt a meghatározott mennyiséget nevezzük egy *blokknak*. Egy tároló pontosan egy blokkhoz tartozó adatokat tartalmazhat. A P7 protokoll speciális felépítésű fejlécet használ, amelyben a dekódolást segítő információt továbbítok. A küldés során az adó nem használ újraküldést, és a vevő csak egy adott blokk sikeres dekódolásának jelzésére alkalmaz nyugtázást. Ebben az esetben azzal a feltételezéssel élek, hogy a nyugták egy megbízható csatornán kerülnek továbbításra, vagy olyan prioritással ellátva, amely kizárja a nyugták vesztését. A vevő oldal megvárja amíg egy adott blokk esetén elegendő mennyiségű adat érkezik ahhoz, hogy nagy valószínűséggel sikeresen végrehajtható legyen a dekódolás. Ekkor végrehajtja a dekódolást az adott blokkra, és siker

esetén visszajelzést küld erről az adónak. A visszajelzés hatására az adó elkezdheti a soron következő blokk küldését, tehát az előző blokkhoz tartozó adatokat tartalmazó tároló felhasználható lesz az új, küldendő adatok számára. A vevő pedig az előzőleg dekódolt adatokat átadhatja a rájuk várakozó felhasználói programnak. A kapcsolatot az adatátvitel végén bontani kell, ez szintén a TCP protokollhoz hasonló módon történik.

A P7 protokoll adategységére számos esetben *csomagként* fogok hivatkozni. Ennek az az oka, hogy a szállítási réteg feladatai közé tartozik ebben az esetben a küldendő információ csomagokba való tördelése, így a csomagok előállítása valójában itt történik meg. A következő részben az új protokoll fejlécének felépítését mutatom be, majd röviden ismertetem a protokoll egyes fázisainak működését.

3.4. A P7 protokoll fejléce

A fejléc felépítését a 3.3. ábrán láthatjuk.



3.3. ábra. A P7 protokoll fejléce

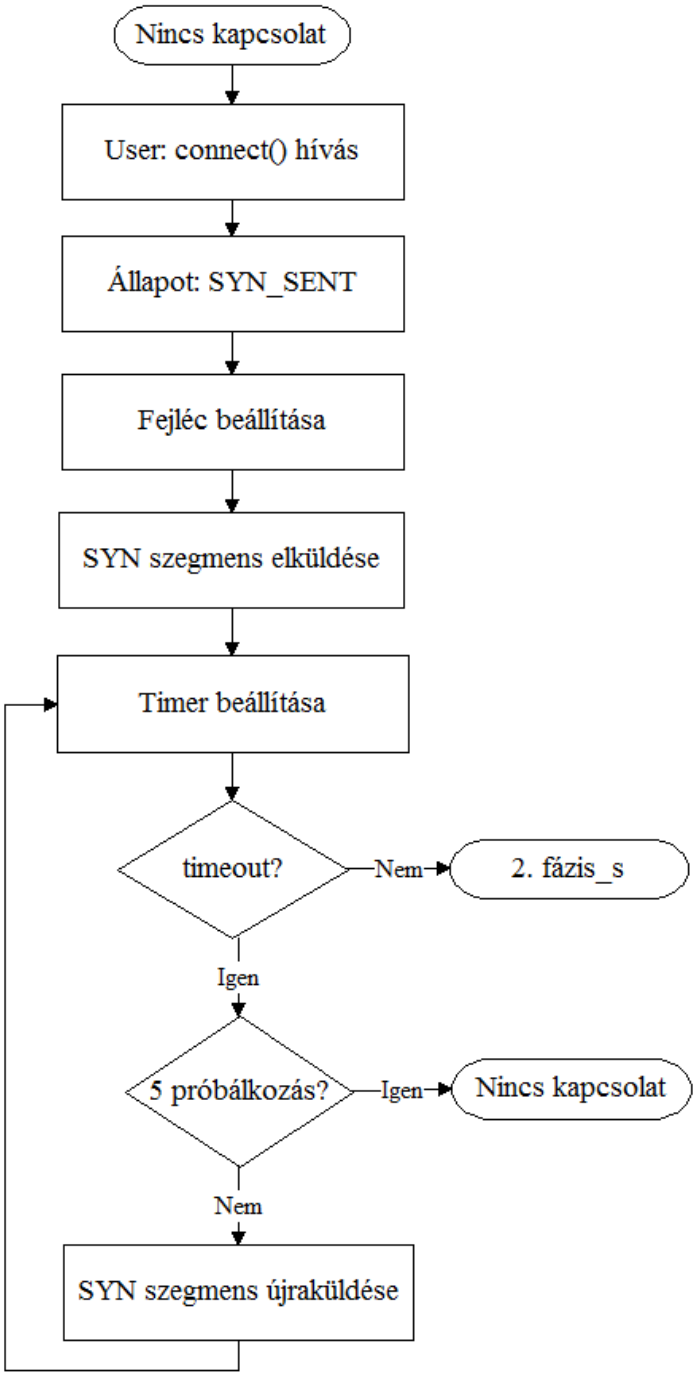
A zárójelben feltüntetett számok az adott mező méretét jelentik, bitekben számítva. Az első, és a második mező azonosítja a küldő, és a vevő alkalmazás számára kiosztott *kommunikációs portot*. A portok segítségével vagyunk képesek különbséget tenni az ugyanazon állomáson futó különböző hálózati alkalmazások között. A *Blokk ID* mező azonosítja azt a blokkot, amelyhez az aktuális csomag tartozik. Az *S1*, *S2*, és *S3* mező 32 bites előjel nélküli egész számokat tartalmaz, amelyek jelentésére a kódolás, és a dekódolás ismertetésénél fogok kitérni. Az *ofszet* mező megmutatja, hogy hány 32 bites szó található a fejlécben, így megmutatja, hogy hol kezdődik az adat. A *flagek* főként a kapcsolat felépítése, és bontása során használatos jelzőbitek. A jelzőbitek közé tartozik például a kapcsolat felépítése során használt *SYN* flag, és a kapcsolat bontása során használt *FIN* flag. Ezen flagek jelentését, és használatukat bővebben a következő szakaszokban, az egyes fázisoknál fogom ismertetni. A továbbiakban az egyes szegmensek neve előtt fogom jelezni, hogy mely flagek vannak beállítva az adott szegmens esetén. Például a *SYNACK* szegmens egy olyan szegmenst

jelent, ahol a *SYN* és az *ACK* flag be van állítva, és a többi flag nincs beállítva. Az *ellenőrzőösszeget* a fejléc, és az adat mező bizonyos részeiből számítjuk, ezzel garantálhatjuk az integritást.

3.5. A kapcsolat felépítése

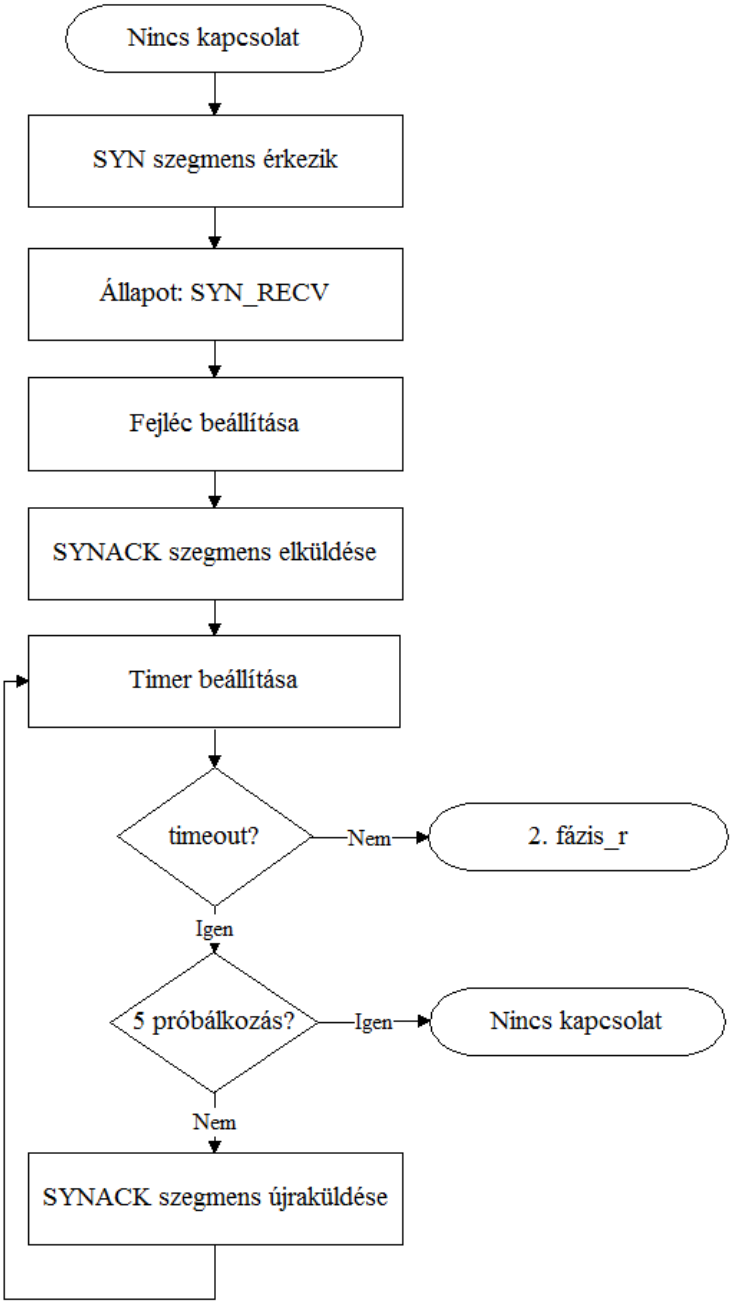
Ebben a részben a P7 protokoll kapcsolat felépítési folyamatát ismertetem. A folyamat a TCP protokollnál alkalmazott módszerhez hasonlít.

A kapcsolat felépítésnek három lépését különböztetem meg. Az első lépést a 3.4. ábrán láthatjuk. Ennek során a kapcsolat felépítést kezdeményező oldal elküld egy *SYN* szegmenst. Az elküldött szegmens fejlécében az adatok dekódoláshoz szükséges információt továbbítunk a vevő számára, ennek részleteit a kódolásról, és a dekódolásról szóló fejezetekben láthatjuk. A kapcsolat felépítési folyamat során a felek karbantartják az állapotukat. A *SYN* szegmens elküldése után a kezdeményező *SYN_SENT* állapotba kerül. Arra az esetre ha a *SYN* szegmens elveszne időzítőt használok, amelynek segítségével szükség esetén újraküldöm a *SYN* szegmenst. Az időzítő kezdetben egy másodpercre van állítva, ennyi idő eltelte után, ha nincs válasz a másik oldaltól, akkor újraküldés történik. A 3.4. ábrán feltételezem, hogy abban az esetben, ha nem történt időtúllépés, akkor megérkezett a *SYN* szegmensre a válasz a másik oldaltól, így átléphetünk a második fázisba. Itt az s a küldőre utal. Az időzítőnél használt időtúllépési értéket minden újraküldés után megduplázom, és 5 újraküldési próbálkozás után a kapcsolat felépítés megszakad, és az erőforrások a küldő oldalon felszabadításra kerülnek.



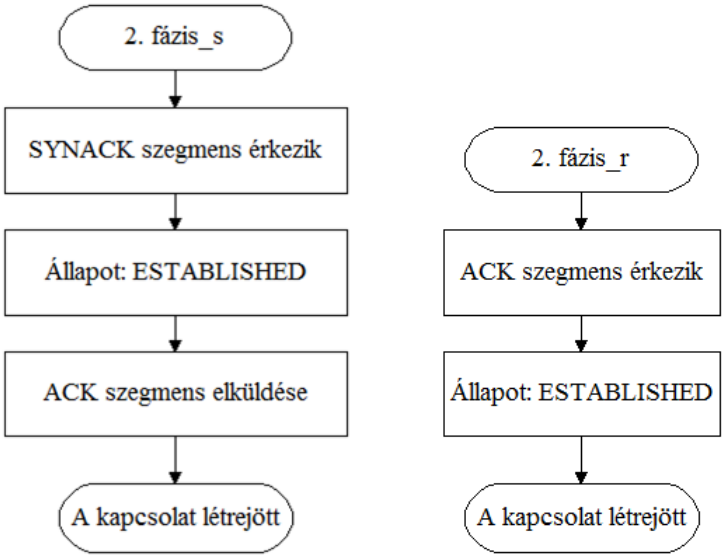
3.4. ábra. A kapcsolat felépítés 1. lépése

A 3.5. ábrán látható, második lépés során a *SYN* szegmenst vevő oldal a szegmens feldolgozása után *SYN_RECV* állapotba kerül, és elküld válaszul egy *SYNACK* szegmenst. A *SYNACK* szegmens fejlécében ő is olyan információkat helyez el, amelyeket később a másik oldal az adatok dekódolásához használhat majd fel. Az ábrán az r a vevőre utal. A *SYNACK* szegmens esetén is időzítőt használok az újraküldéshez. Itt is maximum 5 alkalommal történik újraküldés, utána megszakad a kapcsolat felépítési folyamat, és felszabadulnak az erőforrások.



3.5. ábra. A kapcsolat felépítés 2. lépése

Az utolsó lépést a 3.6. ábra mutatja be. A bal oldali ábra a kezdeményező oldal esetén, a jobb oldali pedig a másik oldal esetén mutatja be a lépéseket. Ezen 3. lépésben a *SYNACK* szegmens feldolgozása után a kezdeményező oldal egy *ACK* szegmenst küld a másik oldalnak, és *ESTABLISHED* állapotba kerül. A másik oldal az *ACK* szegmens feldolgozása után szintén *ESTABLISHED* állapotba kerül. Az *ACK* szegmens esetén nincs szükség időzítőre és újraküldésre, mert ha a szegmens elveszik, akkor a *SYNACK* szegmens újraküldése miatt a másik oldal képes felismerni, hogy az *ACK* szegmens elveszett, és újraküldi azt. Ha elérjük a maximális 5 újraküldött *SYNACK* szegmenst, akkor a szegmenst újraküldő oldal bontja a saját oldalán a kapcsolatot, a másik oldal pedig egy *RST* szegmens által értesülhet a sikertelen kapcsolat felépítésről, és ekkor ő is felszabadíthatja az erőforrásait.



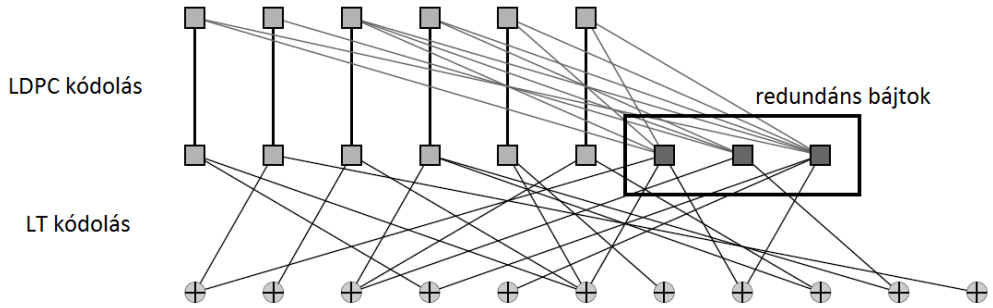
3.6. ábra. A kapcsolat felépítés 3. lépése

3.6. Az adatok kódolása

Ebben a részben az új protokoll esetén alkalmazott hatékony hibavédő kódolást ismertetem. A felhasznált *Raptor kódolás* két részből tevődik össze, egy *előkódolásból*, amelyet *LDPC kódok* (Low-Density Parity-Check) [27] segítségével valósítok meg, és egy *LT kódolásból*. Az első részben a kódolás folyamatát tekintem át, és bemutatom a felhasznált kódokat. A második részben az LDPC, a harmadik részben pedig az LT kódolást tárgyalom részletesen.

3.6.1. A kódolás folyamata

Amikor a kernel megkapja a küldendő adatokat a felhasználótól, akkor ezeket eltárolja az első olyan tárolóban, amely szabad. Az alkalmazott kódolásokat a 3.7. ábrán láthatjuk.

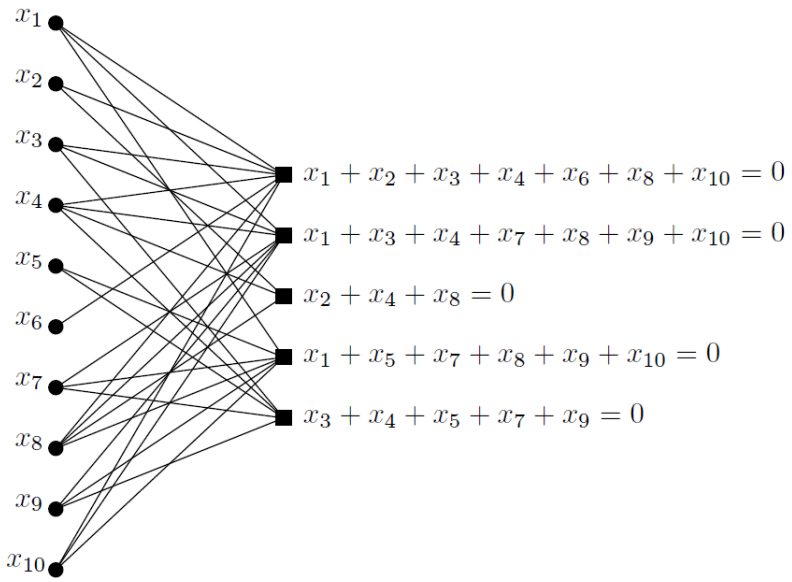


3.7. ábra. Az alkalmazott Raptor kódolás

Annak érdekében, hogy az elkódolt forrásszimbólumok mennyiségénél már kicsivel több kódolt szimbólum beérkezése esetén nagy valószínűséggel sikeres lehessen a dekódolás, először egy hagyományos blokk kódolást alkalmazok előkódolásként, az *LDPC kódokat*. A szimbólumok esetén egységként a bájtot használom a kódolás és a dekódolás során, amely egy 8 bites értéket jelent. Tehát egy bájt alatt egy szimbólumot értek a továbbiakban. Ha a felhasználótól k bájt mennyiségű adatot kapunk, akkor az LDPC kódolás segítségével ebből n bájt fog keletkezni, mert az eredeti k bájthoz redundáns bájtokat rendelünk hozzá. A redundáns bájtok számát tehát e két érték különbsége, azaz $n - k$ adja meg. A jelenlegi implementáció esetén itt 2000 redundáns bájtot generálok, amely az eddigi tapasztalatok alapján elegendő a megfelelő működéshez. Az így kapott n bájt lesz az LDPC kódolás kimenete, és az LT kódolás bemenete, amelyből a később leírtak szerint elméletileg végtelen számú kimenő bájt keletkezik.

3.6.2. Az LDPC kódolás

Egy LDPC kód a definíciója alapján a következőt jelenti. Vegyünk egy páros gráfot, amelynek b bal oldali, és r jobb oldali csomópontja van. A bal oldali csomópontokra gyakran *üzenet csomópontként*, a jobb oldali csomópontokra pedig *ellenőrző csomópontként* hivatkoznak. A továbbiakban én is ezt az elnevezést fogom alkalmazni. Az LDPC kódokra egy konkrét példát mutat a 3.8. ábra.



3.8. ábra. Egy LDPC kód

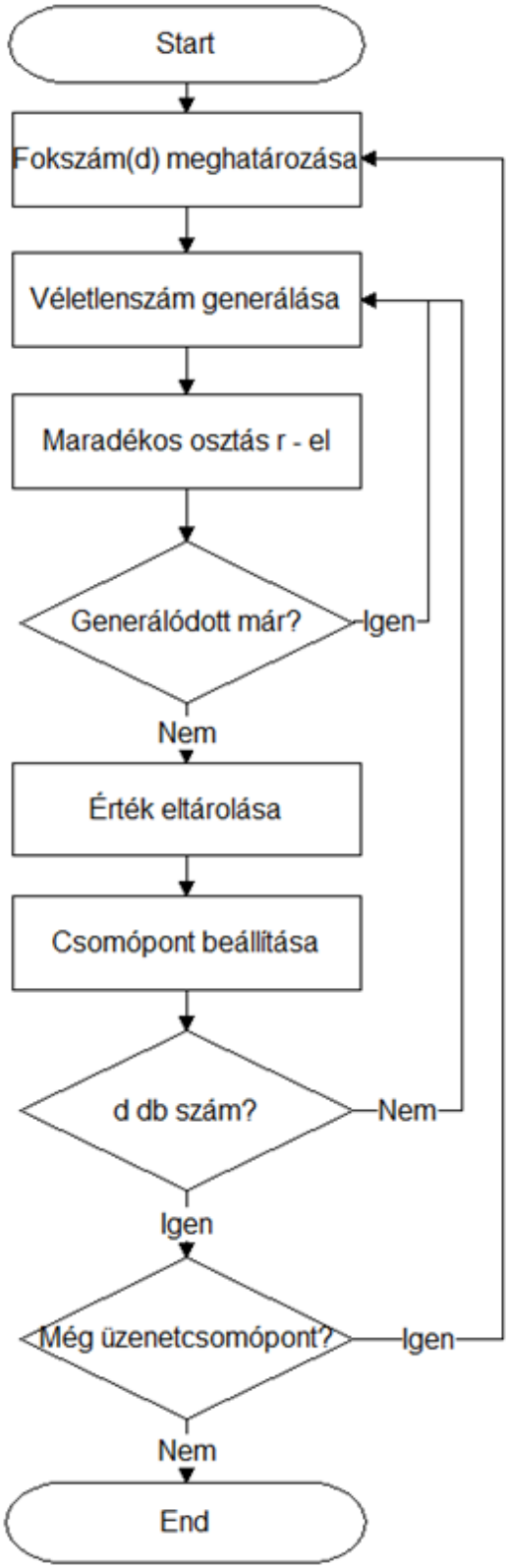
Látható, hogy minden egyes ellenőrző csomópontra teljesül az, hogy a szomszédos üzenet csomópontok összege (amelyet XOR művelettel képzünk) 0 értékű kell legyen.

A jelenlegi implementáció esetén felhasznált LDPC kódokat egy adott valószínűség eloszlás szerint generálom, az ellenőrző csomópontok kezdeti értéke pedig 0. A kódolási folyamatot a 3.9. ábrán láthatjuk. Ennek során minden egyes üzenetcsomópont esetén a konkrét eloszlás segítségével előállítok egy d fokszámot, amely kijelöli azt, hogy az aktuális üzenetcsomópontnak hány szomszédja lesz. Ez után d számú ellenőrző csomópontot választok ki egyenletes eloszlás szerint. Az így kiválasztott ellenőrző csomópontok lesznek a szomszédai az aktuális üzenetcsomópontnak, ezért az értékükhöz (az ábrán látható "csomópont beállítása") hozzáadom az aktuális üzenetcsomópont értékét, a 3.1. képlet szerint:

$$cknode[rand] = cknode[rand] \oplus msg[i] \tag{3.1}$$

Itt *cknode[rand]* a kiválasztott ellenőrző csomópontot, *msg[i]* pedig az aktuális üzenetcsomópontot jelenti. Az üzenetcsomópontok értékeit az LDPC kódolás során a felhasználó által elküldendő üzenet bájtjai jelentik. Tehát minden egyes üzenetcsomópont értéke az elküldendő üzenet egy bájtjának fog megfelelni. A jelenlegi implementáció esetén r értéke 2000, tehát 2000 ellenőrző csomópont értékét állítjuk be. Egy blokk $k = 63536$ üzenetbájtot tartalmaz, így az ellenőrző csomópontokkal együtt (amelyek szintén a blokkhoz tartoznak)

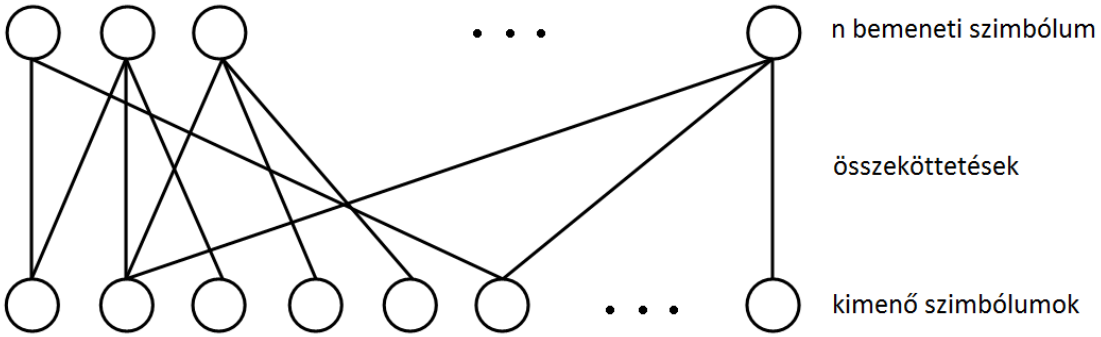
n értéke 65536 lesz. Ez a 65536 bájt képezi az LT kódolás bemenetét.



3.9. ábra. Az LDPC kódolás folyamata

3.6.3. Az LT kódolás

A továbbiakban n fogja jelenteni az LT kódolás bemenetét képező bájtok számát. A kódolás során a megkapott n bájt segítségével állítjuk elő a kimenő bájtokat. Az adatok kódolásához egy $\Omega_0, \Omega_1, \Omega_2, \ldots, \Omega_n$ valószínűség eloszlásra van szükség $\{0, 1, 2, \ldots, n\}$ -n. A bájtok között összeköttetéseket definiálhatunk. Az, hogy hány bemeneti bájt áll egymással összeköttetésben egy kimenő bájt esetén, megadja a kimenő bájt fokát, amit a továbbiakban D fog jelenteni. Ezt a 3.10. ábrán láthatjuk. Az ábrán például az első kimenő szimbólum 2 forrásszimbólummal áll összeköttetésben, ezért a foka 2. Az utolsó kimenő szimbólum egyetlen forrásszimbólummal áll összeköttetésben, ezért a foka 1.



3.10. ábra. A szimbólumok közti összeköttetések

Ekkor $\Omega_i = P(D = i)$. A 3.1. táblázat 2. oszlopában láthatóak ezen Ω_i értékek. Ezeket az értékeket az adott eloszlás esetén előre beállítottam, és $n = 65536$ bemeneti bájt esetén értendők. A táblázatban nem szereplő Ω_i -k 0 értékűek.

A kódolásnak 3 lépését különböztetem meg, az alábbiakban ezt a 3 lépést fogom bemutatni. A kódolás első lépésének megvalósításához a [0,1] intervallumba eső véletlenszámok előállítására lett volna szükség, egyenletes eloszlás szerint. A Linux kernel esetén azonban hatékonysági okokból kerülendű a lebegőpontos számok használata, ezért a $0 \leq \Omega_i \leq 1$ értékek esetén egy olyan transzformációt hajtottam végre, amelynek eredményeként egész számokat kaptam, és ezeket az egész számokat használtam fel a kernel környezetben a véletlenszámok előállításánál. Az Ω_i értékek összege jó közelítéssel 1. A kernel esetén 32 bites előjel nélküli egész számokra volt szükségem, amely a $[0, \ldots, 2^{32} - 1]$ intervallumot jelenti. A transzformáció első lépése egy konstans szorzást jelentett, ahol a c konstans a 3.2. képlet szerint számítható ki:

$$c = \frac{2^{32} - 1}{\sum_{i=0}^{n} \Omega_i} \tag{3.2}$$

Az így kapott c konstanssal megszoroztam minden Ω_i értéket, és az így kapott számot a második lépésben a legközelebbi egész értékre kerekítettem. Az Ω_2 intervallum esetén az így kapott értékből 1-et kivontam, ugyanis itt volt a legkisebb a módosításból adódó eltérés. Az ilyen módon előállt egész számok már pontosan fedték a $[0, \ldots, 2^{32} - 1]$ intervallumot.

A 3.1. táblázatban láthatóak a transzformáció előtti eredeti értékek a 2. oszlopban, és a transzformáció után kapott új értékek a 3. oszlopban.

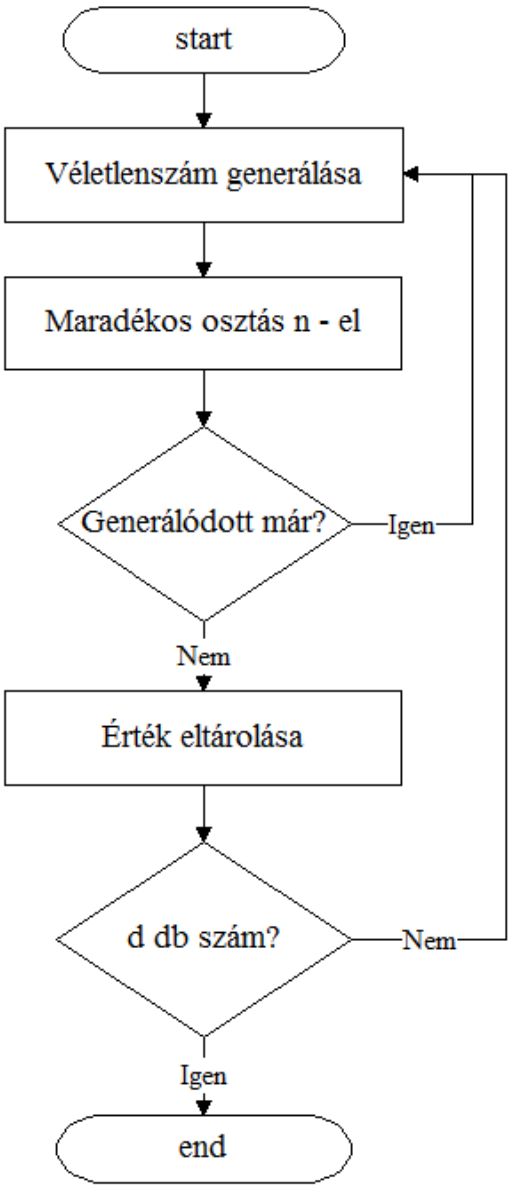
3.1. táblázat. Az elvégzett transzformáció

n=65536 esetén	**A transzformáció előtt**	**A transzformáció után**
Ω_1	0.007969	34226663
Ω_2	0.493570	2119871247
Ω_3	0.166220	713910892
Ω_4	0.072646	312012818
Ω_5	0.082558	354584619
Ω_8	0.056058	240767758
Ω_9	0.037229	159897657
Ω_{19}	0.055590	238757710
Ω_{65}	0.025023	107473182
Ω_{66}	0.003135	13464749

A véletlenszámok előállításához a Linux kernelben megtalálható *Tausworthe* [28] generátort használtam fel, amely 32 bites előjel nélküli egész értékeket generál. A generátor rekurzív elven működik, és az állapotát három változó határozza meg, a továbbiakban ezeket jelenti *s1*, *s2* és *s3*.

A kódolás első lépése során egy véletlenszámot generálok az előbb említett generátor segítségével. Jelen esetben a véletlenszám generálása egyenletes eloszlás szerint történik, de a későbbiekben ezen eloszlás állításával a kód teljesítményjellemzői állíthatóak. Ez a véletlenszám egy 32 bites, előjel nélküli egész szám. Az így generált számról meghatározom, hogy melyik Ω_i intervallumba tartozik. Az első lépés eredménye az intervallum által meghatározott d egész szám, amely a kimenő szimbólum fokát jelenti.

A kódolás második lépését a 3.11. ábra szemlélteti.



3.11. ábra. A kódolás 2. lépése

Az n bemeneti bájtból d bájtot választok ki egyenletes eloszlás szerint. Ehhez d darab különböző véletlenszám generálása szükséges. Minden véletlenszám a $[0, \ldots, n-1]$ intervallumba kell essen. Ennek az oka, hogy az n bemeneti bájt közül az 1. bájt sorszáma 0, az utolsóé pedig *n-1*. Emiatt minden generált véletlenszám esetén maradékos osztást hajtok végre n - el. Az így kapott d különböző véletlenszám kijelöli azoknak a bemeneti bájtoknak a sorszámát, amelyek összeköttetésben állnak majd a kimenő szimbólummal. A különbözőséget úgy garantálom, hogy a generált értékeket eltárolom, és ha az éppen generált érték már előzőleg generálódott, akkor eldobom, és újat generálok helyette.

A harmadik lépéshez az előző lépésben generált d darab sorszámot használom fel. A sorszámok segítségével kiválasztott d darab bájtot XOR kapcsolatba hozom, és így keletkezik

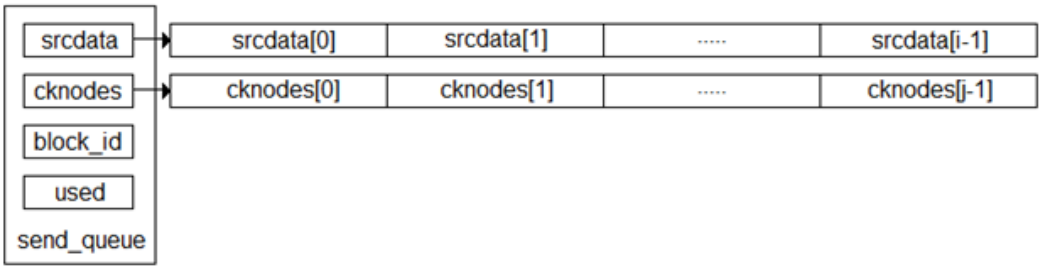
az Y kimenő bájt, azaz $Y = X_{s1} \oplus X_{s2} \oplus \ldots \oplus X_{sd}$, ahol X_{si} jelenti az i - edik kiválasztott bájtot. Bájtok esetén ez bitenkénti *kizáró vagy* művelet elvégzését jelenti, tehát $\{u_1, u_2, \ldots, u_n\} \oplus \{v_1, v_2, \ldots, v_n\} = \{u_1 \oplus v_1, u_2 \oplus v_2, \ldots, u_n \oplus v_n\}$. A harmadik lépés kimenete az Y kódolt bájt. Összesítve tehát így az LT kódolás bemenete n darab szimbólum, és kimenete egyetlen kódolt bájt. Ennek a három lépésnek az egymás utáni ismétlésével érhetjük el azt, hogy tetszőleges hosszúságú kimenetet kapjunk. Az így megkapott, kódolt bájtokból álló folyam lesz az, ami a következő részben leírtak szerint ténylegesen elküldésre kerül.

3.7. Az adatok küldése

A kódolás után most rátérek az adatok küldésének ismertetésére. Először a küldendő adatok ideiglenes tárolására szolgáló tárolók felépítését ismertetem. Ez után tárgyalom az adatok küldésének folyamatát, és végül bemutatom azt is, hogy a fejléc egyes mezői esetén milyen értékeket használok fel, és hogyan állítom elő ezeket az értékeket.

3.7.1. A küldési tárolók felépítése

Minden összeköttetés esetén a kapcsolat felépítése során egy adott paraméter által meghatározott számú tároló kerül lefoglalásra. Ezeknek a tárolóknak egy részét az adatküldés esetén, másik részét az adatfogadás esetén használhatjuk. A továbbiakban az első eset szerinti tárolókra a *küldési tároló* elnevezést, a második esetben pedig a *vételi tároló* elnevezést fogom alkalmazni. Tehát egy küldési tároló az, amelyben a küldendő adatokat a kernel ideiglenesen eltárolja. Egy küldési tároló minden olyan információt tartalmaz, amely szükséges az adatküldéshez. Egy ilyen tároló felépítését a 3.12. ábrán láthatjuk.



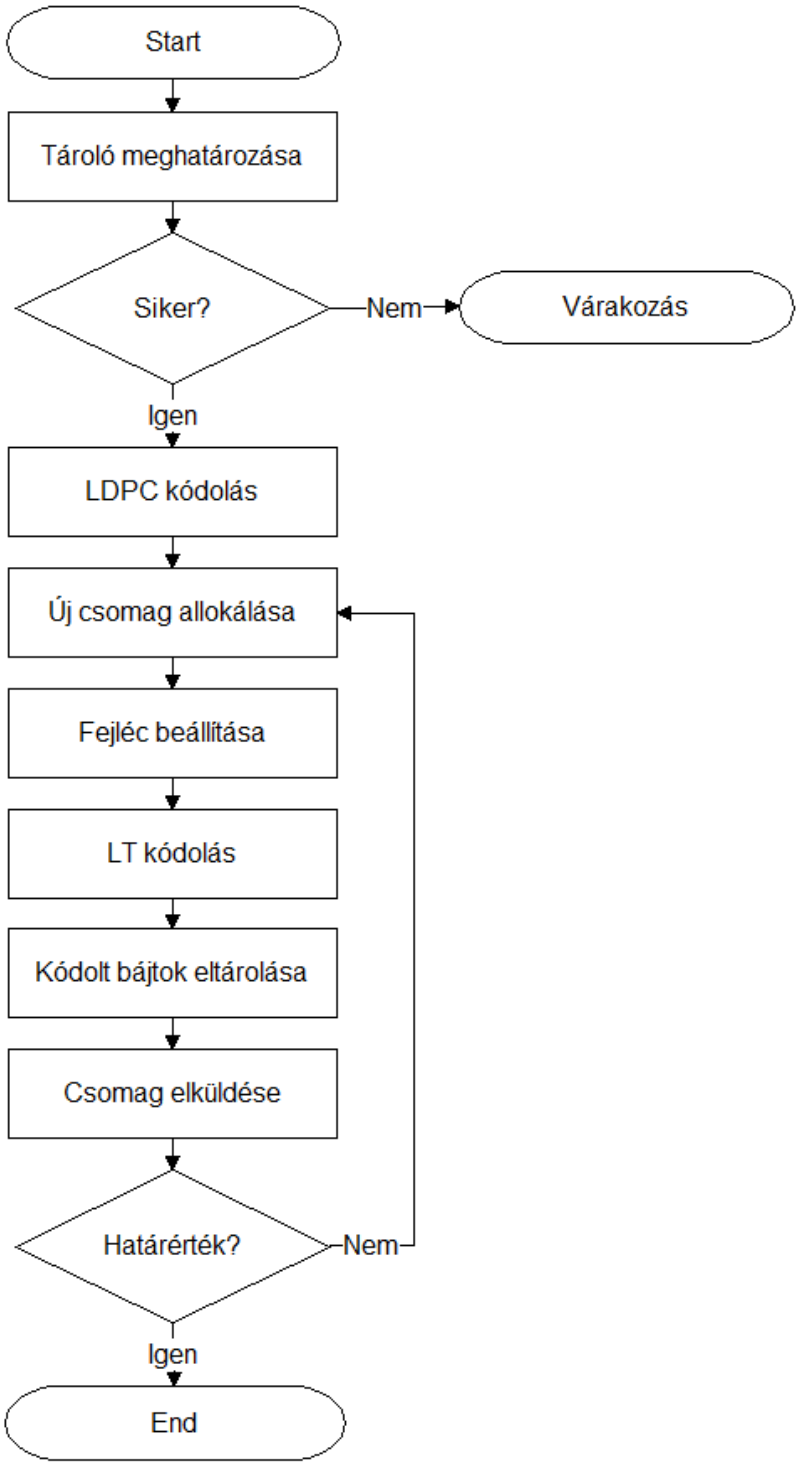
3.12. ábra. Egy küldési tároló felépítése

Tartalmazza a felhasználótól megkapott adatot (*srcdata*), amely maximum 63536 bájt lehet. Ezen kívül itt található az LDPC kódolás eredménye is, tehát a generált ellenőrző csomópontokat is itt tárolom el (*cknodes*), amelyek 2000 bájtot foglalnak. Ezeken kívül minden tároló tartalmaz egy blokk azonosítót (*block_id*), amely egy nemnegatív egész szám, és végül egy olyan értéket, amely azt jelzi hogy szabad-e az adott tároló, vagy foglalt (*used*). A blokkokat 0-tól kezdve, egyesével növekvően sorszámozom. Egy tárolóhoz egyetlen blokkot rendelhetünk hozzá, és ez a tároló ennek a blokknak az azonosítóját tartalmazza.

3.7.2. Az adatok elküldése

Az előző részben leírtak szerint, ha a felhasználó adatot szeretne küldeni, akkor ezt az adatot a kernel egy éppen szabad tárolóban elhelyezi. Abban az esetben, ha nincs szabad tároló, akkor a felhasználói folyamat addig várakozik (alszik), amíg valamelyik tároló fel nem szabadul. Miután az adatokat eltároltuk, elvégezzük az LDPC kódolást is. Az adatküldés esetén egy csúszóablakos megoldást alkalmazok, így a küldő egy adott, konfigurálható számú blokkot elküldhet anélkül, hogy nyugtára kéne várakoznia. Az ablakot a használt állapotú tárolók alkotják, az ablak maximális mérete pedig az összes küldési tároló számával egyezik meg. A küldés az adott blokk LDPC kódolása után azonnal megtörténik, ez azt jelenti, hogy a felhasznált tároló esetén elküldünk egy adott, beállítható számú csomagot, amely a blokkhoz tartozó, LT kódolás során kapott kódolt bájtokat tartalmazza. A leírt folyamat addig folytatódik, amíg az ablakban lévő összes blokk esetén el nem küldtük az adott mennyiségű csomagot. Ekkor abban az esetben, ha még nem érkezett nyugta egyetlen blokkra sem, akkor várakozás történik. A vevőtől érkező, valamely blokkra vonatkozó sikeres visszajelzés esetén azt a tárolót, amelyet az adott blokkhoz rendeltünk, és amelyet a vevő nyugtázott, felszabadítjuk, és ekkor újra felhasználható lesz az újabb küldendő adatok tárolására.

A 3.13. ábrán egyetlen blokk küldése esetén elvégzett lépéseket láthatjuk. Először meghatározzuk az előzőleg leírtak szerint, hogy mely tárolót használjuk a küldendő blokkhoz tartozó információk eltárolására. Ha ekkor azt észleljük, hogy minden tároló foglalt, akkor várakozó állapotba helyezzük a felhasználói folyamatot. A várakozás akkor ér véget, ha valamely blokkra nyugta érkezik, és ilyen módon egy tároló felszabadul. A megfelelő tároló hozzárendelése után elvégezzük az LDPC kódolást is az adott blokk esetén. Az ábrán látható következő lépés során egy csomagot allokálunk, majd beállítjuk a csomag P7 fejlécének mezőit. Ezt a lépést a következő részben részletesen ismertetem. Előzőleg már láthattuk, hogy az LT kódolás során egyetlen kódolt bájt keletkezik. Egy csomagban azonban 1420 bájtot továbbítunk, ezért az LT kódolást addig hajtjuk végre, amíg 1420 kódolt bájt nem keletkezik. Ezt az ábrán egyetlen lépésként láthatjuk. Az 1420 bájtos mennyiség oka az, hogy így egy csomag mérete jó közelítéssel eléri az Ethernet LAN-ok esetén lehetséges maximális átviteli egység (MTU) méretét, ami 1500 bájt. Az így kapott kódolt bájtokat eltároljuk egy ideiglenes tároló helyen, és az IP réteg segítségével elküldésre kerül a létrehozott csomag. A csomagok generálása és elküldése addig történik az aktuális blokk esetén, amíg annyi csomagot nem küldtünk, amely már jó eséllyel elegendő lesz a dekódoláshoz. Az ábrán a *határérték* jelenti ezt az értéket. A jelenlegi implementáció esetén ilyenkor 49 csomagot küldök el, amelyek így összesen $1420 \cdot 49 = 69580$ kódolt bájtot tartalmaznak, tehát 49 csomag a határérték. A további blokkok küldése a leírtakhoz hasonlóan történhet meg.



3.13. ábra. Az adatküldés folyamata

3.7.3. A fejléc mezőinek beállítása

A kódolás során generált véletlenszámokat a vevőnek is elő kell tudnia állítani, mert csak így képes sikeresen elvégezni a dekódolást. Ez mind az LDPC, mind az LT kódolás esetén külön-külön szükséges, ezért két különböző megoldást alkalmazok.

Az LDPC kódolás esetén a kapcsolat felépítés során a *SYN* szegmenshez tartozó fejléc tartalmazza azt a három változót (*s1*, *s2*, *s3*), amely meghatározza a kezdeményező oldal véletlenszám generátorának állapotát. Amikor a másik oldalhoz megérkezik a *SYN* szegmens, akkor a vevő ezt a 3 változót eltárolja, és válaszul a *SYNACK* szegmensben ő is elhelyezi a saját generátorának kezdeti állapotát meghatározó 3 változót, amelyet pedig a kezdeményező oldal tárol majd el. Ha később adatküldésre kerül sor, akkor az adatot küldő oldal abból az állapotból indulhat majd ki, amelyet előzőleg átküldött a kapcsolat felépítés során a másik oldalnak, és ezt az állapotot használja majd az LDPC kódolás során. Így a másik oldal, amelynek az adatot küldték, az LDPC dekódolás esetén képes ugyanazokat a véletlenszámokat előállítani, mert ismeri az adatot küldő fél állapotát. Itt valójában az történik, hogy mielőtt egy adott blokkhoz tartozó ellenőrző csomópontokat előállítanám, a véletlenszámgenerátort egy adott függvény segítségével, a saját kezdeti állapot, és az adott blokk azonosítója alapján egy új állapotba állítom be, és így generálom az ellenőrző csomópontokat. A másik oldal szintén ismeri a felhasznált kezdeti állapotot, és ismeri a blokk azonosítóját is, és ugyanazt a függvényt alkalmazva így képes ugyanabba az állapotba állítani a generátorát, és elvégezni a dekódolást.

Az LT kódolás esetén a 3.13. ábrán látható, hogy a csomag allokálása után beállítjuk a fejléc mezőit. Ezt úgy valósítottam meg, hogy minden csomag esetén az LT kódolás megkezdése előtt kiolvasom a véletlenszám generátor állapotát meghatározó három változót (*s1*, *s2*, *s3*), majd ezeket elhelyezem az adott csomaghoz tartozó fejlécbe, és így küldöm el. Az átvitt információ alapján a vevő oldal képes az LT dekódolás során ugyanazokat a véletlenszámokat generálni, mint amit a küldő oldal az LT kódolás során használt az adott csomag esetén. Végül, az adott blokk azonosítója a *Blokk ID* mezőben kerül továbbításra, ezáltal tudja a vevő eldönteni, hogy a beérkező csomagban található kódolt bájtok mely blokkhoz tartoznak.

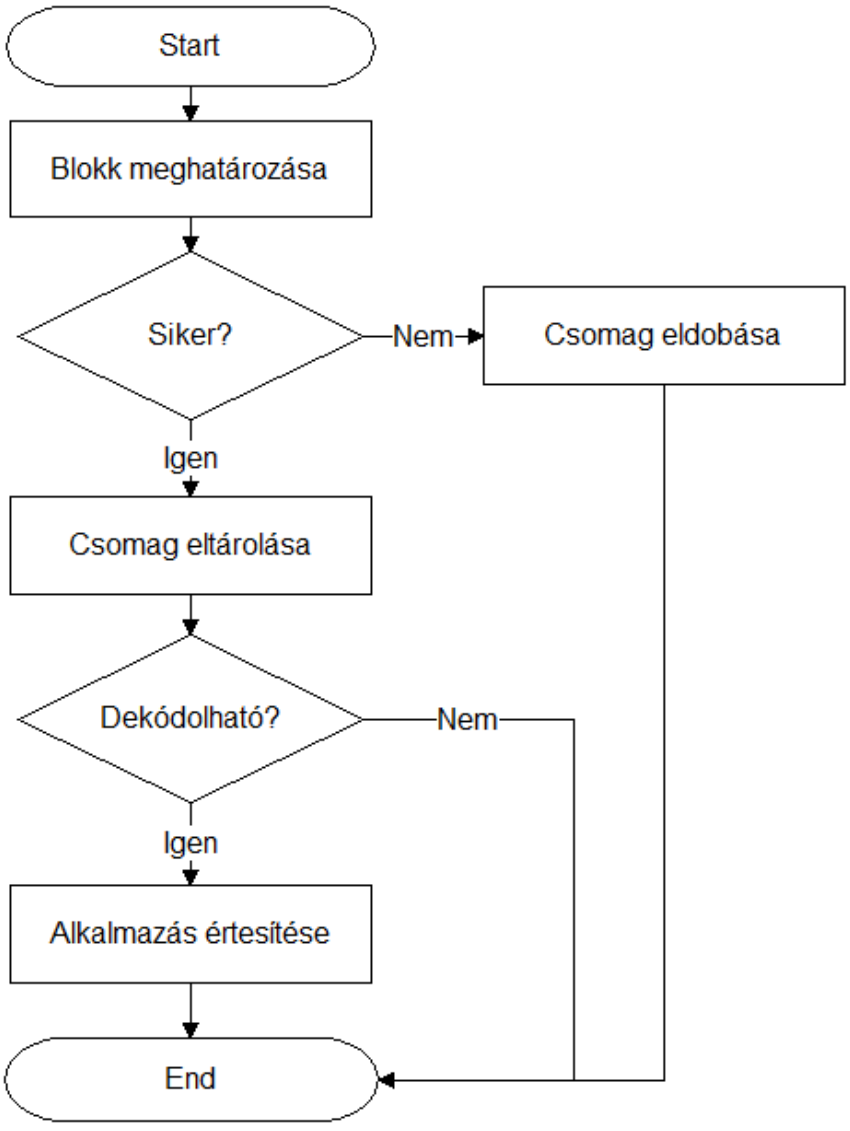
3.8. Az adatok fogadása

Ebben a részben az adatok fogadásának folyamatát mutatom be. Az adatok fogadása, és az adatok dekódolása egymástól teljesen elkülönítve történik meg. Ennek az az oka, hogy az adatok fogadása esetén a megfelelően gyors működés érdekében csak nagyon kevés művelet elvégzésére van lehetőség egy beérkező csomag esetén. Ellenkező esetben egy adott határértékig (alapértelmezett értéke 1000 csomag) a beérkező csomagok egy alsóbb rétegben található listán eltárolásra kerülnek, majd az értéket meghaladva csomagvesztés lép fel. Az adatok dekódolását, és az ehhez szükséges adatszerkezeteket a következő alfejezetben fogom ismertetni.

Amikor a felhasználói alkalmazás adatot szeretne fogadni, olvasni, akkor mindig a következő rendelkezésreálló blokk (dekódolt) adatait használhatjuk fel. Tehát először a 0.

sorszámú blokkhoz, majd egyesével növekedve mindig a következő sorszámú (1, 2, ...) blokkhoz tartozó adatokat felhasználva történhet meg az adatok olvasása, ilyen módon biztosítva a megfelelő sorrendet. Ha a következő szükséges blokk még nem áll rendelkezésre az olvasási kérés idején, akkor a felhasználói alkalmazás addig várakozik, amíg a várt blokkhoz tartozó csomagok meg nem érkeznek.

Az adatfogadás folyamatát a 3.14. ábrán láthatjuk. Az adó oldalhoz hasonlóan a vevő oldal is tárolókat alkalmaz a beérkező csomagokban található információk eltárolására, például egy vételi tároló tartalmazza a hozzárendelt blokk esetén megérkezett kódolt bájtok számát. A tárolók felépítését részletesen ismertetem a következő, dekódolásról szóló alfejezetben.



3.14. ábra. Az adatfogadás folyamata

Egy beérkező csomag esetén először meghatározzuk, hogy melyik blokkhoz tartozik. Ezt a csomag fejlécében található, *Blokk ID* mező alapján tudjuk eldönteni. Ha van olyan vételi tároló, amelyet már előzőleg hozzárendeltünk ehhez a blokkhoz, akkor ezt a vételi tárolót használhatjuk az aktuális csomag esetén is, ellenkező esetben pedig két eset lehetséges. Vagy találunk egy szabad tárolót ennek a blokknak a számára, vagy nincsen szabad tároló. Ez utóbbi esetben eldobjuk a beérkező csomagot. Az első esetben pedig valamely szabad tárolót hozzárendeljük ehhez a blokkhoz. Az így meghatározott vételi tároló esetén a csomagban található kódolt bájtok mennyiségének megfelelően növeljük a megérkezett adatmennyiséget, majd a csomag tartalmát egy láncolt listában eltároljuk azért, hogy később felhasználhassuk. Ha az aktuális blokk esetén megérkezett a dekódoláshoz szükséges adatmennyiség, akkor felébresztjük a felhasználói alkalmazást, ha az éppen várakozott. Ez után megkezdődhet a felhasználói alkalmazás kérésnek teljesítése, amelynek során dekódolhatjuk az adott blokkot az eltárolt adatok alapján.

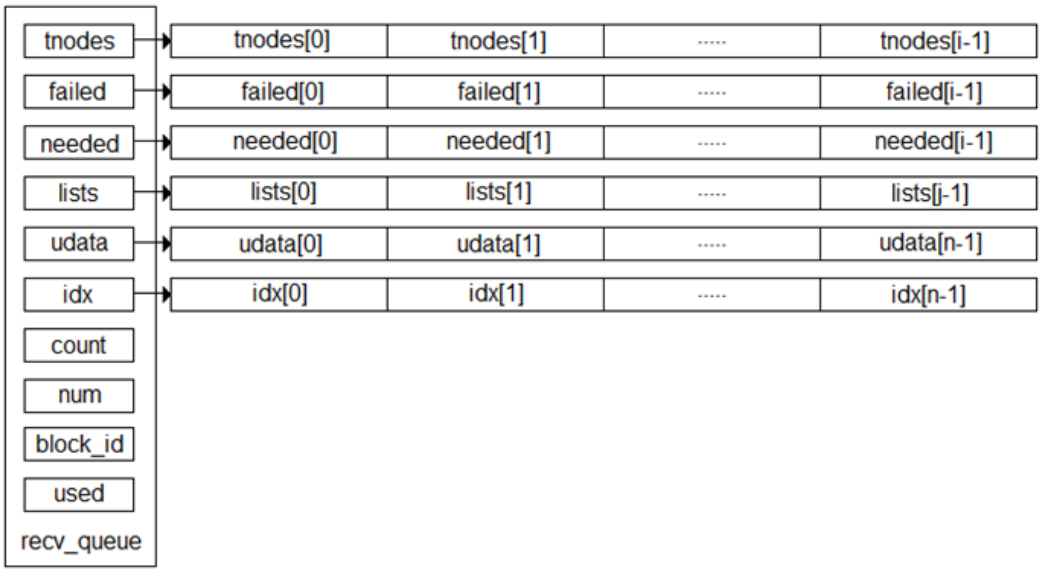
3.9. Az adatok dekódolása

Ebben a részben először a vételi tárolók felépítését fogom bemutatni, és erre számos esetben hivatkozni fogok az alfejezet későbbi részében. Ez után a dekódolás folyamatának ismertetése során külön tárgyalom majd az LT dekódolást, és az LDPC dekódolást.

3.9.1. A vételi tárolók felépítése

A kapcsolat felépítése során egy adott paraméter határozza meg, hogy hány vételi tárolót hozunk létre az összeköttetés számára. A tárolók száma a továbbiakban nem változhat. Minden egyes tárolót egyetlen blokkhoz rendelhetünk hozzá. Ez a hozzárendelés akkor történik meg, amikor egy olyan csomag érkezik be, amelyhez még nem tartozik tároló, és éppen van szabad tároló a csomagban található kódolt bájtok számára. A tároló olyan adatszerkezeteket tartalmaz, amelyeket a dekódolás során használunk fel, illetve itt tárolódik el a dekódolt adat is addig, amíg a felhasználói alkalmazás ki nem olvassa azt.

A 3.15. ábrán egy vételi tároló felépítését láthatjuk. Az első három mezőt (*tnodes*, *failed*, *needed*) az LDPC dekódolás során, a következő három mezőt (*lists*, *udata*, *idx*) pedig az LT dekódolás során használjuk fel. Ezen mezők jelentését a konkrét dekódolásról szóló alfejezetben ismertetem. A *count* mező megmutatja, hogy az adott tárolóban jelenleg hány kódolt bájt található. A *num* mező tartalmazza, hogy hány bájt adatot sikerült eddig dekódolni. Ezt azért szükséges nyilvántartani, mert több iterációra is szükség lehet a dekódolás esetén, mire minden adatot vissza tudunk állítani. A *block_id* a tárolóhoz rendelt blokk azonosítóját tartalmazza. Végül, a *used* mező értéke 1, ha az adott tároló foglalt, és 0, ha szabad. A vételi tárolók által foglalt memóriát a kapcsolat bontása során szabadítjuk fel.



3.15. ábra. Egy vételi tároló felépítése

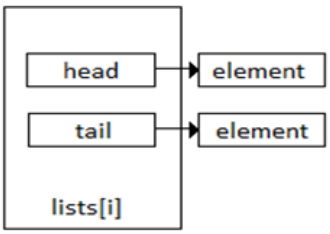
3.9.2. Az LT dekódolás

Az adatok fogadásáról szóló fejezetben láthattuk, hogy a beérkezett csomag tartalmát egy láncolt listában tároljuk el. Mielőtt az LT dekódolást végrehajthatnánk, először ezt a láncolt listát szükséges feldolgozni. A listán található adatok feldolgozása, és a dekódolás a felhasználói alkalmazás kérésének teljesítése során történnek meg.

A láncolt lista feldolgozása során minden egyes eltárolt csomag esetén végignézem a csomagban lévő kódolt bájtokat. Minden kódolt bájtot elhelyezek egy másik, láncolt listákat tartalmazó adatszerkezetben (*lists*), amely az adott blokkhoz tartozó vételi tároló része, és amelynek a felépítése olyan, hogy a későbbiekben segíti majd a dekódolást. Ennek a folyamatnak az idejét a kernel méri, és a mérések bemutatása során *rendezési időként* fogok rá hivatkozni. Ha az adott blokk esetén megérkezett a szükséges mennyiségű kódolt adat, és így már nagy valószínűséggel sikeresen elvégezhető a dekódolás, akkor végrehajtom a dekódolást. Annak meghatározására, hogy mikor érdemes megpróbálkozni a dekódolással a vevő egy heurisztikus értéket használ. Jelenleg ezen érték 69580 bájt beérkezett adatot, azaz 49 csomagot jelent. Ennyi csomag megérkezése esetén próbálkozik a vevő először a dekódolással egy adott blokk esetén. A dekódolás során először az LT dekódolást hajtjuk végre, majd az LDPC dekódolást. Az LDPC dekódolás így az LT dekódolás által visszaállított ellenőrző csomópontok, és a visszaállított üzenetbájtok értékét is felhasználhatja. A két dekódolást addig futtatom ebben a sorrendben, amíg már egyetlen bájtot sem sikerül visszaállítani. Ennek két oka lehet. Az egyik, hogy minden bájtot visszaállítottunk. Ekkor a dekódolás sikeres. A másik, hogy nincs elég információnk a dekódoláshoz, és még vannak dekódolatlan bájtok. Ekkor a dekódolás sikertelen. Ha a dekódolás sikeres, akkor az adott blokk esetén nyugtát küldünk a másik oldal számára, hogy az adatot küldő oldal felszaba-

díthassa a blokkhoz rendelt küldési tárolót. Az elküldött nyugta esetén a fejléc *Blokk ID* mezője tartalmazza a dekódolt blokk sorszámát. Ez után a felhasználói folyamat megkaphatja a dekódolt adatokat. Végül, a vevő felszabadítja a blokkhoz rendelt vételi tárolót, így a tároló az új beérkező adatok számára rendelkezésre fog állni. Abban az esetben ha a dekódolás sikertelen volt, akkor is nyugtát küldünk az adott blokkra, mert a küldő csak így képes felszabadítani a tárolót, de a felhasználói alkalmazás számára a dekódolt adatok helyett egy hibára utaló értéket adunk vissza. A tényleges dekódolás idejét szintén méri a kernel, erre később, a méréseknél *dekódolási időként* fogok hivatkozni.

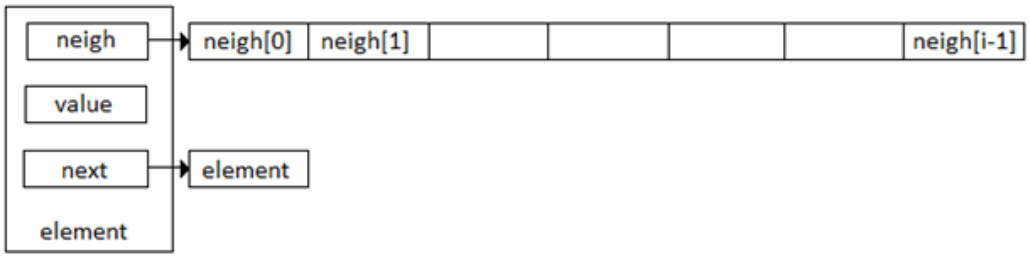
Az LT dekódolás esetén a 3.15. ábrán látható második három mezőt (*lists*, *udata*, *idx*) használjuk. A *lists* mező egy olyan mutatót tartalmaz, amely egy listákat tartalmazó összetett adatszerkezetre mutat. Az ábrán látható j értéke a jelenlegi implementáció esetén 10, tehát 10 listát használunk. Ezeket a listákat az összeköttetésben levő bájtok száma alapján különítem el. A lehetséges értékek a 3.1. táblázatban láthatóak, $n = 65536$ érték esetén ezek a következők: *1, 2, 3, 4, 5, 8, 9, 19, 65, 66*. Ez azt jelenti, hogy az első listán azok a kódolt bájtok fognak szerepelni, amelyek egyetlen üzenetbájttal állnak összeköttetésben, az utolsó listán pedig azok, amelyek 66 üzenetbájttal. A 3.16. ábrán egy adott lista felépítését láthatjuk.



3.16. ábra. Egy lista felépítése

Minden lista két mutatót tartalmaz. A *head* mutató a lista első elemére mutat, a *tail* mutató pedig a lista legutolsó elemét jelöli ki. Ilyen módon ha újabb elemet kell a listához hozzáfűzni, akkor ezt a *tail* mutató segítségével végigjárás nélkül megtehetjük, mert ezen mutató által kijelölt listaelem után kell fűzzük az új elemet.

Végül, a legalsó hierarchiaszinten az egyes listaelemek szerepelnek. A listaelemek felépítését a 3.17. ábrán mutatom be.



3.17. ábra. Egy listaelem felépítése

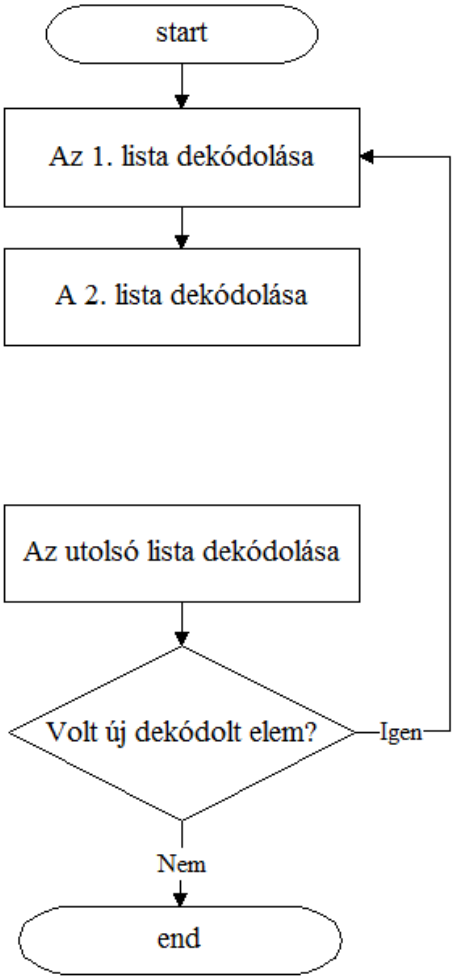
Minden listaelem tartalmaz egy mutatót (*next*), amely az adott lista következő elemére mutat. A *value* mező tartalmazza egyetlen kódolt bájt értékét azon bájtok közül amelyeket a bejövő csomag tartalmazott. A *neigh* tartalmazza azon szomszédok sorszámát, amelyek részt vettek a *value* kódolt bájt esetén az összeköttetésben. Ennek a tömbnek a hossza az egyes listák esetén különböző, de egy adott lista esetén mindig megegyezik. Ilyen módon az első számú listán azon elemek szerepelnek, amelyek esetén a kódolás során az összeköttetésben csak egyetlen elem vett részt. A második lista esetén pedig két elem vett részt az összeköttetésben. Az utolsó lista esetén, ha az $n = 65536$ esetet tekintjük, akkor 66 elem vett részt az összeköttetésben. A listák elemeit úgy állítjuk be, hogy minden egyes bejövő csomag esetén a fejlécben megtalálható 3 értéket (*s1*, *s2*, *s3*) használom fel arra, hogy a vevő véletlenszám generátora a megfelelő állapotba kerüljön. A küldő a kódolás megkezdése előtti állapotot helyezte el a csomag fejlécében, ilyen módon a vevő képes lesz arra, hogy ugyanazokat a véletlenszámokat állítsa elő, mint a küldő a kódolás során. Az egyes csomagok összekeveredhetnek a továbbítás során, de mivel minden csomag tartalmazza ezt az információt, ezért az összekeveredés okozta problémát képesek vagyunk kezelni. Tehát annak meghatározása, hogy hány szimbólum vett részt az összeköttetésben, és az egyes szimbólumok sorszámának előállítása a fejlécben található információ segítségével történik.

Egy vételi tároló esetén a következő, *udata* mező tartalmazza a sikeresen dekódolt bájtokat, illetve itt tárolódnak el a visszaállított ellenőrző csomópontok értékei is, tehát az ábrán látható n értéke 65536. Ebből a mennyiségből 2000 bájtot foglalnak az ellenőrző csomópontok.

Az *idx* mező azt mutatja meg, hogy az *udata* mezőben mely bájtokat dekódoltuk sikeresen. Az *idx[i]* értéke 0, ha az *udata[i]* még nincsen dekódolva, ellenkező esetben 1, ekkor pedig *udata[i]* tartalmazza a dekódolt értéket.

Végül, az LT dekódolás megvalósítását mutatom be. A dekódolás esetén azt az elvet követem, amely szerint az első lista elemei dekódolhatóak. Ennek az oka, hogy ezen elemek esetén az összeköttetésben csak egyetlen elem vett részt, tehát értékük ismert, mert a kódolás esetén a $Y = X_{s1}$ műveletet végeztük el. A második lista elemei esetén két elem vett részt az összeköttetésben. Itt a dekódoláshoz felhasználom az első lista alapján dekódolt elemeket. Ezen kívül a második lista elemeinek dekódolásához felhasználhatóak a második lista már dekódolt elemei is. A dekódolás a *kizáró vagy* művelet tulajdonságai miatt elvégezhető a kódolt bájt Y, és a szomszédok sorszámának ismeretében, ha csak egyetlen olyan szomszéd van, amelyet nem ismerünk. Ennek oka, hogy a második lista elemei esetén a küldő a $Y = X_{s1} \oplus X_{s2}$ műveletet végezte el. A kódolt bájt Y ismert. Ha az első szomszédot nem ismerjük, de a második szomszédot ismerjük, akkor a $X_{s1} = Y \oplus X_{s2}$ műveletet végrehajtva megkaphatjuk az első szomszédot. Hasonlóan meghatározható a második szomszéd is, ha csak az első szomszédot ismerjük. A harmadik, és további listák esetén is ilyen módon működik a dekódolás. Mindig az adott lista már dekódolt elemeit, és az előző listák dekódolt elemeit használom fel az újabb adatok dekódolásához. A dekódolás folyamatát a 3.18. ábra szemlélteti. Az előzőleg leírtak alapján sorban végigmegyek a listákon. Végül ellenőrzöm, hogy volt-e olyan bájt, amelyet sikerült dekódolni valamely lista esetén. Ha egyetlen bájtot sem sikerült dekódolni a listák végigjárása során, akkor biztos, hogy nincs több olyan

szimbólum, amelyet dekódolni tudnánk, ezért nem próbálkozunk újra. Ellenkező esetben újrapróbálkozunk az első listától kezdve végigjárással. Ez amiatt szükséges, mert egy adott listán végiglépve találhatunk olyan elemet, amelyet sikeresen dekódolni tudunk, és az előző listák elemei, vagy az adott lista előző elemei esetén felhasználható lenne a dekódoláshoz. A magasabb sorszámú listák esetén egyre többen vesznek részt egy összeköttetésben, ezért az alacsonyabb sorszámú listák esetén megpróbálok minden lehetséges bájtot dekódolni, hogy a magasabb sorszámú listák esetén kevesebb dekódolatlan bájt maradjon. Az LT dekódolás során számolom azt, hogy hány üzenetbájtot sikerült visszaállítani, és ez lesz az LT dekódolást végző függvény visszatérési értéke.



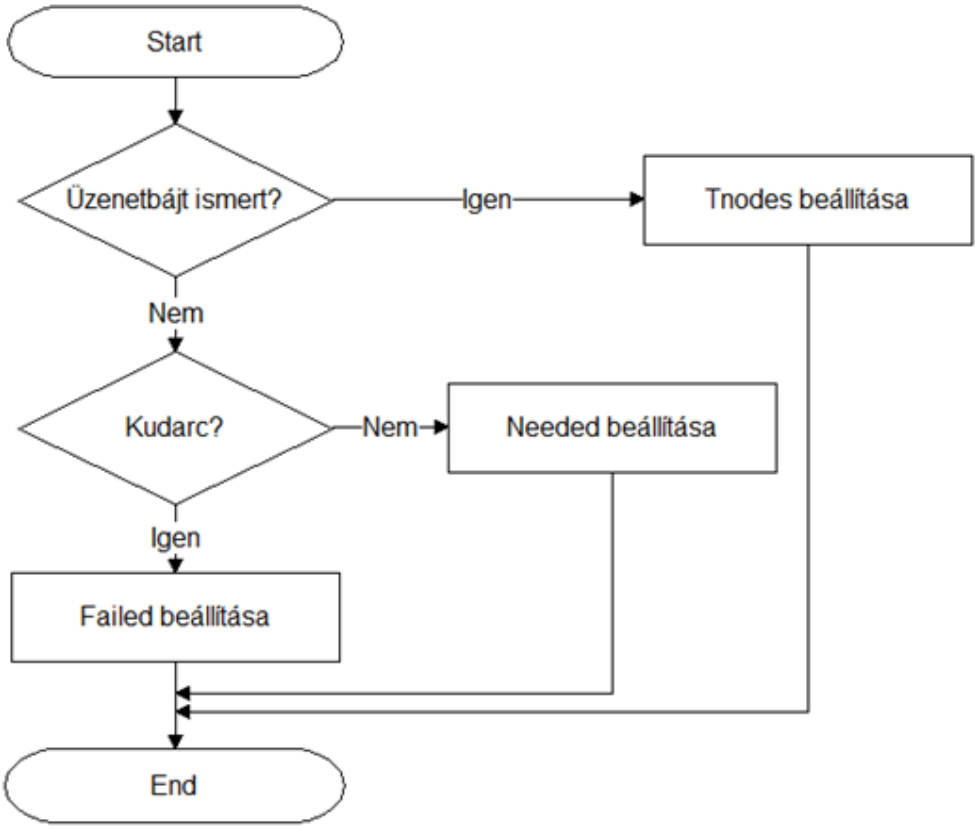
3.18. ábra. Az LT dekódolás folyamata

3.9.3. Az LDPC dekódolás

Az LDPC dekódolás esetén a 3.15. ábrán látható első három mezőt (*tnodes*, *failed*, *needed*) használjuk. Az LDPC dekódolást végző függvény minden futtatáskor inicializálja ezeket a mezőket, és csak az aktuális futás esetén van szerepük az értékeknek. A *tnodes[i]* érték megmutatja, hogy az adott időpontban mi a legjobb ismert értéke az *i.* ellenőrző csomópontnak. A *failed[i]* értéke 1, ha az *i.* ellenőrző csomópontot nem lehetséges visszaállítani. Ellenkező esetben 0. A *needed[i]* érték tartalmazza annak az üzenetbájtnak a sorszámát, amelyre szükség van ahhoz, hogy az adott ellenőrző csomópontot visszaállíthassuk. Az LDPC dekódolást három lépésre bonthatjuk fel.

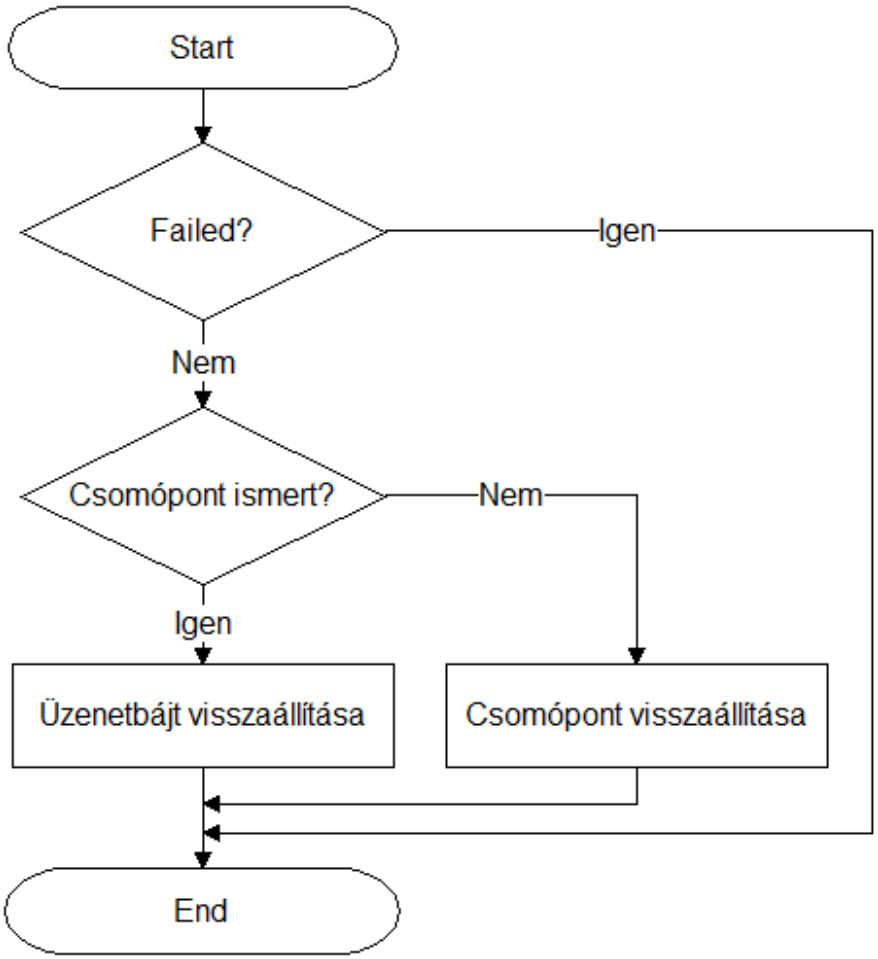
Az első lépés az inicializálás. Ennek során minden *tnodes[i]* értéket, és *failed[i]* értéket 0-ra állítunk be, *needed[i]* kezdeti értéke pedig 65536 lesz, mert ez egy olyan érték, amelyet egy sorszám már nem vehet fel, és így kezdeti értékként felhasználható annak jelzésére, hogy jelenleg érvénytelen a mező értéke. Ez után beállítjuk a véletlenszámgenerátort abba az állapotba, amelyet a küldő használt ezen blokk esetén az LDPC kódolás során. Az állapot beállítása azért lehetséges, mert a kapcsolat felépítés során a másik oldaltól megkaptuk a másik oldal véletlenszámgenerátorának kezdeti állapotát, amelyből előállíthatjuk a megfelelő állapotot.

A második lépés az első lépés során inicializált mezők értékét állítja be a jelenlegi legjobb ismert értékekre. Ez a lépés teljesen ugyanúgy történik mint a 3.9. ábrán láthattuk az LDPC kódolás során, kivéve a csomópont beállítása lépést. Ennek az oka, hogy a kódolás során ismertük azt az értéket, amelyet az ellenőrző csomópont beállításánál fel kellett használnunk, mert ez a felhasználó által küldendő üzenet egyik bájtja volt, amelyet a küldés során ismerünk. Az LDPC dekódolás esetén ezzel szemben lehetséges, hogy nem ismerjük ezt az értékét, mert az LT dekódolás nem tudta visszaállítani az adott bájtot az üzenetben. Az új csomópont beállítása lépést a 3.19. ábrán láthatjuk. Abban az esetben, ha az üzenetbájtot sikerült visszaállítani, tehát értéke ismert, akkor a *tnodes[i]* értékéhez hozzáadjuk (XOR) az üzenetbájt értékét, ugyanis ez az üzenetbájt a generálás során az adott ellenőrző csomópont egyik szomszédja volt. Ha az üzenetbájtot nem sikerült visszaállítani, tehát értéke nem ismert, akkor két eset lehetséges. Ha eddig még nem volt az adott ellenőrző csomópont esetén olyan üzenetbájt, amelynek értéke ismeretlen lett volna (ennek vizsgálatát jelzi a kudarc az ábrán), akkor még lehetséges az, hogy ez az ellenőrző csomópont felhasználható lesz a dekódolás során, mert lehet, hogy ezt az ellenőrző csomópontot sikeresen visszaállította az LT dekódolás, így értékét ismerjük, ezért a *needed[i]* értékét beállítjuk az egyetlen ismeretlen üzenetbájt sorszámára. Ellenkező esetben, ha már előzőleg találtunk egy ismeretlen üzenetbájtot, akkor már biztos, hogy nem tudjuk az adott ellenőrző csomópontot felhasználni a dekódoláshoz, mert ennek szomszédai között több ismeretlen is van. Ilyenkor a *failed[i]* értékét állítjuk be, és ezzel jelezzük, hogy az adott ellenőrző csomópontot nem tudjuk felhasználni a harmadik lépés során.



3.19. ábra. A 2. lépés során használt "csomópont beállítása"

A harmadik lépés esetén a 2. lépés során beállított értékek alapján elvégezzük a tényleges dekódolást, ha lehetséges. A lépéseket a 3.20. ábrán láthatjuk, egyetlen ellenőrző csomópont esetén feltüntetve a lépéseket. A dekódolás egy adott ellenőrző csomópont esetén akkor lehetséges, ha a *failed[i]* az adott ellenőrző csomópont esetén nincs beállítva. Ellenkező esetben az adott ellenőrző csomópontot nem tudjuk felhasználni. Egy adott ellenőrző csomópontot tekintve, ha lehetséges a dekódolás, akkor kétféle eset fordulhat elő attól függően, hogy az ellenőrző csomópont értéke ismert, vagy sem. Az első eset az, amikor az ellenőrző csomópontot ismerjük, és egyetlen olyan üzenetbájtot nem ismerünk, amely az adott ellenőrző csomópont szomszédja. Ilyenkor az adott ellenőrző csomópont, és a *tnodes[i]* értékének összegeként (XOR) megkapjuk az ismeretlen üzenetbájt értékét. A második eset az, amikor magát az ellenőrző csomópontot nem ismerjük, de minden szomszédját ismerjük. Ilyenkor az ellenőrző csomópont értékét tudjuk visszaállítani, amely a éppen a *tnodes[i]* értékkel fog megegyezni.



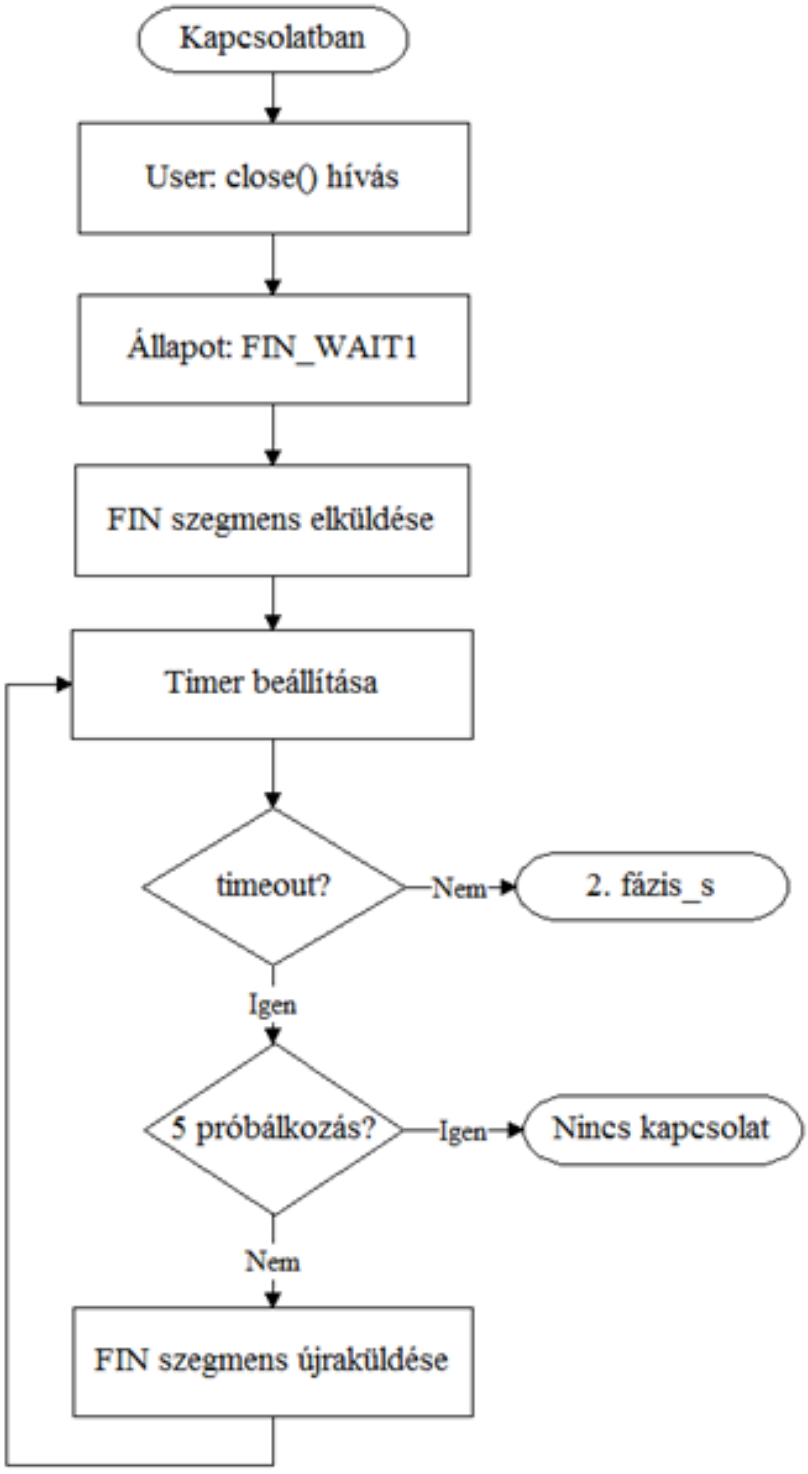
3.20. ábra. A 3. lépés, egyetlen ellenőrző csomópontra vonatkoztatva

Az LDPC dekódolás 3. lépése során végignézünk minden ellenőrző csomópontot, és az előzőleg leírtak szerint cselekszünk minden egyes ellenőrző csomópont esetén. Az LDPC dekódolás során szintén számolom azt, hogy hány üzenetbájtot sikerült visszaállítani, és ez lesz az LDPC dekódolást végző függvény visszatérési értéke.

3.10. Kapcsolatbontás

Ebben a fejezetben a kapcsolatbontás folyamatát fogom ismertetni. A kapcsolatbontás a TCP protokollnál használt módszerhez hasonlóan működik, és a kapcsolat felépítésnél már tárgyalt időzítők segítségével garantálható a megbízhatóság. A kapcsolatbontásnak szintén három lépését különböztetem meg.

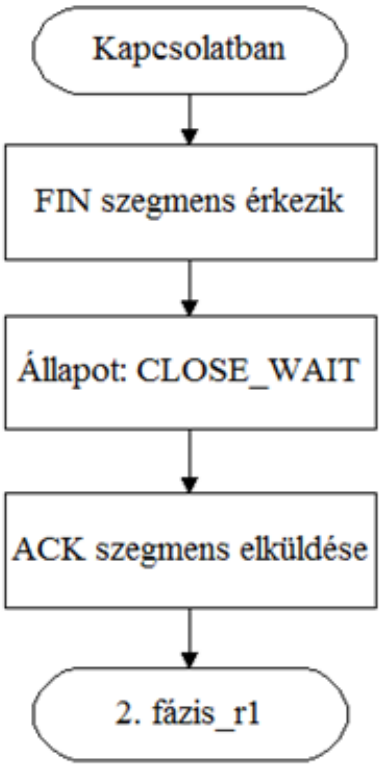
A kapcsolatbontás első lépése a 3.21. ábrán látható módon történik. Ennek során az oldal, amely bontani szeretné a kapcsolatot egy *FIN* szegmenst küld a másik oldalnak.



3.21. ábra. A kapcsolatbontás 1. lépése

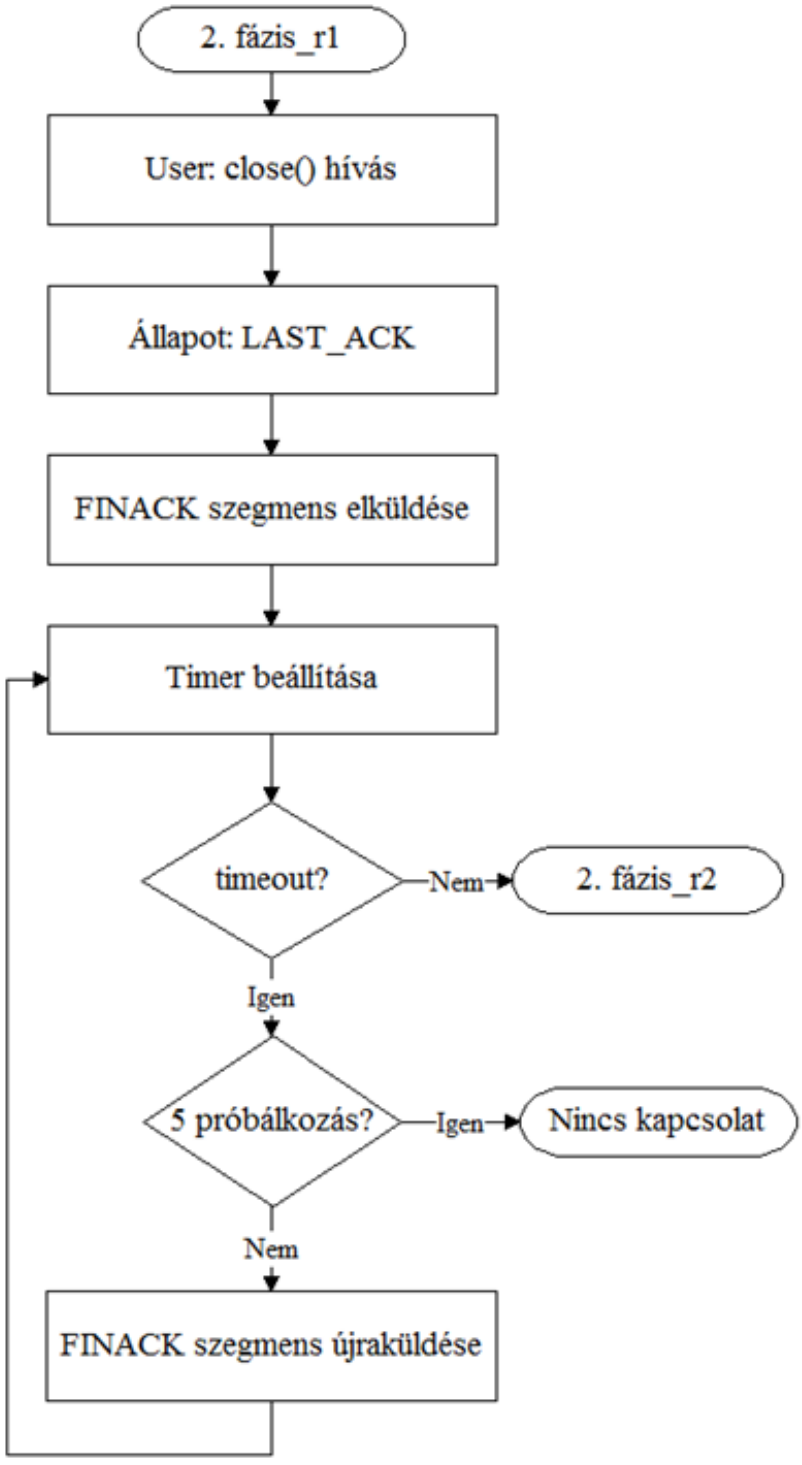
Az első lépés során a küldő állapota *FIN_ WAIT1* állapotra változik. Arra az esetre, ha a szegmens elveszne, időzítőt használok. A *FIN* szegmens esetén is maximum 5 alkalommal történik újraküldés. Az ábrán feltételezem, hogy ha nem történik időtúllépés, akkor megérkezett a nyugta a *FIN* szegmensre. Ha az 5. újraküldés után sem érkezik nyugta, akkor a kapcsolat megszakad, és az erőforrások felszabadulnak.

A második lépést a 3.22. ábra mutatja be. Ezen lépés esetén a *FIN* szegmenst vevő oldal a feldolgozás után azt minden esetben nyugtázza *ACK* szegmens segítségével. Az ACK szegmens esetén nem használok időzítőt, hanem a *FIN* szegmens újraküldése miatt következik be az *ACK* szegmens újraküldése. Az *ACK* szegmens elküldése után *CLOSE_ WAIT* állapotba kerül az *ACK* szegmenst küldő oldal. Ezen szegmens feldolgozása során *FIN_ WAIT2* állapotba kerül a kapcsolatbontást kezdeményező oldal. Ha a *FIN* szegmenst vevő oldal még nem szeretné bontani a kapcsolatot, akkor ő még nem küld *FIN(ACK)* szegmenst, de a másik oldaltól érkező *FIN* szegmenst ekkor is nyugtázza a 3.22. ábrán látható módon.



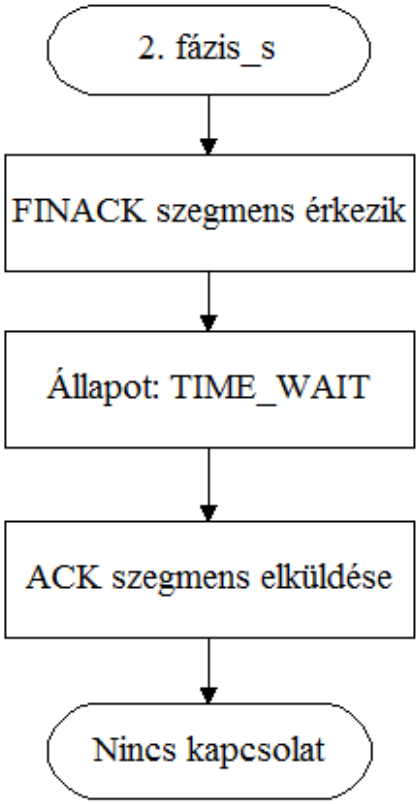
3.22. ábra. A kapcsolatbontás 2.a. lépése

Amikor a másik oldal is bontani szeretné a kapcsolatot, akkor *FINACK* szegmenst küld a kezdeményező oldalnak. Ekkor a *FINACK* szegmenst küldő oldal *LAST_ACK* állapotba kerül. A *FINACK* szegmenst szintén újraküldöm a *FIN* szegmenshez hasonlóan. Ezt a 3.23. ábrán láthatjuk.



3.23. ábra. A kapcsolatbontás 2.b. lépése

A harmadik lépést a 3.24. ábrán láthatjuk. Ekkor a *FINACK* szegmenst vevő oldal *ACK* szegmens segítségével nyugtázza azt, időzítő alkalmazása nélkül, és *TIME_ WAIT* állapotba kerül. Ez a kapcsolat megfelelő lezárása miatt szükséges, mert elképzelhető, hogy elveszik az *ACK* szegmens. Így a másik oldal *LAST_ ACK* állapotban marad, de mivel ő újraküldi a *FINACK* szegmenst, ezért képesek vagyunk észrevenni, hogy elveszett az *ACK* szegmensünk, és újraküldhetjük azt.



3.24. ábra. A kapcsolatbontás 3. lépése

Egy adott ideig *TIME_ WAIT* állapotban történő várakozás után az erőforrások felszabadulnak. Az *ACK* szegmenst vevő oldal az *ACK* szegmens feldolgozásával *CLOSE* állapotba kerül, és nála is felszabadulnak az erőforrások.

3.11. A P7 protokoll paraméterei

Ebben a fejezetben a P7 protokoll működését befolyásoló paramétereket ismertetem. Ezeket a paramétereket a felhasználói alkalmazás kívülről képes beállítani az általa kívánt értékre. A megfelelő értékre való beállítás után a megadott érték lesz az érvényes a kernel további működése során, tehát az új érték azonnal érvénybelép. Fontos, hogy az új érték hatókörre kizárólag az a kapcsolat lesz, amelyet az alkalmazás használt a beállítás során, tehát a többi létező, vagy új kapcsolatra továbbra is a kernel alapértelmezett értékeit használjuk. Öt fontos paramétert mutatok be, amelyeknek funkcióját a 3.2. táblázatban foglalom össze. Az első paraméterrel szabályozhatjuk a kernel által lefoglalt küldési, és vételi tárolók számát. A második paraméterrel megadhatjuk az adatok küldésénél alkalmazandó határ-

értéket (redundanciát). A harmadik paraméterrel a nyugtázást, a negyedik paraméterrel a kódolást, végül az ötödik paraméterrel pedig a dekódolást kapcsolhatjuk ki.

3.2. táblázat. A paraméterek jelentése

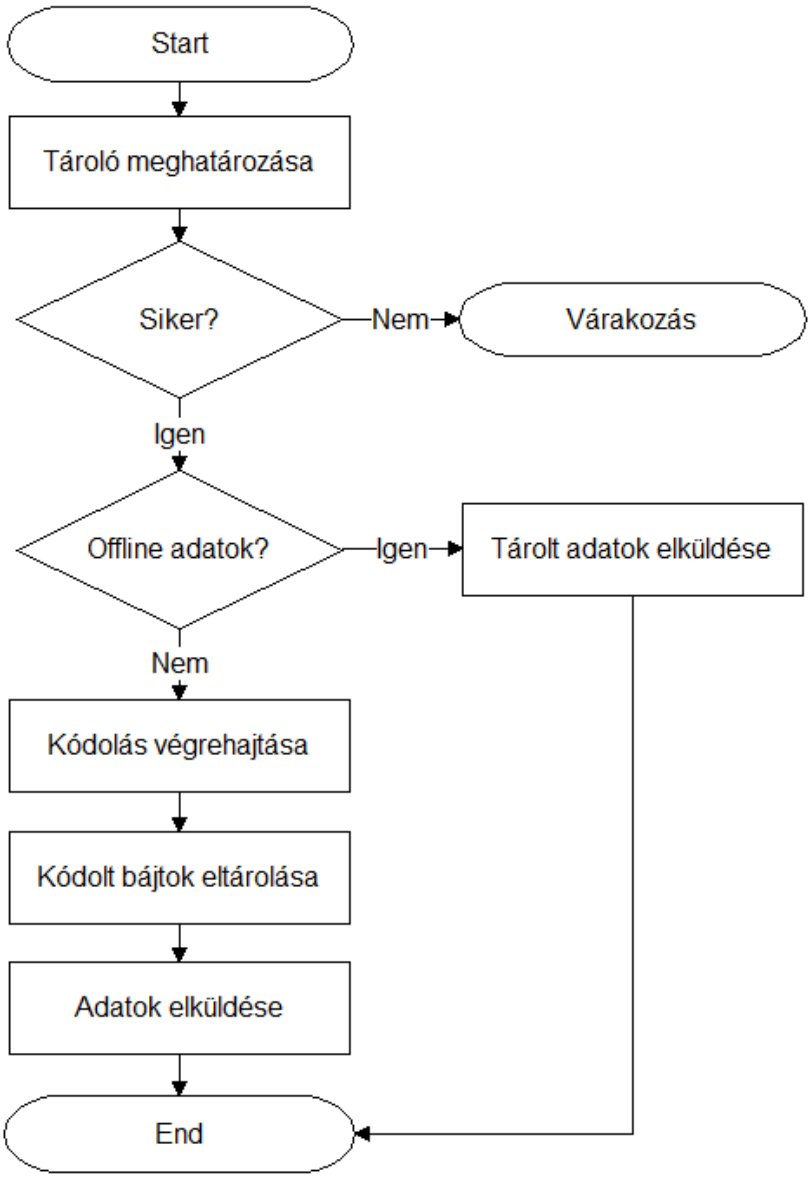
A paraméter neve	**A paraméter funkciója**
Maximális ablakméret	Az adatküldés során használt csúszóablak maximális méretének beállítása.
Redundancia	A blokkok küldése esetén használt határérték meghatározása.
Nyugtázás	A nyugtázás kikapcsolása/bekapcsolása a vevőnél.
Kódolás	Az adatok kódolásának kikapcsolása/bekapcsolása a küldőnél.
Dekódolás	Az adatok dekódolásának kikapcsolása/bekapcsolása a vevőnél.

Az első paraméter segítségével a küldés során használt maximális ablakméretet befolyásolhatjuk. A kapcsolat felépítése során egy adott mennyiségű küldési, és vételi tárolót foglalok le, ezeknek a számát meghatározhatjuk a felhasználói alkalmazás által kívülről. A paraméter beállításának tehát a kapcsolat felépítés megkezdése előtt van értelme, utána már a lefoglalt tárolókat alkalmazzuk az átvitel során, így számuk már nem változtatható meg. A tárolókat a kapcsolat bontása során szabadítjuk fel.

A következő paraméter az adatküldéshez kötődik. A küldés során bemutattam, hogy alapesetben 49 csomagot küldünk ki minden blokk esetén, amely így 69580 kódolt bájtot jelent blokkonként. Ez akkora mennyiség amely éppen elegendő a sikeres dekódoláshoz. Ha csomagvesztés is fellép az átvitel során, akkor a vevő nem kaphatja meg a megfelelő adatmennyiséget, és emiatt nem képes elvégezni a dekódolást. A paraméter segítségével tetszőleges értékre beállíthatjuk a 3.13. ábrán látható, küldési határértéket, így meghatározhatjuk a blokkonként elküldendő csomagok számát. A paraméter értékét a fellépő csomagvesztés értékének megfelelően érdemes beállítani, mert így biztosíthatjuk azt, hogy a vevő képes legyen a blokkok sikeres dekódolására.

A harmadik paraméter segítségével teljesen kikapcsolhatjuk a nyugtázást. Ebben az esetben a küldő egy adott blokk esetén a megfelelő mennyiségű csomag elküldése után azonnal felszabadítja a küldési tárolót, nem várakozik nyugtára. Ekkor az adatküldés ténylegesen maximális sebességgel történhet, így a nyugtázás lassító hatását elkerülhetjük. Azonban a nyugtázás nélküli üzemmód hátránya, hogy a nyugtázás fontos funkciója volt az is, hogy megvédte a vevőt a túl gyors adóktól, azaz áramlásszabályozást valósított meg, ugyanis az adók az ablakméreten felül nem küldhettek további adatot, amíg valamely nyugta meg nem érkezett.

A következő paraméter a kódolás kikapcsolására alkalmazható. Ebben az esetben a kernel csak a legelső blokk esetén végzi el a tényleges kódolást, majd a kódolt bájtokat, és a csomagok fejléce esetén használt változókat (a véletlenszám generátor állapotát) egy külön erre a célra használatos tárolóba helyezi el. A további blokkok küldésekor az eltárolt blokkhoz tartozó adatokat használja fel, tehát tartalmukat tekintve ugyanazokat a csomagokat küldi el újra, mint az első blokk esetén. Az előbb leírt módszerre a továbbiakban *offline kódolásként* fogok hivatkozni. Az offline kódolás esetén használt lépéseket a 3.25. ábrán láthatjuk.



3.25. ábra. Az offline kódolás folyamata

Az első lépés, a tároló meghatározása itt is szükséges, mert a blokkhoz tartozó sorszámot itt tároljuk el, és így tartjuk nyilván az ablak által tartalmazott blokkokat. Nyugta érkezése esetén pedig felszabadíthatjuk a megfelelő tárolót. Offline kódolás esetén a blokk sorszámán kívül itt mást nem tárolunk el. Abban az esetben, ha éppen az első blokk elküldése történik, akkor az offline kódolt adatok még nem állnak rendelkezésre, ezért végrehajtjuk a kódolást, majd ennek eredményét elmentjük egy külön tárolóba. Végül pedig megtörténik az első blokk esetén a csomagok elküldése. Ha a jelenlegi blokk nem az első blokk, akkor már előzőleg történt küldés, így már rendelkezésre állnak az offline kódolt adatok (amelyek az első blokk küldésekor keletkeztek). Ilyenkor nem történik kódolás, hanem az eltárolt adatokat küldjük újra. A csomagok fejlécében ekkor természetesen más sorszám szerepel, de a csomagok ugyanazokat az adatokat, kódolt bájtokat tartalmazzák, mint az első blokk küldése során. Ennek a megoldásnak az az előnye, hogy ilyen módon vizsgálható a kódolás

hatása a küldési sebességre.

Végül, az ötödik paraméter a dekódolás kikapcsolását teszi lehetővé. Ebben az esetben a 3.14. ábrán látható folyamat úgy módosul, hogy a blokk sikeres meghatározása után a csomag tartalmának eltárolása helyett a csomagot eldobjuk, és a hozzá tartozó tárolóban csak a beérkezett adatmennyiséget tartjuk nyilván. Ha megérkezett a dekódoláshoz szükséges adatmennyiség, akkor értesítjük az alkalmazást, amely miután nyugtát küldött a blokkra vonatkozóan, felszabadítja a tárolót. A felhasználói alkalmazás ebben az esetben 0 beérkezett bájtról kap visszajelzést, mert ennyi bájtnyi adatot dekódoltunk. Ez a megoldás az előző paraméterhez hasonlóan szintén azért előnyös, mert használatával lehetővé válik a dekódolás hatásának vizsgálata.

Láthatjuk, hogy az egyes paraméterek segítségével a felhasznált komponenseket (kódolás, dekódolás, és nyugtázás) egymástól elkülönítve vizsgálhatjuk.

4. fejezet

Teszthálózati mérések

Ebben a fejezetben a P7 protokollal elvégzett méréseket, és az eredményeket ismertetem. Először bemutatom a mérések során felhasznált programokat. Itt kitérek az egyes programok alapvető működésére, és paramétereinek jelentésére. Majd láthatjuk a hálózati topológiákat, és az alkalmazott beállításokat a számítógépek esetén. Ez után ábrák, és magyarázatok segítségével illusztrálom az egyes mérések eredményeit. Végül, a protokollal kapcsolatos jövőbeli terveket írom le.

4.1. A felhasznált programok

Ebben az alfejezetben a mérések során felhasznált eszközöket ismertetem. Az első program amelyet bemutatok az *iperf*, amelyet a TCP protokollal történő mérésekhez használtam. Ez után a P7 protokollal elvégzett mérésekhez felhasznált kliens, és szerver program működését ismertetem, amely programokat C nyelven készítettem el. És végül, az utolsó eszköz, amit felhasználtam a *tc*, amelyet a különböző hálózati paraméterek emulációja során alkalmaztam.

4.1.1. Iperf

Az iperf [29] egy olyan eszköz, amellyel különböző paraméterek mellett vizsgálhatjuk a TCP és a UDP protokollal elért *átbocsátóképességet*, és megkaphatjuk az *adatátvitel idejét* is. A teszteléshez szükség van egy kliensre, és egy szerverre. Az iperf mindkét üzemmódban képes működni, és paraméterekkel határozhatjuk meg azt, hogy szervert, vagy klienst szeretnénk-e indítani. Az adatküldés a klienstől a szerver felé történik.

Az alábbiakban bemutatom a fontosabb paramétereket, ezeknek egy részét csak a TCP, másik részét csak a UDP esetén használhatjuk.

- -p: A szerver által használt portot adja meg.
- -u: Az alapértelmezett TCP helyett UDP módot állít be.
- -s: Bekapcsolja a szerver módot.
- -c: Bekapcsolja a kliens módot.

- -Z: A TCP torlódásszabályozási algoritmusát állítja be.
- -w: A TCP ablakméretét adhatjuk meg.
- -M: A TCP MSS értékét határozza meg.
- -b: A UDP küldési sebességét adja meg.
- -n: Megadja az átvinni kívánt adatmennyiséget bájtokban.
- -t: Az átvitel idejét határozza meg.
- -l: Az olvasási, illetve az írási buffer méretét adja meg.

A felsorolt paramétereken kívül számos egyéb paraméter is használható. A mérések elvégzése során csak a TCP üzemmódot használtam, és az alkalmazott beállításokat később, a konkrét méréseknél fogom ismertetni.

4.1.2. P7 kliens és P7 szerver

A P7 kliens, és a P7 szerver lehetőséget nyújt az új protokollal történő mérésekhez. A programok működése az előző fejezetben bemutatott iperfhez hasonlóan történik, a kapcsolat felépítése után megkezdődik az adatküldés, amely a klienstől a szerver felé történik. Mindkét program esetén lehetőségünk van a P7 protokoll paramétereinek beállítására, így az adatátvitel során a protokoll működése a megadott paramétereknek megfelelően történik. Az adatátvitel befejeztével a kapcsolatot elbontjuk.

A továbbiakban ismertetem a kliens, és a szerver esetén használható paraméterek jelentését. Egyes paramétereket csak a kliens, másokat pedig csak a szerver képes beállítani, az adott paraméter funkciójának megfelelően.

A P7 kliens paraméterei:

- -s: A szerver IP címét, vagy hosztját adja meg.
- -p: A szerver által használt portot határozza meg.
- -o: Bekapcsolja a megadott fájlba történő naplózást.
- -w: Az adott kapcsolatra vonatkozóan az ablak méretét állítja be a megadott értéknek megfelelően.
- -c: A kliens által elküldött adatmennyiséget határozza meg.
- -r: Az adatok küldése során használt redundacia paraméter értékét határozza meg.
- -e: Kikapcsolja a kódolást. Ebben az esetben offline kódolás történik.
- -a: Kikapcsolja a nyugtázást. Értéke mindig ugyanaz, mint amit szerver esetén állítottunk be.

A P7 szerver paraméterei:

- -p: A szerver által használt portot határozza meg.
- -o: Bekapcsolja a megadott fájlba történő naplózást.
- -w: Az adott kapcsolatra vonatkozóan az ablak méretét állítja be a megadott értéknek megfelelően.
- -c: A szerver által várt adatmennyiséget határozza meg.
- -e: Kikapcsolja a kódolást. Értéke mindig ugyanaz, mint amit kliens alkalmazás esetén használtunk.
- -a: Kikapcsolja a nyugtázást.
- -d: Kikapcsolja a dekódolást.

A kliens, és a szerver esetén az elvárt működés érdekében szükséges figyelembe venni, hogy a kliens ablakmérete nem haladhatja meg a szerver esetén megadott ablakméretet, mert ebben az esetben a szerver oldal nem képes eltárolni, és így dekódolni az ablakméretén felül érkező blokkokat. Valamint, az elküldött, és a várt adatmennyiség esetén ugyanazt az értéket célszerű alkalmazni. Azt is láthatjuk, hogy mindkét program esetén szerepel a kódolásra (e), és a nyugtázásra vonatkozó paraméter (a). Az első esetben (kódolás) ez azért szükséges, mert az offline kódolás esetén minden blokk dekódolása során ugyanazokat a véletlenszámokat kell alkalmaznia a szervernek az LDPC dekódolás során, és ezt a paraméter megfelelő értékre való beállításával érhetjük el. A második esetben (nyugtázás) pedig így érhetjük el, hogy a kliens egy blokk elküldése után ne várakozzon nyugtára, amely lehetővé tenné a küldési tároló felszabadítását, hanem a küldés után azonnal képes legyen felszabadítani a megfelelő tárolót.

A kliens program a küldendő adatok meghatározásához felhasználhatja egy adott fájl tartalmát, ekkor a fájlban található adatokat küldi el. Ezen kívül képes arra is, hogy a felhasználó által, közvetlenül a küldés megkezdése előtt megadott karaktersorozatot küldi el úgy, hogy ezen karaktersorozatot minden küldött blokk esetén egy egyedi karaktersorozattá alakítja, és ezt a megváltoztatott adatot küldi el. A mérések során a második módszert alkalmaztam, mert a fájlműveletek végrehajtása lassíthatja a protokoll működését.

Mindkét program meghatározza az *adatátvitel idejét*, és az elért *átbocsátóképességet*. Az adatátvitel idejének meghatározása során a kapcsolatfelépítés, és kapcsolatbontás idejét nem számítjuk. Az átbocsátóképesség kiszámítása esetén az adatmennyiség meghatározásánál a csomagokban található hasznos adatokkal számolunk, a fejrészt nem számítjuk. A szerver program ezen kívül képes a lekérdezni a kernel által számított *dekódolási időt*, és az LT dekódolásról szóló fejezetben ismertetett, a beérkezett kódolt bájtoknak a megfelelő láncolt listákba való *rendezési idejét* is.

A mérések során alkalmazott paramétereket, beállításokat szintén később, a konkrét méréseknél fogom ismertetni.

4.1.3. tc

A tc [30] elnevezés az angol nyelvű, *traffic control* kifejezés rövidített formája, amely kifejezés egy adott router, csomópont által a csomagok küldése és fogadása során alkalmazott sorbaállítási rendszereket, mechanizmusokat jelenti. Ide tartozik például annak eldöntése, hogy mely bejövő csomagokat milyen sebesség mellett fogadjunk el egy adott hálózati interfész esetén. A küldés esetén pedig annak meghatározása, hogy mely csomagokat milyen sorrendben, és milyen sebességgel küldjünk egy interfész esetén.

A forgalomszabályozás alapvető elemei:

- Formálás (Shaping): Egy forgalom sebességének szabályozása, amely a sebesség korlátozásán túl a börsztös forgalom simítására is használatos, így kedvezőbb hálózati viselkedést biztosíthatunk.

- Ütemezés (Scheduling): A csomagok továbbítási sorrendjének meghatározása. Az interaktivitást igénylő forgalmakat előnyben részesíthetjük, illetve sávszélességet biztosíthatunk az ezt igénylő forgalomnak.

- Osztályozás (Classifying): A forgalom adott szempontok alapján történő osztályozása. Ezeket az osztályokat különböző módon kezelhetjük. Osztályozás történhet például a csomagok megjelölésével.

- Felügyelet (Policing): A forgalom korlátozása úgy, hogy egy adott sávszélességet ne haladjon meg. A korlátot meghaladó forgalmat meghatározott módon kezelhetjük. Például eldobhatjuk, vagy újra osztályozhatjuk.

- Eldobás (Dropping): Egy adott korlátot meghaladó forgalom, vagy egy meghatározott osztályba tartozó csomagok eldobása.

- Megjelölés (Marking): Egy csomag megváltoztatása, megjelölése. Ez például történhet a DSCP használatával [31] egy csomag esetén. A további routerek ezt a jelölést figyelembe vehetik.

Ezeket az alapvető elemeket a tc *linux komponensekre* képezi le, és ezeknek a komponenseknek a használatával valósíthatjuk meg a kívánt szabályozást. A mérésekhez a tc egy kiterjesztését, a *netem*-et használtam, amely a Linux kernel 2.6.7 verziójától kezdve a kernel részeként érhető el, és különböző hálózati paraméterek emulációját teszi lehetővé. A netem segítségével emulálható paraméterek:

- Késleltetés: Megadhatjuk a kívánt késleltetés értékét, amelyhez egy változó komponenst is hozzáadhatunk, ennek értéke az előző értékektől is függhet, tehát korrelációt is beállíthatunk. Meghatározhatjuk a késleltetés eloszlását is, alapértelmezés szerint elérhető eloszlások: *uniform*, *normal*, *pareto*, *paretonormal*, de lehetőség van ezeket az eloszlásokat továbbiakkal bővíteni is.

- Csomagvesztés: A kívánt csomagvesztés értékét adhatjuk meg, százalékos értékként. A késleltetésnél ismertetett lehetőségeket itt is alkalmazhatjuk.

- Csomagok duplikálása: Az előző paraméterhez hasonlóan itt is százalékos értéket adhatunk meg.
- Csomagok tönkretétele: Véletlen zaj emulációjához használható. Egy csomagnak egy véletlen pontján okozhatunk bithibát. Az érintett csomagok számát itt is százalékos értékként adhatjuk meg.
- Csomagok átsorrendezése: A csomagok sorrendjét megváltoztathatjuk. Itt megadhatunk egy fix eltolást is, ebben az esetben minden n-edik csomag rossz sorrendben kerül továbbításra. A másik megoldás, hogy százalékos értéket adunk meg, és ennek megfelelően történik az átsorrendezés. Ezen kívül itt is használhatunk korrelációt.

Az elvégzett mérések során a késleltetés, és a csomagvesztés hatását vizsgáltam meg. A késleltetés esetén egy meghatározott, fix értéket állítottam be, a csomagvesztés esetén pedig normális eloszlást alkalmaztam. A paraméterek konkrét értékeit a méréseknél fogom bemutatni.

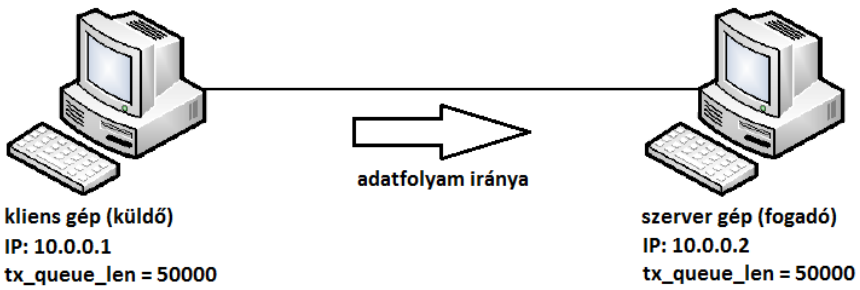
4.2. Topológiák

A méréseket a Távközlési és Médiainformatikai Tanszék IB.213-as laborjában végeztem el 2011. novemberében. Itt különböző hálózati topológiákat alakítottam ki, amelyeknek részleteit ebben a fejezetben adom meg. A *hálózati topológia* egy olyan hálózati elrendezést jelent, amelynek esetén a különböző elemek (linkek, csomópontok, stb.) kapcsolatait, összeköttetéseit megadjuk.

A méréseket két különböző topológián végeztem el. Az egyik elrendezés a *két számítógépet tartalmazó hálózat*, a másik pedig a *Dumbbell topológia* volt. A következő alfejezetekben ezt a két topológiát, és a konkrét megvalósításukat mutatom be.

4.2.1. Két számítógépből álló topológia

Az első topológia a legegyszerűbb, két számítógépet tartalmazó elrendezés volt, ahol a két számítógép egymással közvetlen összeköttetésben állt. Az elrendezést a 4.1. ábrán láthatjuk.



4.1. ábra. Két számítógépet tartalmazó topológia

Az ábrán látható a küldő, és a vevő oldal IP címe, és a *tx_queue_len* változó értéke. A *tx_queue* a kernelben található *transmit queue* elnevezésű adatszerkezetre utal. Minden hálózati eszközhöz tartozik egy transmit queue, amelyből a megfelelő eszközmeghajtó (driver) viszi át a csomagokat a hardverhez tartozó sorba. A transmit queue méretét befolyásolhatjuk, ezt a változót ebben az esetben a P7 protokoll nagysebességű küldése miatt 50000 csomag értékűre állítottuk be.

A kliens, és a szerver számítógép fontosabb paramétereit a 4.1. táblázatban láthatjuk, a felhasznált hardver komponensek mindkét gép esetén azonosak.

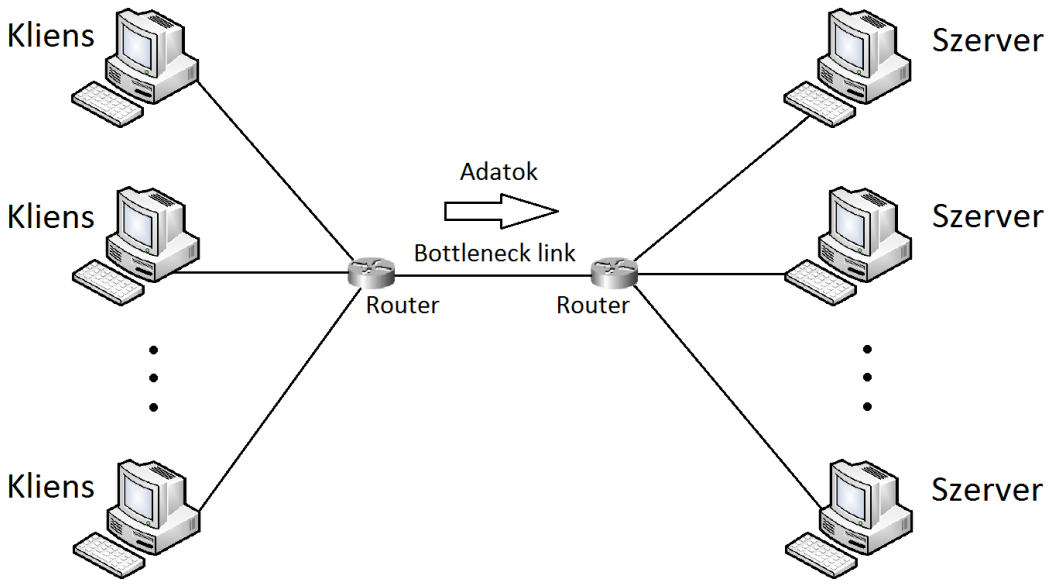
4.1. táblázat. A kliens, és a szerver konfigurációja

A hardver neve	**A hardver paraméterei**
Processzor	Intel® Core™ 2 Duo Processor E8400
Memória	2 GB
Hálózati kártya	1 Gbps

Ezen a topológián nem állítottam be sávszélesség korlátozást, csomagvesztést, késleltetést, vagy további paramétert, hanem a különböző TCP verziókat, és a P7 protokoll működését vizsgáltam meg úgy, hogy egyszerre mindig csak egyetlen folyam volt a hálózaton. Így bemutatom majd, hogy milyen teljesítményt érnek el ezek a protokollok ideális esetben, illetve a P7 protokoll esetén látni fogjuk azt is, hogy a protokoll paramétereinek különböző beállításai hogyan befolyásolják az elért teljesítményét.

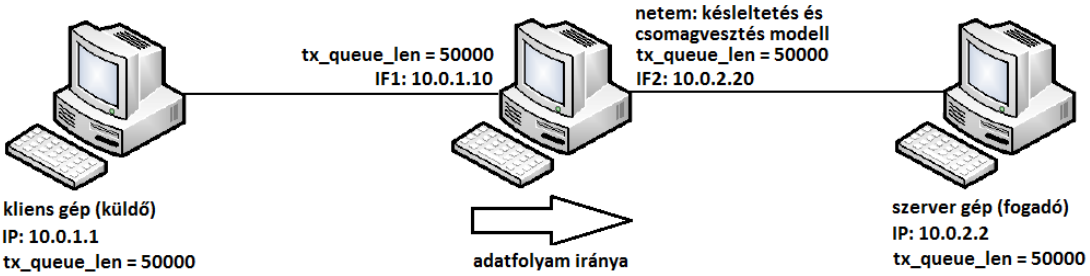
4.2.2. Dumbbell topológia

A második topológia a Dumbbell topológia volt, amelynek az elvi felépítését a 4.2. ábrán láhatjuk.



4.2. ábra. A Dumbbell topológia elvi felépítése

Az ábra bal oldalán láthatjuk a klienseket, a jobb oldalon a szervereket. Az adatátvitel a kliens gépektől a szerver gépek felé történik. Az ábrán szereplő két router között elhelyezkedő link jelenti a hálózatban a szűk keresztmetszetet, ahol az egyes folyamok versenyezhetnek a sávszélességért. A 4.3. ábrán láhatjuk a topológia tényleges megvalósítását.



4.3. ábra. A Dumbbell topológia megvalósítása

A középső, router gépen két interfészt használtam. Ebben az esetben a kliens, és a szerver számítógép konfigurációját a 4.2. táblázat, a router konfigurációját pedig a 4.3. táblázat foglalja össze.

4.2. táblázat. A kliens, és a szerver konfigurációja

A hardver neve	**A hardver paraméterei**
Processzor	Intel® Core™ 2 Duo Processor E8400
Memória	2 GB
Hálózati kártya	1 Gbps

4.3. táblázat. A router konfigurációja

A hardver neve	**A hardver paraméterei**
Processzor	Intel® Core™ i3-530 Processor
Memória	2 GB
Hálózati kártya	1 Gbps

Láthatjuk, hogy a kliens, és a szerver paraméterei megegyeznek az előző topológia esetén bemutatott paraméterekkel, de a router számítógép konfigurációja ettől eltérő. Az alkalmazott hálózati kártyák névlegesen minden esetben azonos, 1 Gbps sebességűek.

Ezen a topológián emuláltam különböző késleltetés, és csomagvesztés értékeket is a középső, router gépen. Látható, hogy a netem-et a második interfész esetén alkalmaztam, és a beállított hálózati paraméterek a klienstől származó, szerver felé irányuló, kimenő forgalmat érintették, de a visszairányuló forgalmat nem. Így a P7 protokoll esetén a nyugták nem veszhettek el, és nem is késhettek, csak az átvitt adatokat érintették a paraméterek. Az elvégzett mérések során a különböző TCP verziók, és a P7 protokoll teljesítményét vizsgáltam meg az adott paraméterek esetén.

4.3. Az elvégzett mérések

Ebben az alfejezetben az elvégzett méréseket, és a kapott eredményeket mutatom be. Először a két számítógépet tartalmazó hálózaton, majd a Dumbbell topológián történt mérések eredményeit láthatjuk. Az eredményeket ábrákon, és táblázatokon keresztül mutatom be a hozzájuk kapcsolódó magyarázatok segítségével.

Az egyes hálózati topológiákon elvégzett mérések során használt paramétereket, és értékeiket a 4.4. táblázat foglalja össze. A táblázatban látható paraméterek esetén a megvizsgált értéktartományt adtam meg, tehát nem próbáltam ki a felsorolt értékek összes lehetséges kombinációját. A felhasznált konkrét értékeket a méréseknél láthatjuk.

4.4. táblázat. A megvizsgált paraméterek

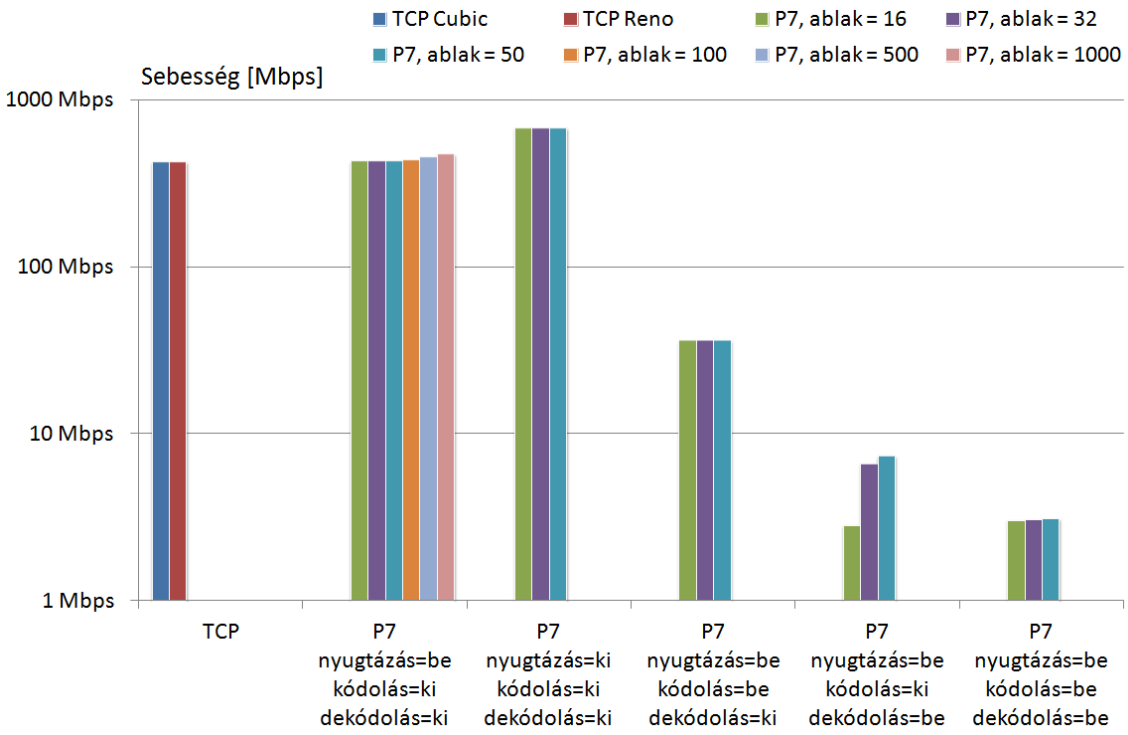
Topológia	**A paraméter neve [mértékegysége]**	**A paraméter értékei**
Mindkettő	Maximális ablakméret [blokk]	16, 32, 50, 100, 500, 1000
Mindkettő	Redundancia [csomag]	49, 60, 70, 150
Mindkettő	Nyugtázás	Kikapcsolva, Bekapcsolva
Mindkettő	Kódolás	Kikapcsolva, Bekapcsolva
Mindkettő	Dekódolás	Kikapcsolva, Bekapcsolva
Dumbbell	Késleltetés [ms]	5, 100
Dumbbell	Csomagvesztés [%]	0.1, 1, 10, 30

4.3.1. Két számítógépből álló topológia

Ezen a topológián a TCP Cubic, a TCP Reno, és a P7 protokoll működését vizsgáltam meg ideális esetben. Egyszerre mindig csak egyetlen folyam volt a hálózaton. A P7 protokoll esetén megnéztem, hogy a protokoll paramétereinek különböző értékei hogyan befolyásolják a teljesítményt. A négy vizsgált paraméter a kódolás, a dekódolás, a nyugtázás, és az ablakméret voltak. A továbbított adatmennyiség minden esetben azonos, 700 MB volt, amely közelítőleg 1 CD lemezen eltárolható mennyiségű adatot jelent. A 4.4. ábrán láthatjuk az elért átviteli sebességet, és az alkalmazott paraméterek értékeit.

A vízszintes tengelyen látható, hogy mely protokoll szerepel a diagram megfelelő oszlopában csoportosítva, és a P7 protokoll esetén azt is feltüntettem, hogy milyen paramétereket állítottam be. Az ábra felső részén látható az is, hogy milyen ablakméreteket használtam. Egy oszlopon belül csoportosítva láthatjuk a P7 protokoll eredményeit a különböző ablakméretek mellett.

Abban az esetben, ha offline kódolás történt, és nem volt dekódolás, megközelítjük a TCP teljesítményét. Ekkor az ablakméret növelése kis mértékben növeli az átviteli sebességet. Ennek az az oka, hogy a nyugtázás blokkonként történik, és a küldőt korlátozza az, hogy nyugtára kell várakoznia, több adatot is képes lenne elküldeni, amíg a nyugta megérkezik. Nagyobb ablakméret esetén nyugtára való várakozás nélkül több adatot is elküldhetünk, így ez növeli a sebességet. Az ablakméret esetén korlátot jelentett az, hogy 1GB kernel memória áll rendelkezésre. Ezt az értéket nagyobb ablakméret beállítása esetén könnyen elérhetjük, ezért a maximális beállított ablak 1000 blokk méretű volt.



4.4. ábra. A vizsgált protokollok átviteli sebessége

A leírtakból következik az is, hogy az átviteli sebesség tovább növekszik, ha teljesen kikapcsoljuk a nyugtázást. Ebben az esetben a nagyobb ablakméret beállítása már nem tudja tovább növelni a sebességet, mert mindig maximális sebességgel történik az adatküldés. Ekkor az elért sebesség mindhárom vizsgált ablakméret esetén megegyezett.

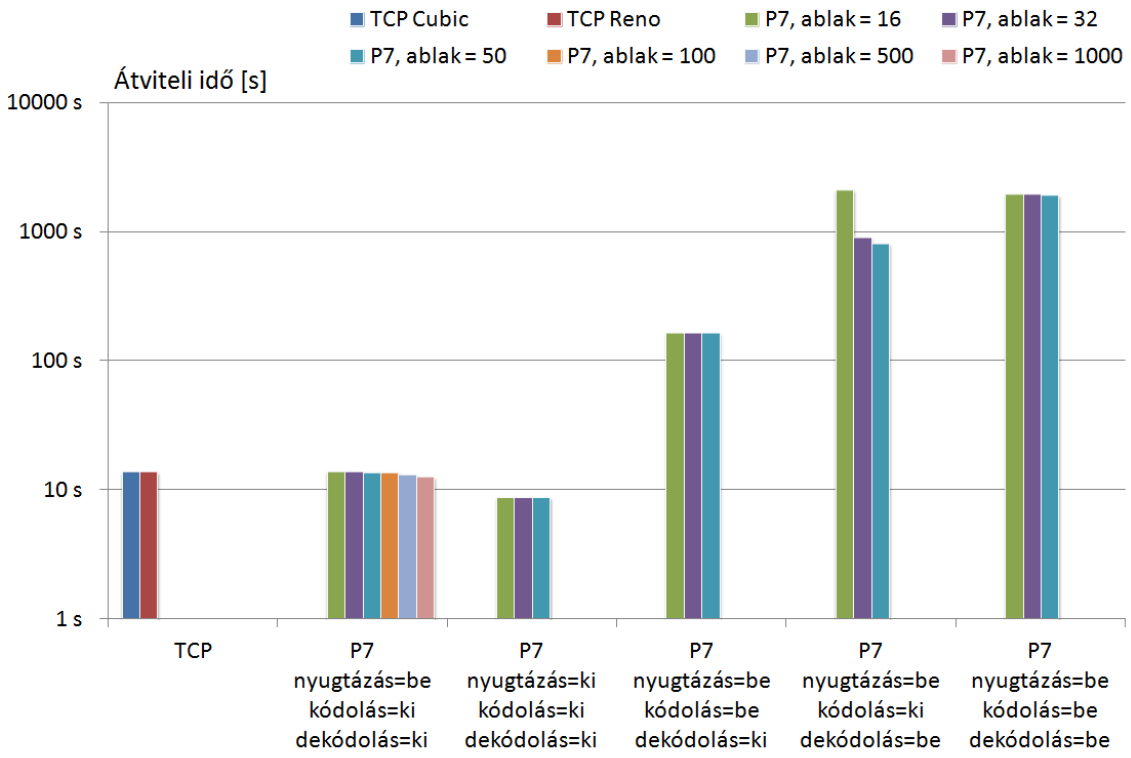
Ha bekapcsoljuk a kódolást, akkor a sebesség visszaesik, mert a kódolást ekkor már minden blokk esetén végrehatjuk, és ez több időt igényel az offline kódoláshoz képest, amikor az első blokk kivételével nem történik kódolás, hanem csak az eltárolt adatokat küldjük újra.

A következő esetben, tehát ha offline kódolás történik, és bekapcsoljuk a dekódolást, akkor láthatjuk, hogy az előző mérésekhez képest még inkább visszaesik az átviteli sebesség. Ebből azt a következtetést lehet levonni, hogy a dekódolás még időigényesebb, mint a kódolás. Azt is megfigyelhetjük, hogy különböző ablakméreteknél jelentősen eltér az elért sebesség, azonban ezt most nem tulajdoníthatjuk a nyugtázásnak, mert a dekódolás időigényéhez képest a nyugták késleltetése nagyságrendekkel kisebb. Ebben az esetben azért van eltérés a sebességek között, mert egy blokk dekódolásának időigénye nagymértékben függ attól, hogy milyen véletlenszámokat kell alkalmazni a dekódolás során. A kódolásról szóló fejezetben bemutattam, hogy a felhasznált véletlenszámok határozzák meg azt, hogy milyen konkrét összeköttetéseket hozunk létre a küldendő üzenet bájtjai között. Ha jelentősebb számban vannak azok a kódolt bájtok, amelyeknek sok szomszédos üzenetbájtja van, akkor ez időigényesebb dekódolást jelenthet. Offline kódolás esetén természetesen minden blokk esetén ugyanazokat a véletlenszámokat alkalmazzuk. Így, ha a küldő olyan véletlenszámokat alkalmazott a kódolás során, amelyek időigényesebb dekódolást eredmé-

nyeznek, akkor ez minden blokk dekódolása esetén tapasztalható lesz. Ezzel szemben, ha a küldő olyan véletlenszámokat alkalmaz, amelyek gyorsabb dekódolást eredményeznek, akkor minden blokk esetén gyorsabb lesz a dekódolás folyamata is.

Az utolsó esetben, tehát ha kódolás, és dekódolás is történik, akkor láthatjuk, hogy a sebességkülönbség is eltűnik, mivel ekkor már minden blokk esetén különböző véletlenszámokat alkalmazunk a kódoláshoz, és a dekódoláshoz.

A 4.5. ábrán az előző ábrához kapcsolódóan az átvitel idejét láthatjuk, itt az értékek összhangban vannak az elért sebességgel.



4.5. ábra. Az adatátviteli ideje

A szerver alkalmazás képes lekérdezni a rendezési művelet, és a dekódolás összesített idejét a kerneltől, tehát ezek az időértékek a teljes adatátvitelre vonatkoznak. A 4.5. táblázatban láthatjuk a megfelelő értékeket offline kódolás esetén, valamint offline kódolás nélkül.

4.5. táblázat. A rendezés, és dekódolás időigénye

Ablakméret	Kódolás	Rendezési idő	Dekódolási idő
16	Ki	196.92 s	1896.6 s
32	Ki	153.296 s	741.82 s
50	Ki	146.95 s	659.12 s
16	Be	196.03 s	1752.8 s
32	Be	198.93 s	1739.952 s
50	Be	199.51 s	1706.82 s

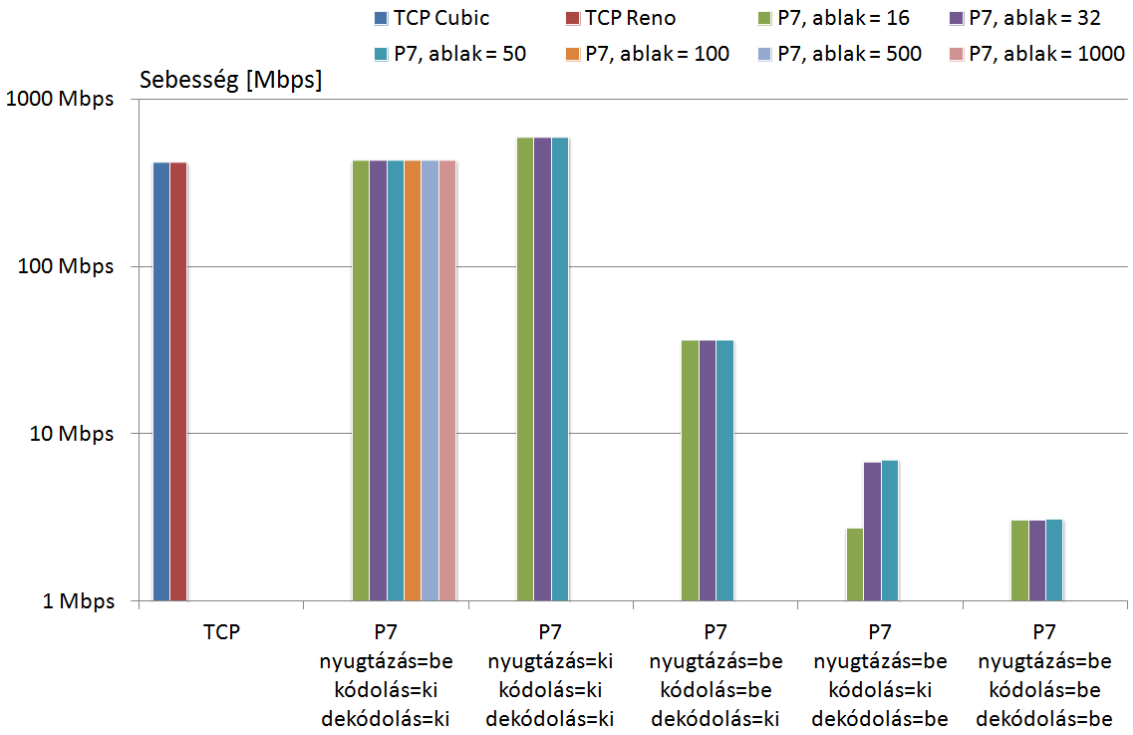
A táblázat alapján azt a következtetést lehet tenni, hogy abban az esetben, ha történik

dekódolás, akkor a teljes átvitel idejének nagy részét a dekódolás, egy ennél kisebb részét pedig a rendezés teszi ki. A megadott rendezési, és dekódolás időértékek nem pontosak, ennek az oka, hogy a kernel esetén az idő mérésének felbontása nem elégséges ahhoz, hogy teljes pontossággal meghatározzuk ezeket az értékeket, ugyanis a rendezési idő esetén nagyon kis időintervallumok összegzésére lenne szükség. A tesztelés során kapott pontos sebesség, és idő értékeket a függelékben találhatjuk meg.

4.3.2. Dumbbell topológia

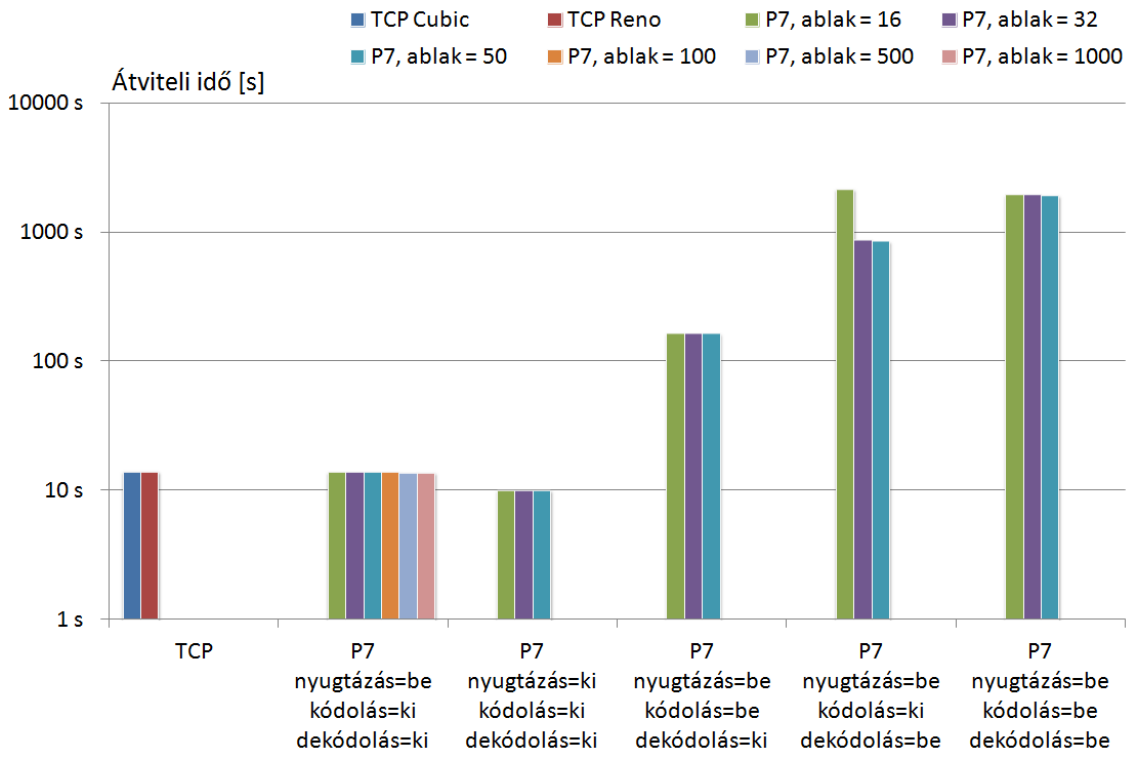
A Dumbbell topológián különböző késleltetés, és csomagvesztés értékek mellett vizsgáltam meg a TCP Cubic, a TCP Reno, és a P7 protokoll viselkedését. Először azt az esetet mutatom be, amikor nem volt emulált késleltetés, és csomagvesztés. Ezek az eredmények fognak az összehasonlítás alapjául szolgálni. Az alapvető működés után láthatjuk majd a késleltetés, és a csomagvesztés hatásait is a vizsgált protokollok esetén. Fontos megjegyezni, hogy a beállított hálózati paraméterek mindig csak a klienstől szerver felé menő forgalmat érintik, így a nyugtázást sosem befolyásolják majd a beállított késleltetés, illetve csomagvesztés értékek. A továbbított adatmennyiség ebben az esetben is egységesen 700 MB volt.

Ideális esetben ezen a topológián a protollok viselkedése nagyon hasonló volt ahhoz, mint amit láttunk a két számítógépből álló hálózat esetén. A protokollok átbocsátóképességét a 4.6. ábrán láthatjuk. Offline kódolás, és kikapcsolt dekódolás esetén az ablak méretének növelése kismértékben itt is növelte a sebességet, ha nyugtázást alkalmaztunk. Nyugtázás nélküli esetben pedig láthatjuk, hogy még ennél is nagyobb értéket kaptunk. A többi eredmény is hasonló a két számítógépből álló hálózat esetén látottakhoz.



4.6. ábra. Az átbocsátóképesség ideális esetben

Az adatátvitel idejét ebben ideális esetben a 4.7. ábrán láthatjuk.



4.7. ábra. Az adatátviteli idő ideális esetben

A 4.6. táblázatban láthajtuk ezekhez a mérési eredményekhez kapcsolódóan a rendezési, és dekódolási időt.

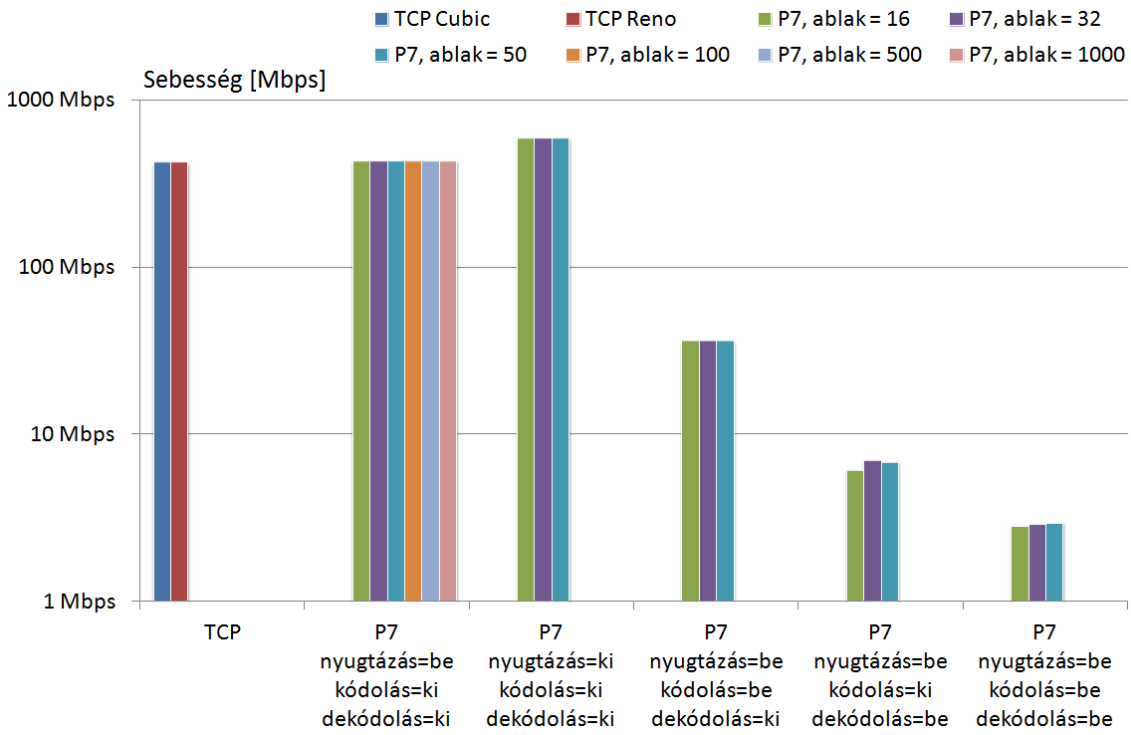
4.6. táblázat. A rendezés, és dekódolás időigénye ideális esetben

Ablakméret	Kódolás	Rendezési idő	Dekódolási idő
16	Ki	202.04 s	1946.05 s
32	Ki	149.51 s	726.24 s
50	Ki	145.08 s	701.36 s
16	Be	195.42 s	1738.9 s
32	Be	198.99 s	1736.66 s
50	Be	199.13 s	1712.56 s

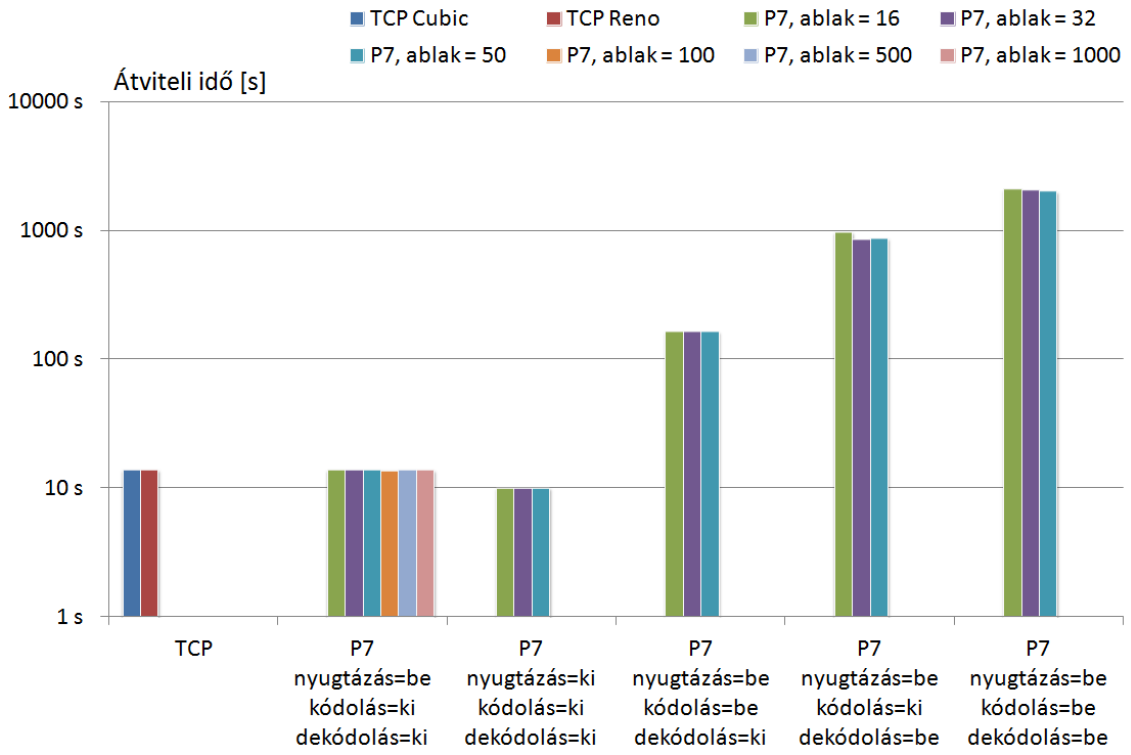
A kapott eredmények itt is hasonlóak a két számítógépből álló elrendezésnél bemutatott értékekhez.

A továbbiakban a késleltetés hatásait fogom bemutatni. Két különböző késleltetés értéket vizsgáltam meg. Az első egy kismértékű, 5 ms-os késleltetés volt, a második érték pedig egy nagyobb, 100 ms-os késleltetés volt.

Az 5 ms késleltetéshez tartozó sebességeket a 4.8. ábra, az adatátviteli időt pedig a 4.9. ábra mutatja. Láthatjuk, hogy az eredmények szinte teljesen megegyeznek azzal, mint amikor nem is alkalmaztunk késleltetést. Ennek az oka, hogy az 5 ms-os késleltetés annyira kis érték, amely szinte megegyezik azzal, amekkora késleltetés eredetileg is tapasztalható volt a kliens, és a szerver között. A tényleges késleltetés közelítőleg 1 ms volt alapesetben, tehát ennek az ötszörösét állítottuk be.



4.8. ábra. Az átbocsátóképesség 5 ms késleltetés esetén



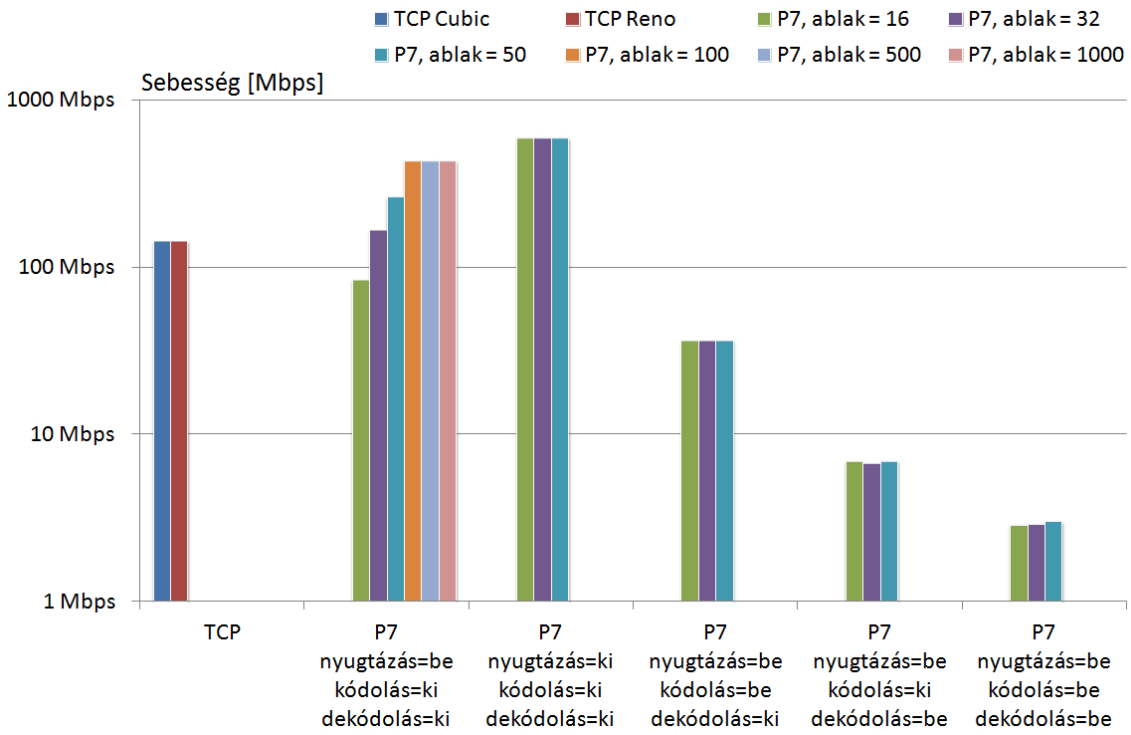
4.9. ábra. Az adatátviteli idő 5 ms késleltetés esetén

Ebben az esetben rendezési és a dekódolási időt a 4.7. táblázatban láthatjuk.

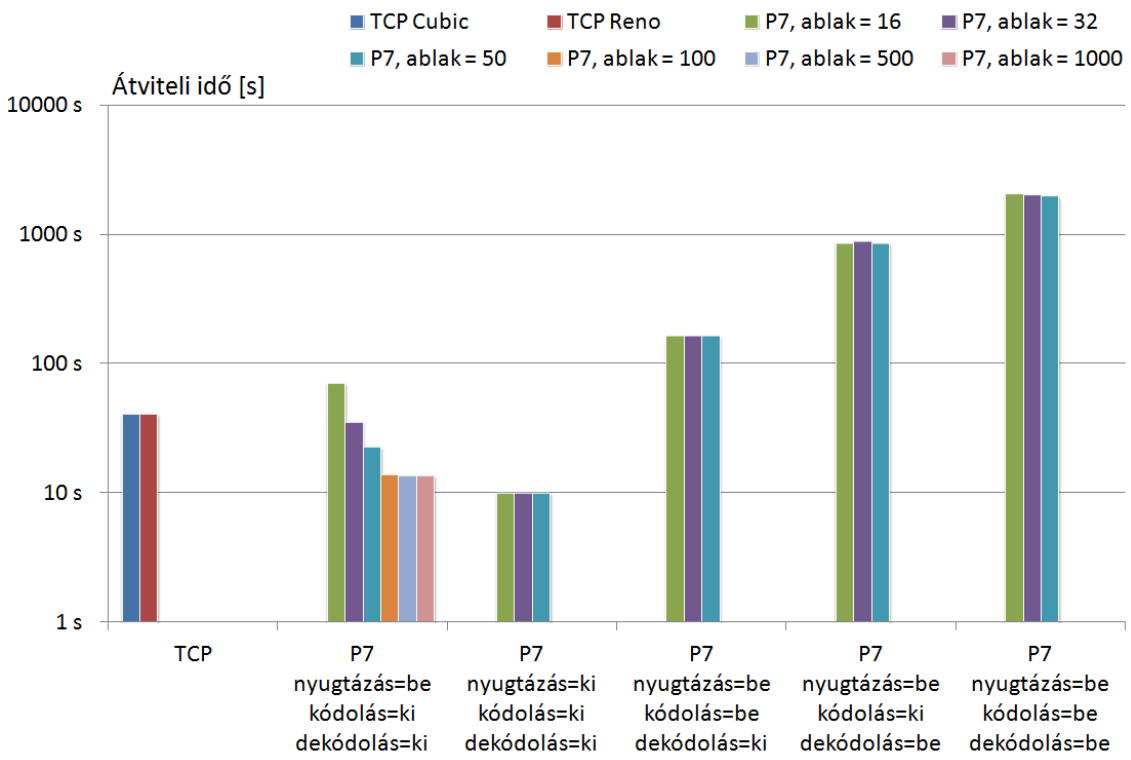
4.7. táblázat. A rendezés, és dekódolás időigénye 5 ms késleltetés esetén

Ablakméret	Kódolás	Rendezési idő	Dekódolási idő
16	Ki	146.17 s	821 s
32	Ki	148.94 s	696.62 s
50	Ki	141.52 s	724.49 s
16	Be	200.76 s	1882.84 s
32	Be	202.78 s	1841.64 s
50	Be	201.5 s	1798.86 s

A 100 ms késleltetés esetén elért átbocsátóképességet a 4.10. ábra, az átvitel idejét pedig a 4.11. ábrán láthatjuk. Ebben az esetben már látható, hogy nyugtázás, offline kódolás, és kikapcsolt dekódolás mellett az alkalmazott 100 ms késleltetés korlátozza a küldési sebességet, mivel nyugtákra kell várakoznunk minden egyes kiküldött, ablakméretnyi blokkból álló börszt után. Természetesen ha növeljük az ablakméretet, akkor egy börszt több blokkból állhat, ezért ilyenkor nő az átviteli sebesség. Ha a nyugtázást is kikapcsoljuk akkor további növekedést láthatunk, mert ilyenkor egyáltalán nem kell nyugtára várakoznunk. A többi esetben a kapott értékek hasonlóak az előző teszteknél látott eredményekhez, mert a kódolás, és a dekódolás időigénye nagyobb mértékben korlátozza a működést, mint az alkalmazott késleltetés.



4.10. ábra. Az átbocsátóképesség 100 ms késleltetés esetén



4.11. ábra. Az adatátviteli idő 100 ms késleltetés esetén

A rendezés, és dekódolás időigényét a 4.8. táblázatban tüntettem fel.

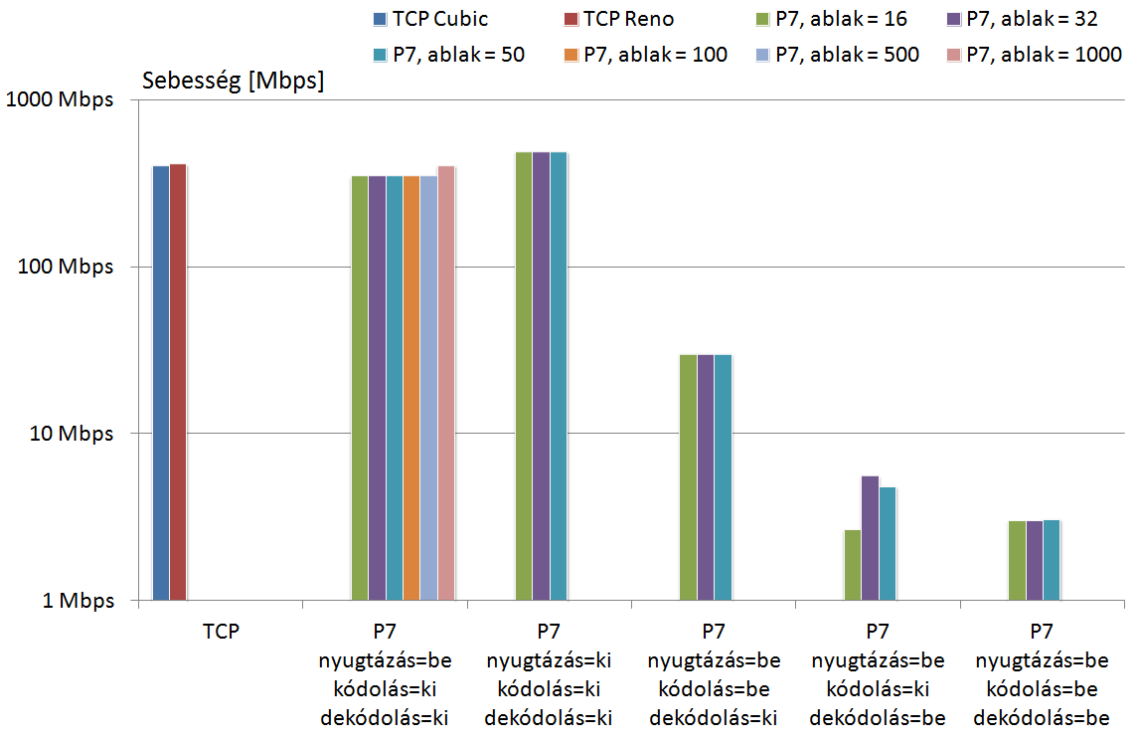
4.8. táblázat. A rendezés, és dekódolás időigénye 100 ms késleltetés esetén

Ablakméret	Kódolás	Rendezési idő	Dekódolási idő
16	Ki	149.47 s	708.74 s
32	Ki	146.66 s	740.9 s
50	Ki	147.31 s	708.16 s
16	Be	200.03 s	1866.3 s
32	Be	199.86 s	1827.29 s
50	Be	199.38 s	1765.52 s

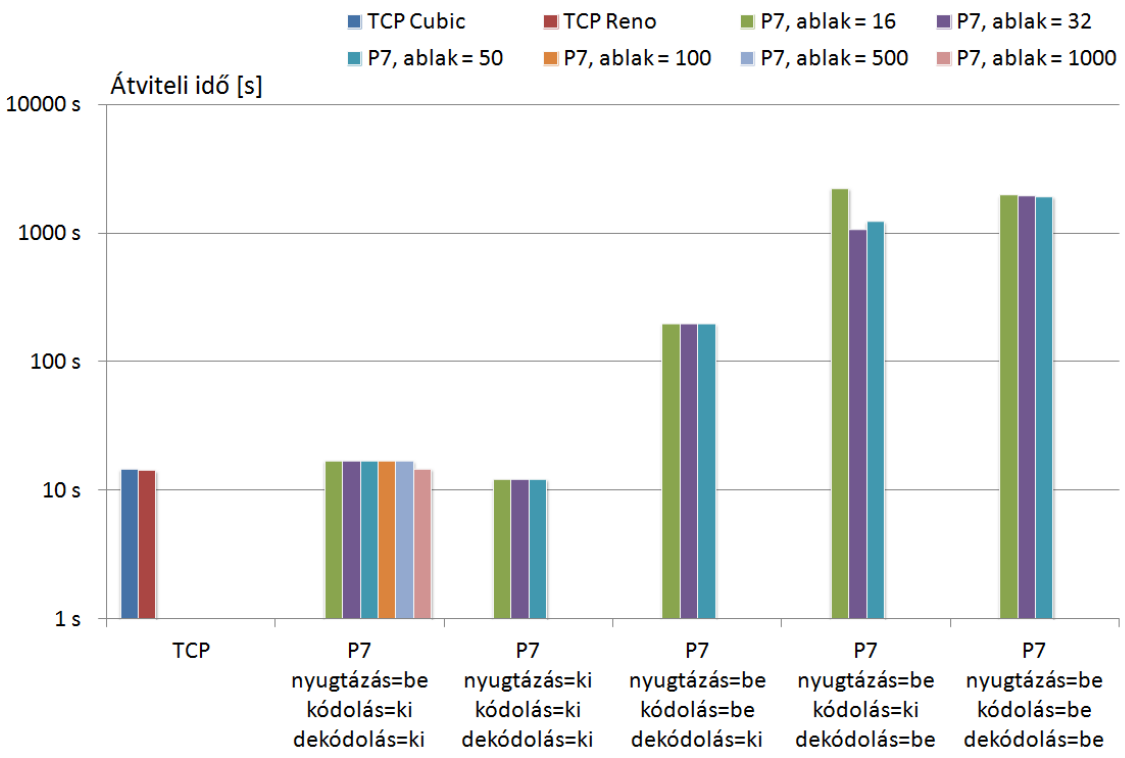
A továbbiakban ismertetem a csomagvesztést alkalmazó tesztek eredményeit. Négy különböző csomagvesztés paramétert vizsgáltam meg, a konkrét értékek 0.1%, 1%, 10%, és 30%. Minden paraméter esetén szükség volt a kliens program esetén beállítandó redundancia (r) paraméter meghatározására, a beállított értékeket a méréseknél láthatjuk. Az értékek minden esetben nagyobbak a szükséges minimális redundancia értéknél, mert a tapasztalatok azt mutatták, hogy a tc számos esetben börsztös csomagvesztést okoz, így a vevő oldal számára nem érkezik meg az érintett blokkok esetén a dekódoláshoz szükséges adatmennyiség. A protokollban jelenleg nincs beépített védelem ennek a helyzetnek a megoldására, ezért a redundacia paramétert olyan értékre kell beállítani, amely lehetővé teszi azt, hogy minden blokk esetén megérkezzen a szükséges adatmennyiség, és így végrehajtható legyen a dekódolás. Ezen kívül a tesztelés során az is megfigyelhető volt, hogy a tc jelentősebb csomagvesztést okoz, mint az elvárt, beállított érték. Ez főként a nagyobb csomagvesztés értékek esetén jelentkezett, a 30%-os csomagvesztés értéknél már közel 50%-os

volt a tényleges csomagvesztés. A tesztek során a csomagvesztés a késleltetéshez hasonlóan csak a klienstől szerver felé tartó forgalmat érintette, a visszairányú forgalmat, így a nyugtákat nem.

A legkisebb, 0.1%-os csomagvesztéshez tartozó átbocsátóképességet a 4.12. ábrán, az átviteli időt pedig a 4.13. ábrán láthatjuk. A redundancia paraméter ebben az esetben 60 csomagra volt beállítva, ez azt jelenti hogy minden blokk esetén az alapértelmezett 49 csomag helyett 60 csomagot küldtünk el. Az ábrán láhatjuk, hogy a P7 protokoll esetén az eredmények nagyon hasonlóak ahhoz, mint amit a csomagvesztés nélküli, ideális esetben láthattunk. Nagyobb különbséget azokban az esetekben láthatunk csak, amikor nem történik dekódolás, és offline kódolást alkalmazunk. Ekkor kisebb sebességet érünk el az ideális esethez képest, ezt a redundancia paraméter megnövelt értéke okozza. Ez a csökkenés a többi esetben is mindig jelentkezik, csak sokkal kisebb mértékben. Az ábrán mindig azt a sebesség, és idő értéket tüntettem fel, amelyet a szerver alkalmazás határozott meg, tehát csak a hasznos beérkező adatmennyiséggel számoltam, a redundanciával okozott növekményt, elveszett csomagokat nem számíthattam be. Azt is megfigyelhetjük, hogy a TCP protokoll is valamivel kisebb sebességet ért el az ideális esethez képest.



4.12. ábra. Az átbocsátóképesség 0.1% csomagvesztés esetén



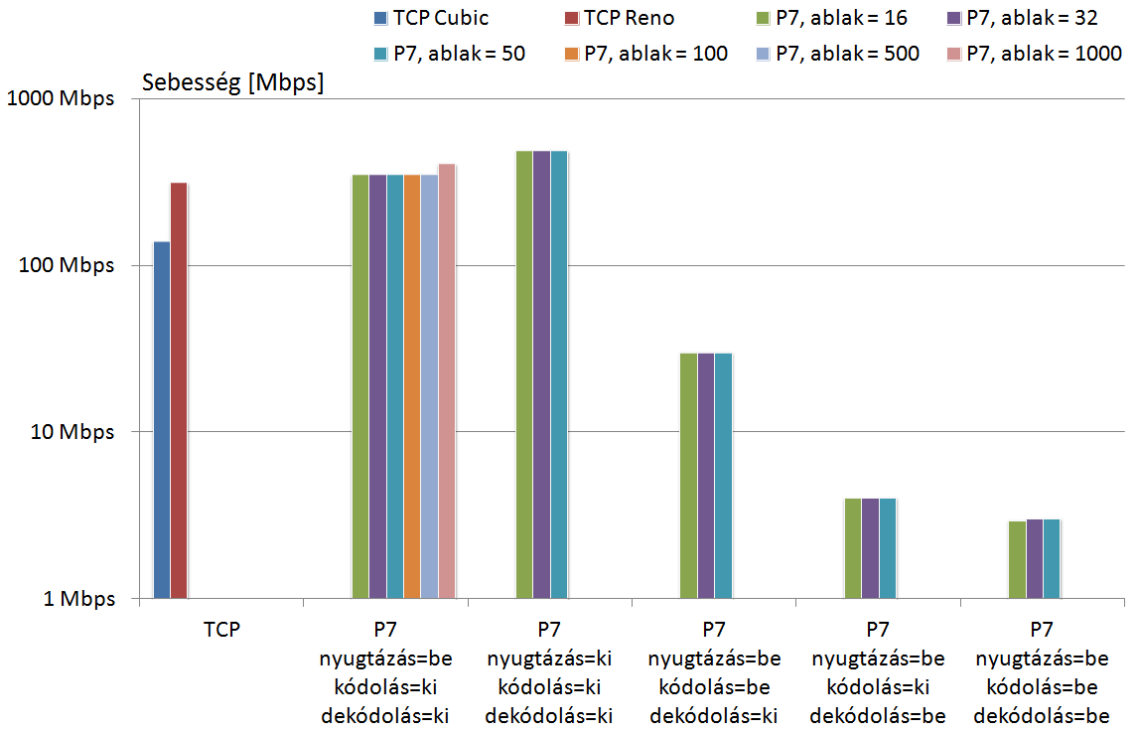
4.13. ábra. Az adatátviteli idő 0.1% csomagvesztés esetén

A méréshez kapcsolódó rendezési, és dekódolási időt a 4.9. táblázatban tüntettem fel.

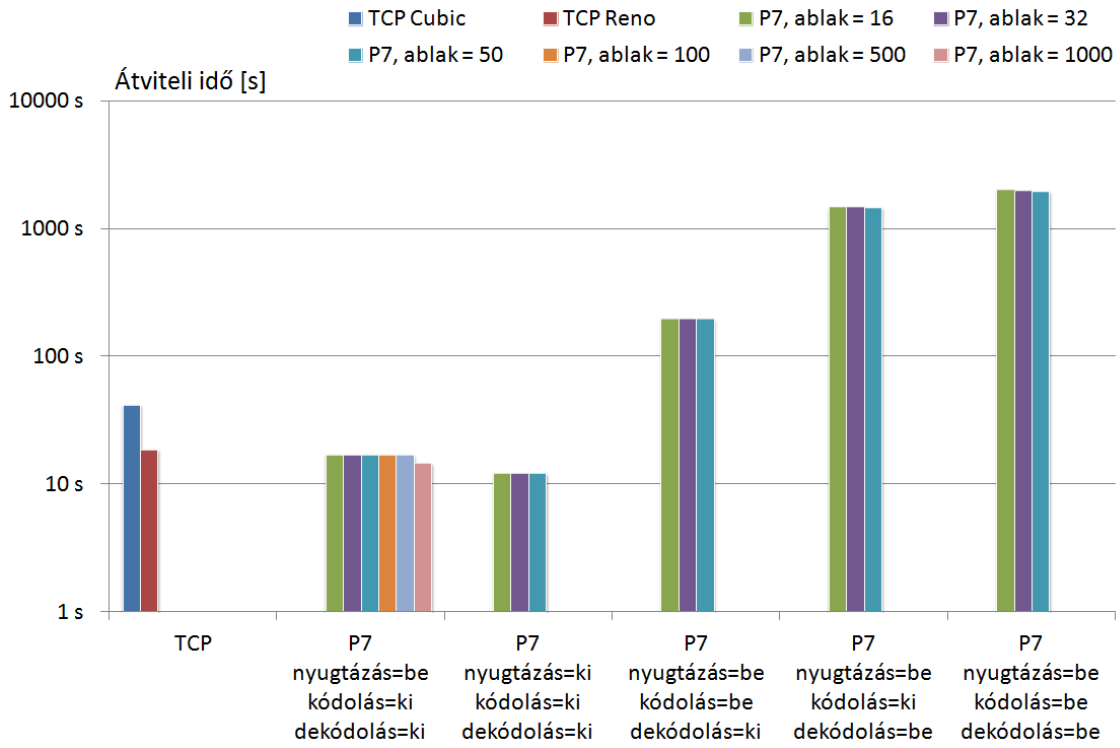
4.9. táblázat. A rendezés, és dekódolás időigénye 0.1% csomagvesztés esetén

Ablakméret	Kódolás	Rendezési idő	Dekódolási idő
16	Ki	202.59 s	2022.94 s
32	Ki	168.38 s	893.88 s
50	Ki	173.06 s	1054.42 s
16	Be	197.34 s	1765.06 s
32	Be	200.164 s	1758.05 s
50	Be	200.24 s	1722.63 s

A következő mérés esetén 1% csomagvesztést állítottam be. Ebben az esetben szintén 60 csomagra állítottam be a redundancia paraméter értékét. Az eredményeket a 4.14. ábra, és a 4.15. ábra szemlélteti. A kapott eredmények szinte teljesen megegyeznek a 0.1% esetén látottakkal, mivel a redundancia paraméter értéke nem változott. Másként viselkedett viszont a TCP protokoll, amelynél tovább csökkent az átviteli sebesség. A TCP Cubic esetén ez a csökkenés jelentősebb volt, mint a Reno esetén.



4.14. ábra. Az átbocsátóképesség 1% csomagvesztés esetén



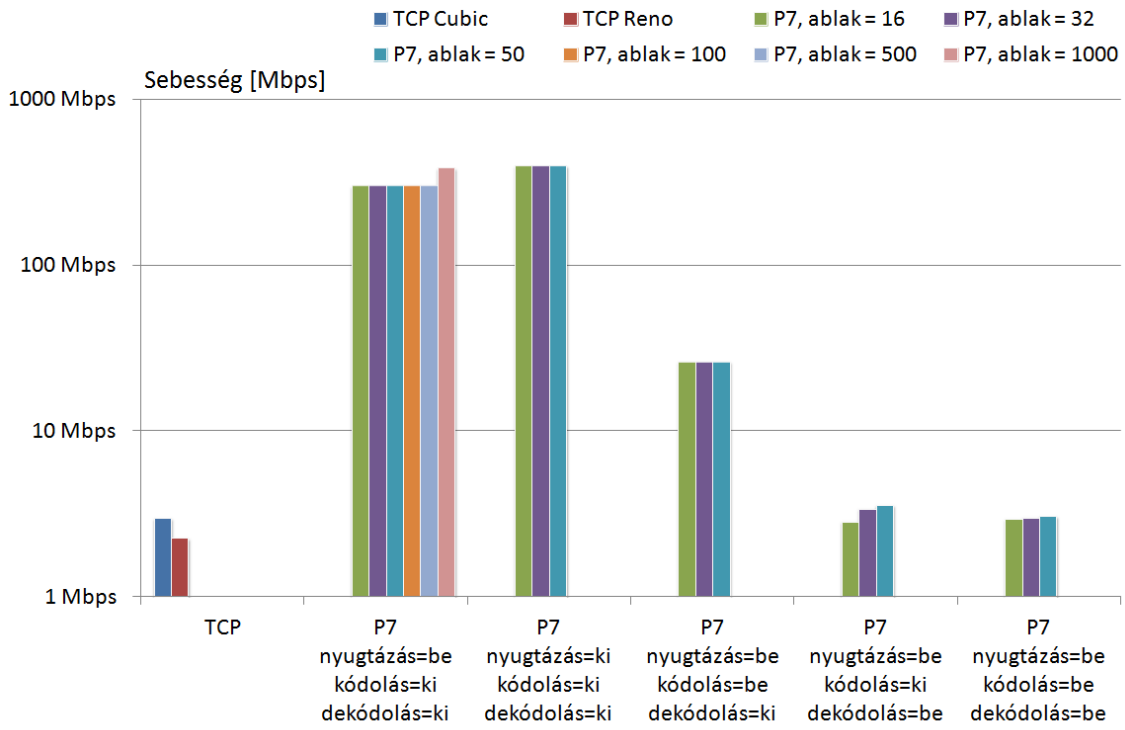
4.15. ábra. Az adatátviteli idő 1% csomagvesztés esetén

Ebben az esetben a rendezési, és a dekódolási időt a 4.10. táblázatban olvashatjuk.

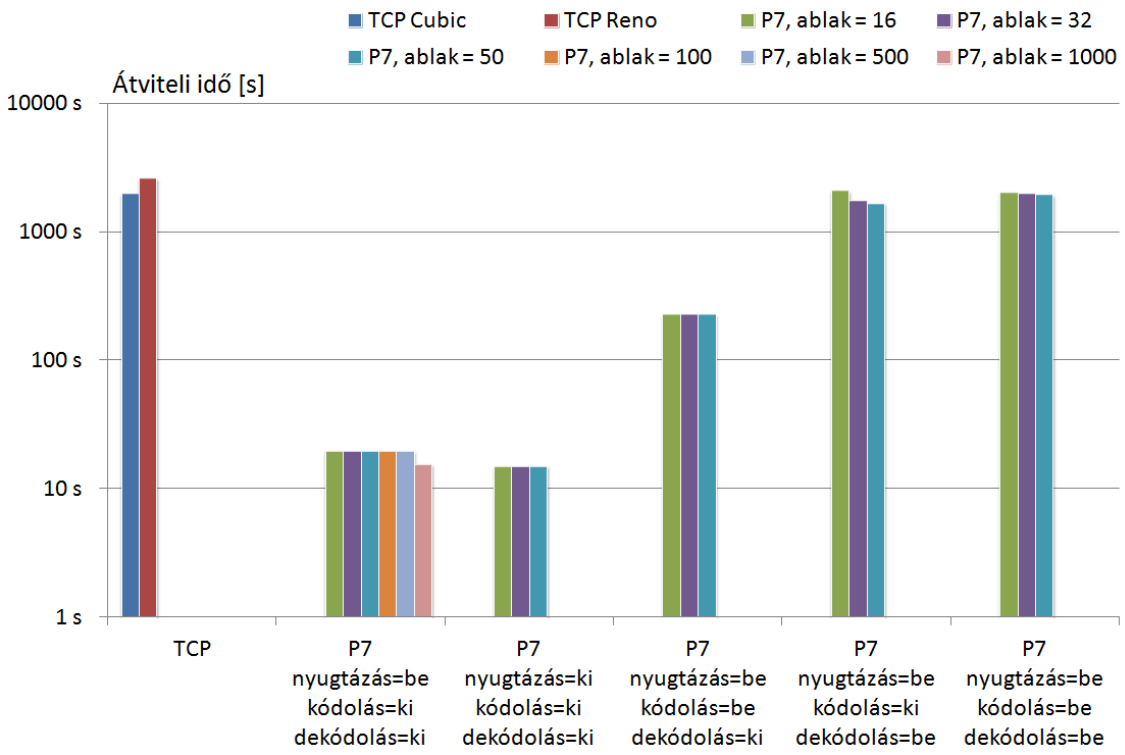
4.10. táblázat. A rendezés, és dekódolás időigénye 1% csomagvesztés esetén

Ablakméret	Kódolás	Rendezési idő	Dekódolási idő
16	Ki	185.68 s	1282.43 s
32	Ki	186.3 s	1282.89 s
50	Ki	185.4 s	1276.26 s
16	Be	199.18 s	1800.8 s
32	Be	200.24 s	1769.57 s
50	Be	200.56 s	1746.86 s

Ez után 10%-os csomagvesztést állítottam be, amelyhez már a redundancia paramétert is nagyobb értékre kellett beállítani, ebben az esetben 70 csomagra. A mérési eredményeket a 4.16. ábrán, és 4.17. ábrán tekinthetjük meg. A P7 protokoll esetén minden esetben egy kis mértékű csökkenés tapasztalható az elért sebességben, mert a redundancia paramétert nagyobb értékre állítottuk be. A TCP protokoll esetén pedig ennél sokkal jelentősebb csökkenés következett be, az adott körülmények mellett a P7 protokoll jobban teljesített.



4.16. ábra. Az átbocsátóképesség 10% csomagvesztés esetén



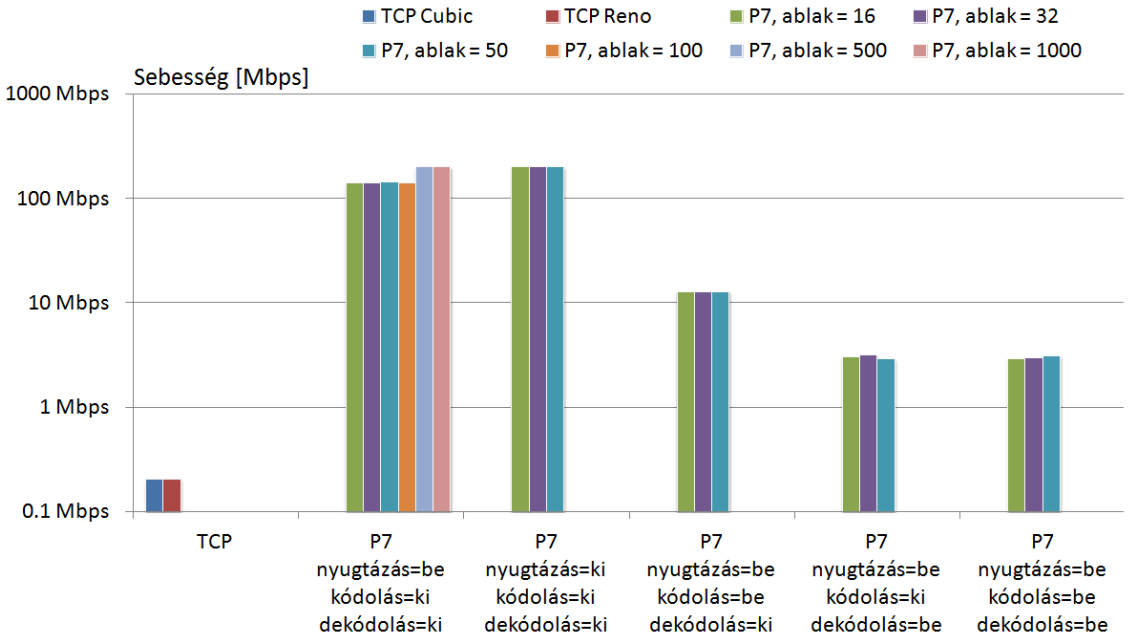
4.17. ábra. Az adatátviteli idő 10% csomagvesztés esetén

Ehhez a méréshez kapcsolódóan a 4.11. táblázat tartalmazza a rendezési, és dekódolási időt.

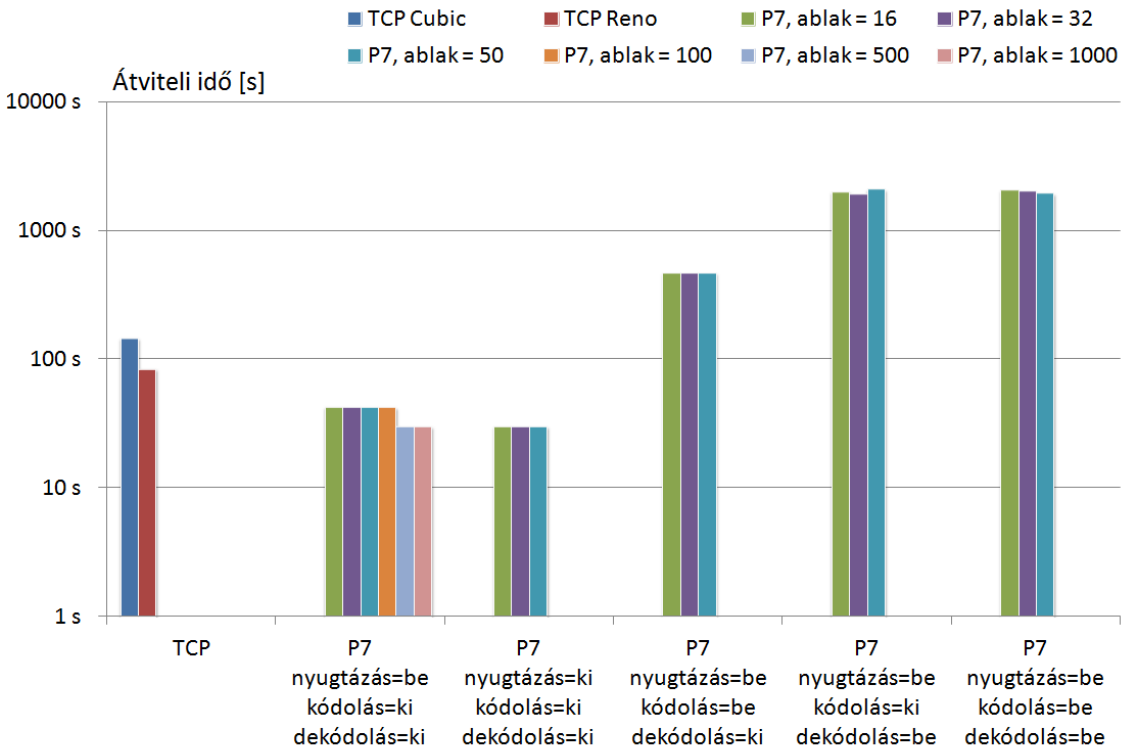
4.11. táblázat. A rendezés, és dekódolás időigénye 10% csomagvesztés esetén

Ablakméret	Kódolás	Rendezési idő	Dekódolási idő
16	Ki	201.46 s	1886.55 s
32	Ki	196.56 s	1550.22 s
50	Ki	193.65 s	1464.58 s
16	Be	199.69 s	1816.77 s
32	Be	200.88 s	1775.34 s
50	Be	199.16 s	1736.95 s

Az utolsó méréshez tartozó csomagvesztés érték 30% volt, amelynél a redundancia paramétert 150 csomagra kellett beállítani. Az elért átbocsátóképességet, és az adatátvitel időt a 4.18. ábra, és a 4.19. ábra szemlélteti. Láthatjuk, hogy a P7 protokoll esetén ebben az esetben is a beállított redundancia paraméterrel arányosan csökkent a sebesség. A TCP protokoll ebben az esetben már annyira kis sebességet ért el, hogy nem futtattam végig a tesztet, így az ábrán a megvizsgált néhány perces működés eredményét láthatjuk, amely alatt a TCP Cubic 3.58 MB, a TCP Reno pedig 2.09 MB adatot volt képes továbbítani. A pontos értékeket minden esetben a függelékben találhatjuk meg.



4.18. ábra. Az átbocsátóképesség 30% csomagvesztés esetén



4.19. ábra. Az adatátviteli idő 30% csomagvesztés esetén

A rendezési, és a dekódolás idő ebben az esetben a 4.12. táblázatban található meg.

4.12. táblázat. A rendezés, és dekódolás időigénye 30% csomagvesztés esetén

Ablakméret	Kódolás	Rendezési idő	Dekódolási idő
16	Ki	200.99 s	1793.22 s
32	Ki	200.76 s	1705.4 s
50	Ki	206.48 s	1872.82 s
16	Be	199.82 s	1853.01 s
32	Be	200.24 s	1806.34 s
50	Be	199 s	1727 s

A bemutatott mérési eredményekből látható, hogy ideális esetben a P7 protokoll képes volt megközelíteni a TCP protokoll által elért teljesítményt, azonban a dekódolás bekapcsolásával, annak időigényessége miatt az új protokoll rosszabbul teljesített. A kódolás komplexitása igaz kisebb mértékben, de hasonlóan rontja a protokoll teljesítményét. A késleltetés, illetve a csomagvesztés növekedésével azonban a TCP protokoll egyre kevésbé ért el jó eredményeket. Láthattuk, hogy ezzel szemben a P7 protokoll nagyobb csomagvesztés értékek esetén már jobb teljesítményt is képes volt elérni, mint a TCP protokoll, és ekkor egyedül a beállított redundancia paraméterből adódott az átviteli sebesség csökkenése. Ez azt mutatja, hogy a TCP protokoll működését sokkal nagyobb mértékben befolyásolják az elromló hálózati körülmények, mint a P7 protokoll esetén, ahol szinte változatlan működés történt, és így képes volt hasonló teljesítményt nyújtani, mint amit ideális esetben elért.

4.4. Jövőbeli tervek

A P7 protokoll egy folyamatban lévő kutatás és fejlesztés. A mérési eredményekből is látható, hogy számos javítási lehetőség van.

Jelenleg az egyik legfontosabb feladat az, hogy a P7 protokoll nagysebességű küldését tetszőlegesen befolyásolni tudjuk. Úgy tűnik, hogy számos esetben kedvezőtlen az, hogy maximális sebeséggel történik az adatok küldése, például nem feltétlen jó ez a működés akkor, ha egy közbeeső, kisebb sebességű link található a hálózatban. Ebben az esetben teljesen elárasztjuk a hálózat érintett részét, így természetesen maximálisan kihasználjuk a hálózat erőforrásait, de ez szükségtelen, mivel egy kisebb sebességű link esetén már a maximálisnál kisebb sebesség mellett is teljesen ki tudnánk használni az erőforrásokat, mert fizikai okokból nem lehetséges az adott, kis sebességű a szakaszon nagyobb sebességet elérni. Így nem feltétlenül indokolt a maximális sebességű adatküldés. A küldési sebesség esetén például Token Bucket mechanizmus segítségével történhetne a sebesség korlátozása, és egy olyan küldési sebesség meghatározása, amely képes a legjobb teljesítményt nyújtani az adott környezetben.

A következő fejlesztési lehetőség a kódolás, és a dekódolás optimalizálása úgy, hogy nagyobb sebességet érjünk el, de a jó tulajdonságokat megtartsuk. Ehhez a kódolási, és a dekódolási folyamatot teljesen újra kell tervezni, és átgondolni az alkalmazott algoritmust, megkeresni az egyszerűsítési lehetőségeket, és ilyen módon csökkenteni a folyamat komplexitását.

Láthattuk azt is, hogy a protokoll egy olyan feltételezéssel él, amely szerint a nyugták nem veszhetnek el. Ez a valóságban természetesen nem így működik, ezért a nyugtázási mechanizmus kiváltása is szükséges egy olyan módszerrel, amely nem, vagy csak kis mértékben korlátozza az elért átbocsátóképességet, de mégis valamilyen módon visszajelzést nyújt a küldő felé az átvitel, és a dekódolás sikerességéről.

Mindezen problémák mellett még szükséges a protokoll memória igényének csökkentése is. A tesztek során is megfigyelhető volt, hogy a kernelmemória nagysága korlátozta a beállítható ablakméretet. Természetesen minden változtatást csak úgy lehet végrehajtani, hogy a protokoll kedvező tulajdonságait megtartsuk.

5. fejezet

Összefoglalás

A jelenleg széleskörűen alkalmazott szállítási rétegbeli protokoll, a TCP nem képes teljes mértékben kihasználni a hálózati erőforrásokat, és nem képes univerzális megoldást nyújtani a torlódásszabályozásra. Annak érdekében, hogy a TCP minél inkább képes legyen kihasználni a hálózat kapacitásait, számos új verziója jelent meg, amely verziók a torlódásszabályozó algoritmus megváltoztatásával próbáltak jobb teljesítményt nyújtani. A TCP verziók esetén jelenlévő számos probléma miatt kétséges, hogy univerzális megoldást lehet találni, így szükség van olyan új protokollok kutatására, fejlesztésére, amelyek képesek a bemutatott problémákra megoldást nyújtani.

Dolgozatomban egy olyan új elv szerint működő szállítási rétegbeli protokollt mutattam be, amely a torlódásszabályozás nélküli koncepciót alkalmazza a működése során, és ekkor a fellépő csomagvesztéseket hatékony hibajavító kódolás segítségével állítja helyre. A protokoll Linux kernelben történő megvalósítása mellett bemutattam a hozzá kapcsolódóan elvégzett mérések eredményét is.

A munkám elején bemutattam a TCP esetén fennálló problémákat, és néhány jelentősebb TCP verzió esetén tárgyaltam az alkalmazott torlódásszabályozó algoritmust. Ez után rámutattam arra, hogy miért szükséges új ötletek kipróbálása a gyakorlatban, és mik a jelenlegi problémák a TCP esetén. Majd ismertettem az új protokoll koncepcióját, és a Linux kernel hálózati alrendszerének működési alapjait, az új socketeket. Bemutattam a protokoll működését. Itt részletesen tárgyaltam az egyes fázisok megvalósítását, és láthattuk, hogy a protokoll milyen módon képes garantálni a megbízható működést. Ez után a protokollal elvégzett mérések során felhasznált programokat, és topológiákat, majd a mérések eredményeit, és az azokból levonható következtetéseket írtam le. Végül kitértem a protokollal kapcsolatos jövőbeli fejlesztési lehetőségekre.

A P7 protokoll egy folyamatban lévő kutatás, ezért még számos olyan optimalizáció lehetséges, amelyek megvalósítása után elvárható a gyorsabb, és hatékonyabb működés. Az eddigi mérésekből azonban levonható volt az a következtetés, hogy kedvezőtlen hálózati paraméterek esetén a P7 protokoll egyre inkább megközelíti a TCP átbocsátóképességét, és jelentősebb csomagvesztések esetén jobban is teljesített annál. Ebből az látszik, hogy a kutatás bíztató, ugyanis a protokoll képes lehet arra, hogy teljesítse a jövőben azon elvárásokat, amelyek esetén a TCP protokoll nem tud megfelelően helytállni, és biztosítsa a

hálózati erőforrások jobb kihasználtságát. Ezáltal képesek vagyunk a torlódásszabályozásból eredő problémákra megoldást nyújtani úgy, hogy ezen új koncepció alapján egyáltalán nem alkalmazunk torlódásszabályozást.

Köszönetnyilvánítás

Szeretnék köszönetet mondani konzulensemnek, Dr. Molnár Sándornak a folyamatos konzultációkért, és azért, hogy bármikor a rendelkezésemre állt. Köszönöm Dr. Sonkoly Balázsnak a laboratóriumi mérések során nyújtott folyamatos támogatását, és a topológiák laboratóriumi megvalósításában nyújtott segítségét. Továbbá, köszönöm Temesváry Andrásnak a munkám kezdetén az algoritmusok megértésében nyújtott segítségét.

Külön köszönöm mindhármójuknak, hogy a nyári szüneti időszak alatt is folyamatosan segítették a munkámat.

Irodalomjegyzék

[1] David Clark, Scott Shenker, and Aaron Falk. GENI Research Plan. Version 4.5, Global Environment for Network Innovations, April 23 2007. `http://groups.geni.net/geni/attachment/wiki/OldGPGDesignDocuments/GDD-06-28.pdf`; accessed November 6, 2011.

[2] Sándor Molnár, Balázs Sonkoly, and Tuan Anh Trinh. A Comprehensive TCP Fairness Analysis in High Speed Networks. *Computer Communications*, 32:1460–1484, August 2009.

[3] Van Jacobson. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM 1988*, pages 314–329, Stanford, CA, USA, August 16–18 1988.

[4] Sally Floyd. Highspeed TCP for Large Congestion Windows. RFC 3649, Internet Engineering Task Force, December 2003. `http://www.ietf.org/rfc/rfc3649.txt`; accessed November 1, 2011.

[5] Tom Kelly. Scalable TCP: Improving performance in highspeed wide area networks. *ACM SIGCOMM Computer Communication Review*, 33(2):83–91, April 2003.

[6] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *Proceedings of IEEE Infocom 2004*, volume 4, pages 2514–2524, Hong Kong, China, March 7–11 2004.

[7] Injong Rhee and Lisong Xu. CUBIC: a new TCP-friendly high-speed TCP variant. In *Proceedings of Third International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet 2005)*, Lyon, France, February 3–4 2005.

[8] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking (ToN)*, 14(6):1246–1259, 2006.

[9] Kun Tan, Jingmin Song, Qian Zhang, and Murari Sridharan. A compound TCP approach for highspeed and long distance networks. In *Proceedings of IEEE Infocom 2006*, Barcelona, Spain, April 23–29 2006.

[10] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of ACM SIGCOMM 2002*, Pittsburgh, PA, USA, August 19–23 2002.

[11] Andrew S. Tanenbaum. *Számítógép hálózatok, Második, bővített, átdolgozott kiadás.* Panem, Budapest, 2004. ISBN: 963–545–384–1.

[12] Kevin Fall and Sally Floyd. Simulation-based Comparisons of Tahoe, Reno and SACK TCP. *ACM SIGCOMM Computer Communication Review*, 26(3), July 1996.

[13] Sally Floyd, Sylvia Ratnasamy, and Scott Shenker. Modifying TCP's Congestion Control for High Speeds, May 5 2002. `http://www.icir.org/floyd/notes.html`; accessed November 5, 2011.

[14] Lawrence S. Brakmo and Larry L. Peterson. TCP Vegas: End to End Congestion Avoidance on a Global Internet. *IEEE Journal on selected Areas in communications*, 13:1465–1480, 1995.

[15] Carlo Caini and Rosario Firrincieli. TCP Hybla: a TCP enhancement for heterogeneous networks. *International Journal of Satellite Communications and Networking*, 22, 2004.

[16] Shao Liu, Tamer Başar, and Rayadurgam Srikant. TCP-Illinois: a loss and delay-based congestion control algorithm for high-speed networks. In *Proceedings of the 1st international conference on Performance evaluation methodologies and tools*, valuetools, 2006.

[17] Andrea Baiocchi, Angelo P. Castellani, and Francesco Vacirca. YeAH-TCP: Yet Another Highspeed TCP.

[18] Kun Tan and Jingmin Song. A compound TCP approach for high-speed and long distance networks. In *Proceedings of IEEE INFOCOM 2006*, Barcelona, Spain, April 23–29 2006.

[19] Barath Raghavan and Alex C. Snoeren. Decongestion Control. In *Proceedings of the 5th ACM Workshop on Hot Topics in Networks (HotNets-V)*, Irvine, CA, USA, November 2006.

[20] Thomas Bonald, Mathieu Feuillet, and Alexandre Proutiére. Is the "Law of the Jungle" Sustainable for the Internet? In *Proceedings of INFOCOM 2009*, Rio de Janeiro, Brazil, 2009.

[21] Luis López, Antonio Fernández, and Vicent Cholvi. A game theoretic comparison of TCP and digital fountain based protocols. *Computer Networks*, 51(12):3413–3426, 2007.

[22] Shakeel Ahmad, Raouf Hamzaoui, and Marwan Al-akaidi. Robust Live Unicast Video Streaming with Rateless Codes. In *Proceedings of 16th International Workshop on Packet Video, PV 2007*, pages 78–84, Lausanne, Switzerland, November 2007.

[23] Michael Luby. LT Codes. In *Proceedings of 43rd Symposium on Foundations of Computer Science (FOCS 2002)*, pages 271–280, 2002.

[24] Amin Shokrollahi. Raptor codes. *IEEE Transactions on Information Theory*, 52(6), June 2006.

[25] Sameer Seth and M. Ajaykumar Venkatesulu. *TCP/IP Architecture, Design and Implementation in Linux*. Wiley-IEEE Computer Society Pr, 2008. ISBN: 9780470147733.

[26] Thomas Herbert. *The Linux TCP/IP Stack: Networking for Embedded Systems*. Charles River Media, Inc., Rockland, MA, USA, 2004. ISBN: 1584502843.

[27] Amin Shokrollahi. LDPC codes: An Introduction. *Digital Fountain, Inc., Tech. Rep*, April 2 2003.

[28] Robert C. Tausworthe. Random Numbers Generated by Linear Recurrence Modulo Two. *Mathematics of Computation*, 19:201–209, 1965.

[29] Mark Gates and Alex Warshavsky. Iperf. `http://iperf.sourceforge.net/`; accessed November 13, 2011.

[30] Alexey N. Kuznetsov. TC (Traffic Control). `http://lartc.org/`; accessed November 14, 2011.

[31] Kathleen Nichols, Steven Blake, Fred Baker, and David Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474, Internet Engineering Task Force, December 1998. `http://www.ietf.org/rfc/rfc2474.txt`; accessed November 23, 2011.

Ábrák jegyzéke

Táblázatok jegyzéke

Rövidítések jegyzéke

AIMD	Additive Increase Multiplicative Decrease
BDP	Bandwidth-Delay Product
BIC	Binary Increase Congestion control
GENI	Global Environment for Network Innovations
HSTCP	High Speed TCP
ICMP	Internet Control Message Protocol
IGMP	Internet Group Management Protocol
IP	Internet Protocol
LDPC	Low-Density Parity-Check
MIMD	Multiplicative Increase Multiplicative Decrease
MSS	Maximum Segment Size
MTU	Maximum Transmission Unit
RIP	Routing Information Protocol
RTT	Round-Trip Time
STCP	Scalable TCP
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
XCP	eXplicit Control Protocol
YeAH TCP	Yet Another High-speed TCP

Függelék

F.1. A mérésekhez kapcsolódó táblázatok

F.1. táblázat. Két számítógépből álló topológia eredményei

Protokoll	Ablak	Nyugtázás	Kódolás	Dekódolás	Átviteli idő	Sebesség
TCP Cubic	85.3 kB	-	-	-	13.8 s	425 Mbps
TCP Reno	85.3 kB	-	-	-	13.8 s	425 Mbps
P7	16 blokk	Be	Ki	Ki	13.91 s	427.54 Mbps
P7	32 blokk	Be	Ki	Ki	13.87 s	428.59 Mbps
P7	50 blokk	Be	Ki	Ki	13.78 s	431.39 Mbps
P7	100 blokk	Be	Ki	Ki	13.67 s	434.89 Mbps
P7	500 blokk	Be	Ki	Ki	13.16 s	451.94 Mbps
P7	1000 blokk	Be	Ki	Ki	12.58 s	472.55 Mbps
P7	16 blokk	Ki	Ki	Ki	8.76 s	678.48 Mbps
P7	32 blokk	Ki	Ki	Ki	8.76 s	678.37 Mbps
P7	50 blokk	Ki	Ki	Ki	8.76 s	678.36 Mbps
P7	16 blokk	Be	Be	Ki	163.99 s	36.255 Mbps
P7	32 blokk	Be	Be	Ki	163.99 s	36.255 Mbps
P7	50 blokk	Be	Be	Ki	164.103 s	36.23 Mbps
P7	16 blokk	Be	Ki	Be	2094.399 s	2.84 Mbps
P7	32 blokk	Be	Ki	Be	896.234 s	6.63 Mbps
P7	50 blokk	Be	Ki	Be	806.63 s	7.37 Mbps
P7	16 blokk	Be	Be	Be	1949.815 s	3.05 Mbps
P7	32 blokk	Be	Be	Be	1939.71 s	3.065 Mbps
P7	50 blokk	Be	Be	Be	1907.23 s	3.117 Mbps

F.2. táblázat. A Dumbbell topológián elért eredmények ideális esetben

Protokoll	Ablak	Nyugtázás	Kódolás	Dekódolás	Átviteli idő	Sebesség
TCP Cubic	85.3 kB	-	-	-	14 s	420 Mbps
TCP Reno	85.3 kB	-	-	-	14 s	420 Mbps
P7	16 blokk	Be	Ki	Ki	13.89 s	427.93 Mbps
P7	32 blokk	Be	Ki	Ki	13.9 s	427.85 Mbps
P7	50 blokk	Be	Ki	Ki	13.8 s	430.75 Mbps
P7	100 blokk	Be	Ki	Ki	13.8 s	430.72 Mbps
P7	500 blokk	Be	Ki	Ki	13.77 s	431.81 Mbps
P7	1000 blokk	Be	Ki	Ki	13.75 s	432.43 Mbps
P7	16 blokk	Ki	Ki	Ki	10.05 s	591.69 Mbps
P7	32 blokk	Ki	Ki	Ki	10.04 s	592.35 Mbps
P7	50 blokk	Ki	Ki	Ki	10.05 s	591.65 Mbps
P7	16 blokk	Be	Be	Ki	164.04 s	36.25 Mbps
P7	32 blokk	Be	Be	Ki	164.07 s	36.24 Mbps
P7	50 blokk	Be	Be	Ki	164.01 s	36.25 Mbps
P7	16 blokk	Be	Ki	Be	2149.02 s	2.77 Mbps
P7	32 blokk	Be	Ki	Be	876.42 s	6.78 Mbps
P7	50 blokk	Be	Ki	Be	847.16 s	7.02 Mbps
P7	16 blokk	Be	Be	Be	1935.11 s	3.07 Mbps
P7	32 blokk	Be	Be	Be	1936.41 s	3.07 Mbps
P7	50 blokk	Be	Be	Be	1912.46 s	3.11 Mbps

F.3. táblázat. A Dumbbell topológián elért eredmények 5 ms késleltetés esetén

Protokoll	Ablak	Nyugtázás	Kódolás	Dekódolás	Átviteli idő	Sebesség
TCP Cubic	85.3 kB	-	-	-	13.9 s	423 Mbps
TCP Reno	85.3 kB	-	-	-	13.9 s	422 Mbps
P7	16 blokk	Be	Ki	Ki	13.86 s	428.88 Mbps
P7	32 blokk	Be	Ki	Ki	13.86 s	428.9 Mbps
P7	50 blokk	Be	Ki	Ki	13.85 s	429.41 Mbps
P7	100 blokk	Be	Ki	Ki	13.78 s	431.33 Mbps
P7	500 blokk	Be	Ki	Ki	13.8 s	430.76 Mbps
P7	1000 blokk	Be	Ki	Ki	13.79 s	431.15 Mbps
P7	16 blokk	Ki	Ki	Ki	10.05 s	591.74 Mbps
P7	32 blokk	Ki	Ki	Ki	10.04 s	592.05 Mbps
P7	50 blokk	Ki	Ki	Ki	10.04 s	591.91 Mbps
P7	16 blokk	Be	Be	Ki	163.32 s	36.4 Mbps
P7	32 blokk	Be	Be	Ki	163.75 s	36.31 Mbps
P7	50 blokk	Be	Be	Ki	163.69 s	36.32 Mbps
P7	16 blokk	Be	Ki	Be	968.21 s	6.14 Mbps
P7	32 blokk	Be	Ki	Be	846.52 s	7.02 Mbps
P7	50 blokk	Be	Ki	Be	866.82 s	6.86 Mbps
P7	16 blokk	Be	Be	Be	2084.5 s	2.85 Mbps
P7	32 blokk	Be	Be	Be	2045.21 s	2.91 Mbps
P7	50 blokk	Be	Be	Be	2001.19 s	2.97 Mbps

F.4. táblázat. A Dumbbell topológián elért eredmények 100 ms késleltetés esetén

Protokoll	Ablak	Nyugtázás	Kódolás	Dekódolás	Átviteli idő	Sebesség
TCP Cubic	85.3 kB	-	-	-	41.2 s	143 Mbps
TCP Reno	85.3 kB	-	-	-	41.1 s	143 Mbps
P7	16 blokk	Be	Ki	Ki	71.12 s	83.6 Mbps
P7	32 blokk	Be	Ki	Ki	35.56 s	167.21 Mbps
P7	50 blokk	Be	Ki	Ki	22.76 s	261.22 Mbps
P7	100 blokk	Be	Ki	Ki	13.89 s	428.07 Mbps
P7	500 blokk	Be	Ki	Ki	13.74 s	432.67 Mbps
P7	1000 blokk	Be	Ki	Ki	13.77 s	431.81 Mbps
P7	16 blokk	Ki	Ki	Ki	10.05 s	591.58 Mbps
P7	32 blokk	Ki	Ki	Ki	10.04 s	592.31 Mbps
P7	50 blokk	Ki	Ki	Ki	10.05 s	591.89 Mbps
P7	16 blokk	Be	Be	Ki	163.68 s	36.22 Mbps
P7	32 blokk	Be	Be	Ki	163.72 s	36.22 Mbps
P7	50 blokk	Be	Be	Ki	163.69 s	36.22 Mbps
P7	16 blokk	Be	Ki	Be	859.2 s	6.92 Mbps
P7	32 blokk	Be	Ki	Be	888.59 s	6.69 Mbps
P7	50 blokk	Be	Ki	Be	856.05 s	6.95 Mbps
P7	16 blokk	Be	Be	Be	2067.15 s	2.88 Mbps
P7	32 blokk	Be	Be	Be	2027.9 s	2.93 Mbps
P7	50 blokk	Be	Be	Be	1965.62 s	3.02 Mbps

F.5. táblázat. A Dumbbell topológián elért eredmények 0.1% csomagvesztés esetén

Protokoll	Ablak	Nyugtázás	Kódolás	Dekódolás	Átviteli idő	Sebesség
TCP Cubic	85.3 kB	-	-	-	14.6 s	403 Mbps
TCP Reno	85.3 kB	-	-	-	14.3 s	410 Mbps
P7	16 blokk	Be	Ki	Ki	17.05 s	348.69 Mbps
P7	32 blokk	Be	Ki	Ki	17.02 s	349.26 Mbps
P7	50 blokk	Be	Ki	Ki	16.98 s	350.08 Mbps
P7	100 blokk	Be	Ki	Ki	16.9 s	351.74 Mbps
P7	500 blokk	Be	Ki	Ki	16.9 s	351.85 Mbps
P7	1000 blokk	Be	Ki	Ki	14.72 s	403.86 Mbps
P7	16 blokk	Ki	Ki	Ki	12.21 s	486.91 Mbps
P7	32 blokk	Ki	Ki	Ki	12.22 s	486.59 Mbps
P7	50 blokk	Ki	Ki	Ki	12.21 s	486.85 Mbps
P7	16 blokk	Be	Be	Ki	197.03 s	30.18 Mbps
P7	32 blokk	Be	Be	Ki	197.1 s	30.17 Mbps
P7	50 blokk	Be	Be	Ki	197.1 s	30.17 Mbps
P7	16 blokk	Be	Ki	Be	2226.42 s	2.67 Mbps
P7	32 blokk	Be	Ki	Be	1063.18 s	5.59 Mbps
P7	50 blokk	Be	Ki	Be	1228.32 s	4.84 Mbps
P7	16 blokk	Be	Be	Be	1963.29 s	3.03 Mbps
P7	32 blokk	Be	Be	Be	1959.01 s	3.03 Mbps
P7	50 blokk	Be	Be	Be	1923.78 s	3.09 Mbps

F.6. táblázat. A Dumbbell topológián elért eredmények 1% csomagvesztés esetén

Protokoll	Ablak	Nyugtázás	Kódolás	Dekódolás	Átviteli idő	Sebesség
TCP Cubic	85.3 kB	-	-	-	42 s	140 Mbps
TCP Reno	85.3 kB	-	-	-	18.7 s	314 Mbps
P7	16 blokk	Be	Ki	Ki	17.04 s	348.99 Mbps
P7	32 blokk	Be	Ki	Ki	17.02 s	349.35 Mbps
P7	50 blokk	Be	Ki	Ki	16.92 s	351.45 Mbps
P7	100 blokk	Be	Ki	Ki	16.88 s	352.32 Mbps
P7	500 blokk	Be	Ki	Ki	16.89 s	351.92 Mbps
P7	1000 blokk	Be	Ki	Ki	14.59 s	407.54 Mbps
P7	16 blokk	Ki	Ki	Ki	12.21 s	486.75 Mbps
P7	32 blokk	Ki	Ki	Ki	12.2 s	487.16 Mbps
P7	50 blokk	Ki	Ki	Ki	12.2 s	487.16 Mbps
P7	16 blokk	Be	Be	Ki	197.03 s	30.18 Mbps
P7	32 blokk	Be	Be	Ki	197.1 s	30.16 Mbps
P7	50 blokk	Be	Be	Ki	197.06 s	30.17 Mbps
P7	16 blokk	Be	Ki	Be	1469.1 s	4.05 Mbps
P7	32 blokk	Be	Ki	Be	1470.12 s	4.04 Mbps
P7	50 blokk	Be	Ki	Be	1462.59 s	4.07 Mbps
P7	16 blokk	Be	Be	Be	2000.9 s	2.97 Mbps
P7	32 blokk	Be	Be	Be	1970.67 s	3.02 Mbps
P7	50 blokk	Be	Be	Be	1948.39 s	3.05 Mbps

F.7. táblázat. A Dumbbell topológián elért eredmények 10% csomagvesztés esetén

Protokoll	Ablak	Nyugtázás	Kódolás	Dekódolás	Átviteli idő	Sebesség
TCP Cubic	85.3 kB	-	-	-	1970.6 s	2.98 Mbps
TCP Reno	85.3 kB	-	-	-	2594 s	2.26 Mbps
P7	16 blokk	Be	Ki	Ki	19.84 s	299.74 Mbps
P7	32 blokk	Be	Ki	Ki	19.78 s	300.66 Mbps
P7	50 blokk	Be	Ki	Ki	19.7 s	301.79 Mbps
P7	100 blokk	Be	Ki	Ki	19.65 s	302.6 Mbps
P7	500 blokk	Be	Ki	Ki	19.65 s	302.64 Mbps
P7	1000 blokk	Be	Ki	Ki	15.47 s	384.34 Mbps
P7	16 blokk	Ki	Ki	Ki	14.99 s	396.68 Mbps
P7	32 blokk	Ki	Ki	Ki	14.99 s	396.7 Mbps
P7	50 blokk	Ki	Ki	Ki	15 s	396.47 Mbps
P7	16 blokk	Be	Be	Ki	226.96 s	26.2 Mbps
P7	32 blokk	Be	Be	Ki	227.03 s	26.19 Mbps
P7	50 blokk	Be	Be	Ki	227.04 s	26.19 Mbps
P7	16 blokk	Be	Ki	Be	2088.91 s	2.85 Mbps
P7	32 blokk	Be	Ki	Be	1747.72 s	3.4 Mbps
P7	50 blokk	Be	Ki	Be	1659.15 s	3.58 Mbps
P7	16 blokk	Be	Be	Be	2017.35 s	2.95 Mbps
P7	32 blokk	Be	Be	Be	1977.03 s	3.01 Mbps
P7	50 blokk	Be	Be	Be	1937.07 s	3.07 Mbps

F.8. táblázat. A Dumbbell topológián elért eredmények 30% csomagvesztés esetén

Protokoll	**Ablak**	**Nyugtázás**	**Kódolás**	**Dekódolás**	**Átviteli idő**	**Sebesség**
TCP Cubic	85.3 kB	-	-	-	145.3 s	207 Kbps
TCP Reno	85.3 kB	-	-	-	84.1 s	208 Kbps
P7	16 blokk	Be	Ki	Ki	42.58 s	139.63 Mbps
P7	32 blokk	Be	Ki	Ki	42.27 s	140.67 Mbps
P7	50 blokk	Be	Ki	Ki	42.14 s	141.09 Mbps
P7	100 blokk	Be	Ki	Ki	42.21 s	140.86 Mbps
P7	500 blokk	Be	Ki	Ki	29.91 s	198.77 Mbps
P7	1000 blokk	Be	Ki	Ki	29.88 s	198.99 Mbps
P7	16 blokk	Ki	Ki	Ki	29.92 s	198.75 Mbps
P7	32 blokk	Ki	Ki	Ki	29.92 s	198.72 Mbps
P7	50 blokk	Ki	Ki	Ki	29.92 s	198.77 Mbps
P7	16 blokk	Be	Be	Ki	466.74 s	12.74 Mbps
P7	32 blokk	Be	Be	Ki	466.98 s	12.73 Mbps
P7	50 blokk	Be	Be	Ki	466.95 s	12.73 Mbps
P7	16 blokk	Be	Ki	Be	1995.25 s	2.98 Mbps
P7	32 blokk	Be	Ki	Be	1907.29 s	3.12 Mbps
P7	50 blokk	Be	Ki	Be	2080.28 s	2.86 Mbps
P7	16 blokk	Be	Be	Be	2053.84 s	2.89 Mbps
P7	32 blokk	Be	Be	Be	2007.56 s	2.96 Mbps
P7	50 blokk	Be	Be	Be	1927.08 s	3.09 Mbps