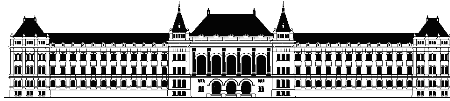


Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai Tanszék

Torlódásszabályozás nélküli transzport protokoll tervezése és fejlesztése

Dr. Molnár Sándor

Tartalomjegyzék

Kivonat **V**

1. Az új transzport protokoll **1**

1.1. Működési elv 1
1.2. A Linux kernel alrendszere 3
1.3. Az új protokoll működése 4
1.4. A protokoll fejléce 5
1.5. A kapcsolatfelépítés 6
1.6. Az adatok kódolása 10
1.6.1. A kódolási folyamat 10
1.6.2. Az LDPC kódolás 11
1.6.3. Az LT kódolás 13
1.7. Az adatokküldés 16
1.7.1. A küldési tárolók felépítése 16
1.7.2. Az adatok elküldése 17
1.7.3. A fejléc mezőinek beállítása 19
1.8. Az adatfogadás 19
1.9. Az adatok dekódolása 21
1.9.1. A vételi tárolók felépítése 21
1.9.2. Az LT dekódolás 22
1.9.3. Az LDPC dekódolás 26
1.10. Kapcsolatbontás 28
1.11. A protokoll paraméterei 32

2. Összefoglalás **36**

Irodalomjegyzék **38**

Kivonat

Az Internet folyamatos változásának következtében a torlódásszabályozási feladatokra újabbnál újabb TCP (Transmission Control Protocol) verziókat fejlesztettek ki. Ezek TCP verziók képesek egyre hatékonyabb megoldást nyújtani a forgalom átvitelére, azonban az eltérő és egyre inkább változó hálózati környezet miatt nem képesek univerzális, optimális megoldást nyújtani a folyamatosan változó, heterogén környezet okozta kihívásokra. Lehetséges alternatív megoldás a jövő Internetére nézve az, hogy egyáltalán nem alkalmazunk torlódásszabályozást. Ebben az esetben megengedünk a hálózatban maximális sebességgel történő adatküldést, ugyanakkor hatékony hibajavító kódolást használunk a fellépő csomagvesztések javítására.
A tanulmány Dr. Molnár Sándor egyetemi docens vezetésével dolgozó kutatócsoport ezirányú kutatási eredményeinek az összefogalalása.

1. fejezet

Az új transzport protokoll

1.1. Működési elv

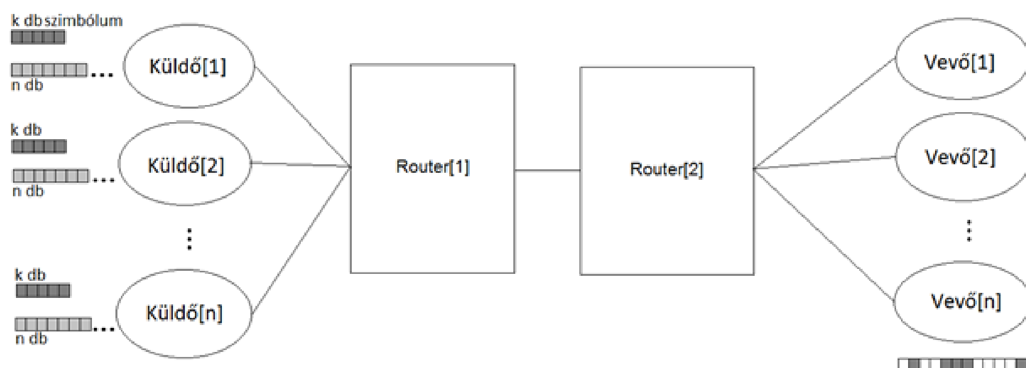
A kiindulási alapötlet a GENI [1] javaslataiban jelent meg először. Az ötlet ígéretesnek tűnik, de eddig néhány, ezzel az elvvel kapcsolatos munkán kívül semmilyen realizációt, vagy további finomítást nem publikáltak. A következőkben röviden áttekintem az ehhez kapcsolódó munkákat. Raghavan és Snoeren tanulmányozta a torlódásszabályozás nélküli hálózat nyújtotta előnyöket, és bemutatott egy torlódásmentesítőt, mint egy lehetséges megoldás alapját [2]. Bonald és mások tanulmányozták a torlódásszabályozás nélküli hálózat viselkedését [3]. 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 [4]. 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 [5] 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) [6] 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 [7]. 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 1.1. 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.



1.1. 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

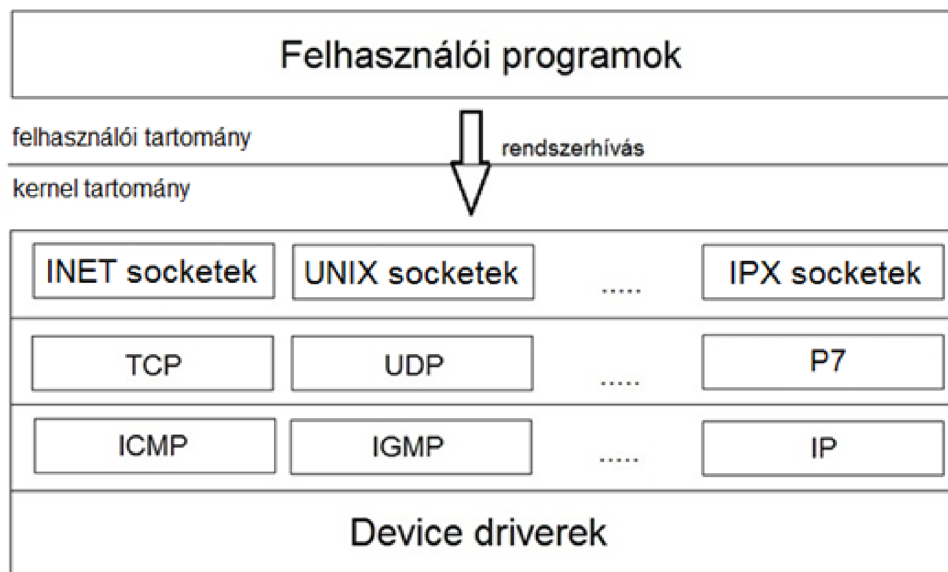
visszaállíthassuk az eredeti k szimbólumot. Ez a tulajdonság rendkívül hibatűrő adatátvitelt tesz lehetővé, még abban az esetben is, ha a csomagvesztés mértéke dinamikusan változik, és így nagymértékben csomósodik. Ha ekkor a dekódolás sikertelen lenne, akkor a vevő megpróbálhat újabb kódolt szimbólumokat 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 protokoll Linux kernelben implementálásra került [8] és a következő részben a Linux kernel hálózati alrendszerének felépítése kerül ismertetésre.

1.2. A Linux kernel 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 [9]. Az egyes rétegeket a 1.2. ábra szemlélteti.



1.2. 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 *ÚJ*. 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 új az INET család támogatja.

Az operációs rendszer által használt memóriát két különböző tartományra oszthatjuk fel, ezen tartományok a *kernel tartomány*, és a *felhasználói tartomány* [10]. 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.

1.3. Az új protokoll működése

A új 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 új 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 új 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.

1.4. A protokoll fejléce

A fejléc felépítését a 1.3. ábrán láthatjuk.

<table>
<tr><td colspan="2">Forrás port (16)</td><td colspan="2">Cél port (16)</td></tr>
<tr><td colspan="4">Blokk ID (32)</td></tr>
<tr><td colspan="4">S1 (32)</td></tr>
<tr><td>adat ofszet (4)</td><td></td><td>flagek (6)</td><td></td></tr>
<tr><td colspan="2">Ellenörzőösszeg (16)</td><td colspan="2"></td></tr>
<tr><td colspan="4">S2 (32)</td></tr>
<tr><td colspan="4">S3 (32)</td></tr>
<tr><td colspan="4">Adatok</td></tr>
</table>

1.3. ábra. A új protokoll fejléce

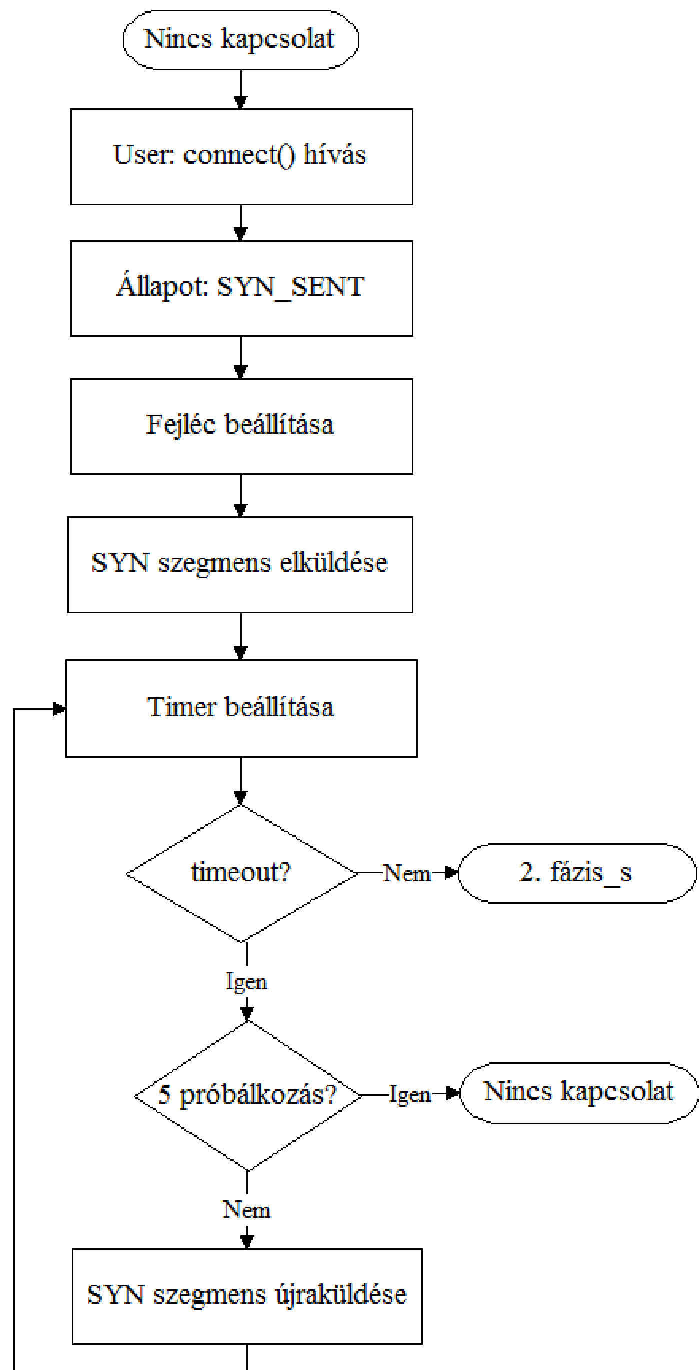
A zárójelben feltüntetett számok az adott mező méretét jelentik, bitekben számítva. Az első, és a második mező azonosítja a küldő, és a vevő alkalmazás számára kiosztott *kommunikációs portot.* A portok segítségével vagyunk képesek különbséget tenni az ugyanazon állomáson futó különböző hálózati alkalmazások 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.

1.5. A kapcsolatfelépítés

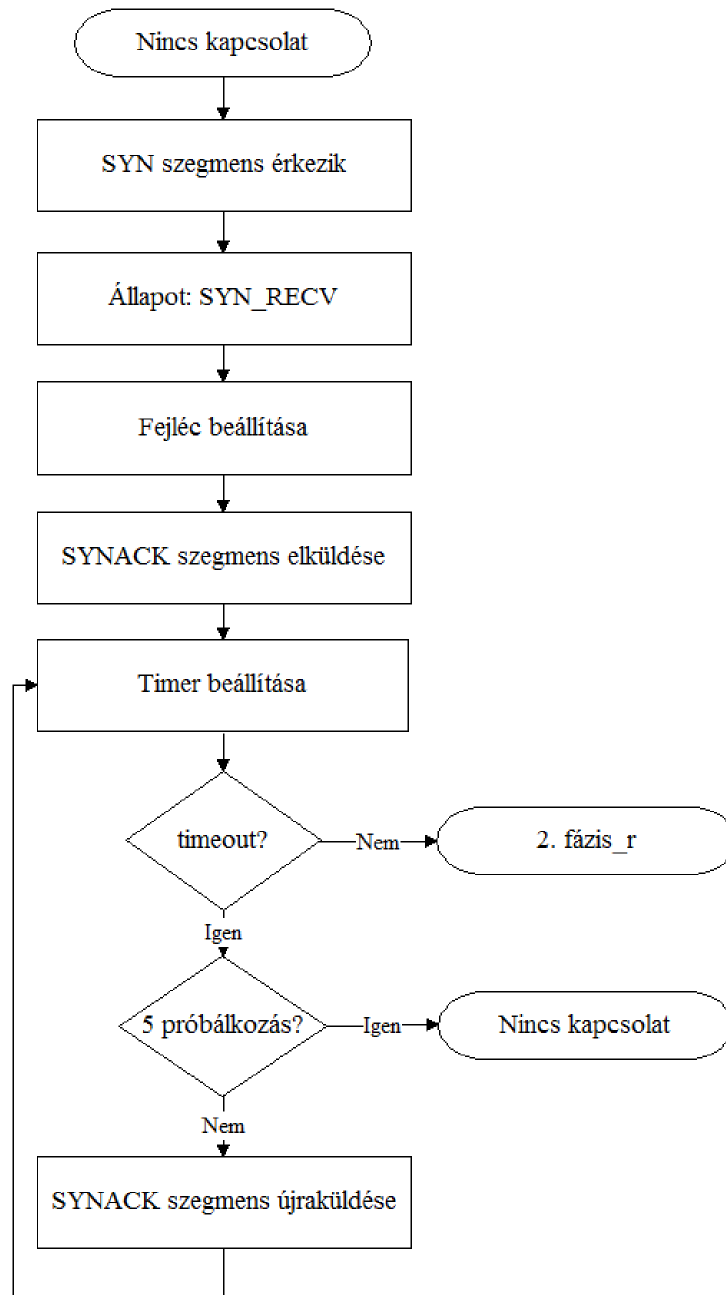
Ebben a részben a új protokoll kapcsolat felépítési folyamatát ismertetem. A folyamat a TCP protokollnál alkalmazott módszerhez hasonlít.

A kapcsolat felépítésnek három lépését különböztetem meg. Az első lépést a 1.4. 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 1.4. 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.



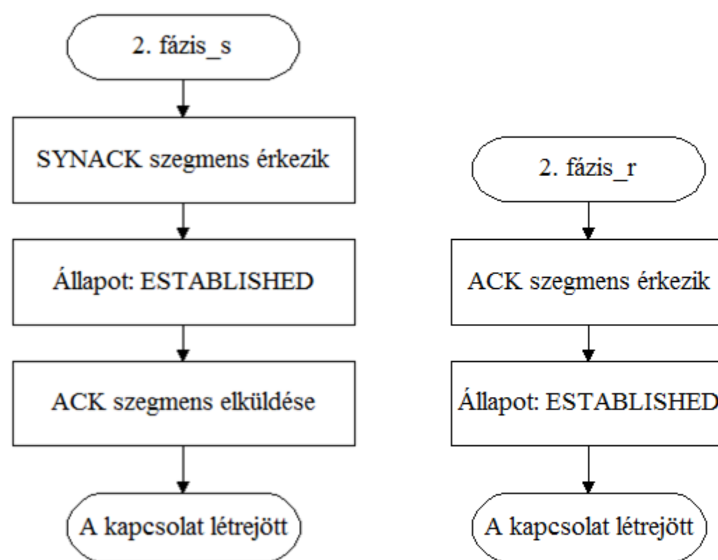
1.4. ábra. A kapcsolat felépítés 1. lépése

A 1.5. 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.



1.5. ábra. A kapcsolat felépítés 2. lépése

Az utolsó lépést a 1.6. 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.



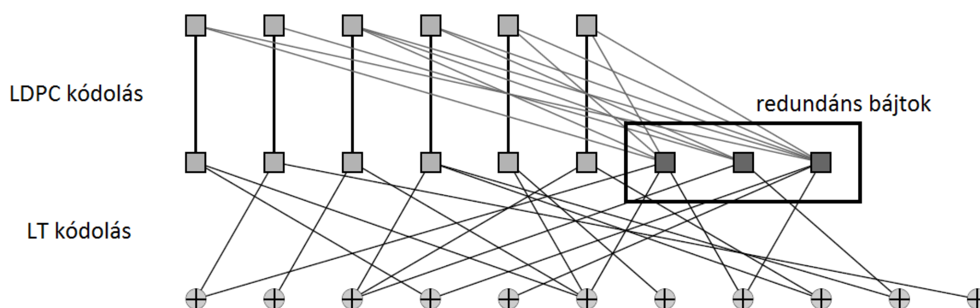
1.6. ábra. A kapcsolat felépítés 3. lépése

1.6. 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) [11] 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.

1.6.1. A kódolási folyamat

Amikor a kernel megkapja a küldendő adatokat a felhasználótól, akkor ezeket eltárolja az első olyan tárolóban, amely szabad. Az alkalmazott kódolásokat a 1.7. ábrán láthatjuk.

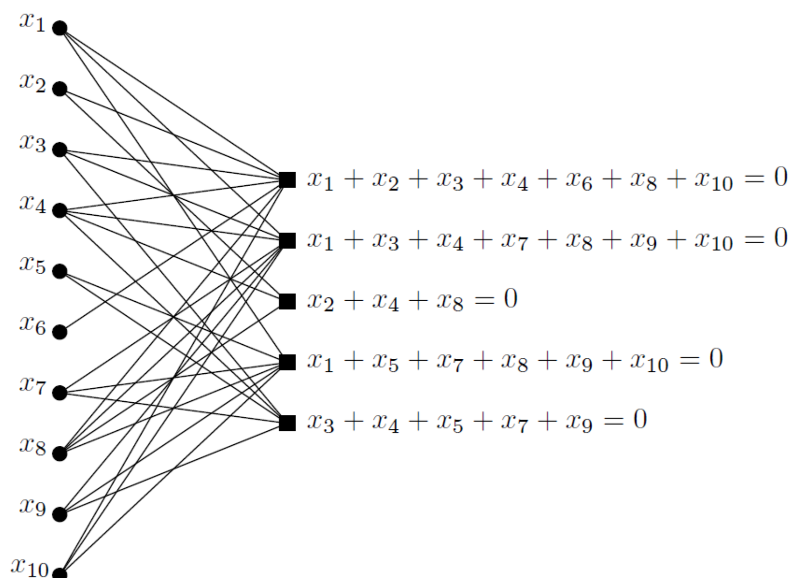


1.7. ábra. Az alkalmazott Raptor kódolás

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.

1.6.2. 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 1.8. ábra.



1.8. ábra. Egy LDPC kód

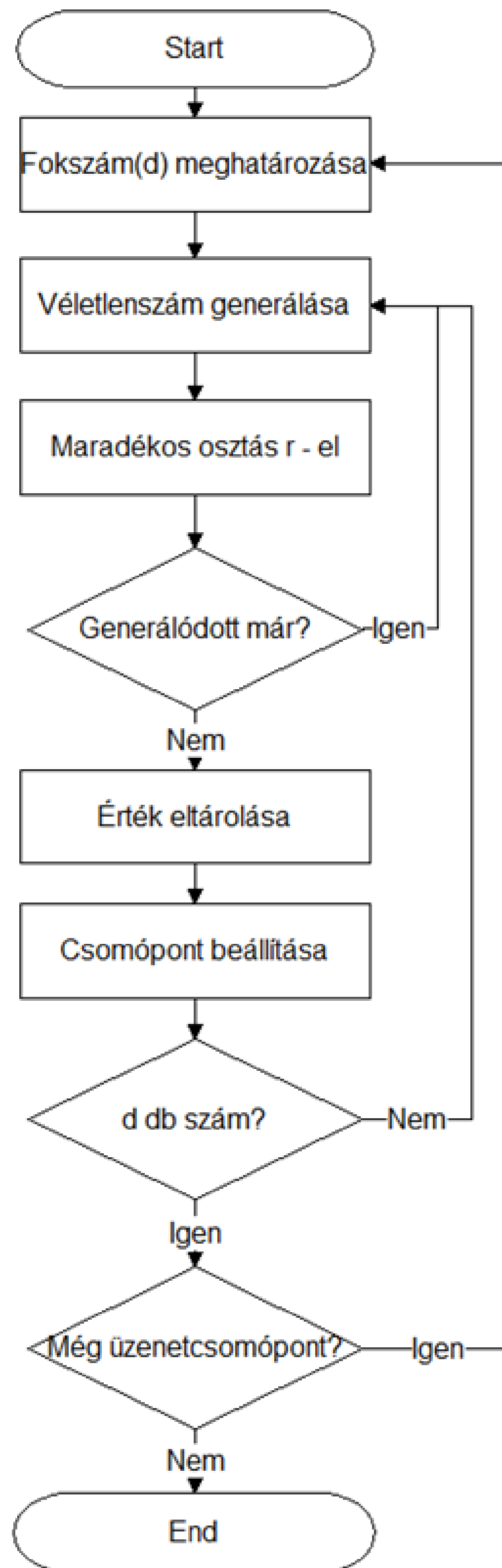
Látható, hogy minden egyes ellenőrző csomópontra teljesül az, hogy a szomszédos üzenet csomópontok összege (amelyet XOR művelettel képzünk) 0 értékű kell legyen.

A jelenlegi implementáció esetén felhasznált LDPC kódokat egy adott valószínűség eloszlás szerint generálom, az ellenőrző csomópontok kezdeti értéke pedig 0. A kódolási folyamatot a 1.9. 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 1.1. képlet szerint:

$$cknode[rand] = cknode[rand] \oplus msg[i] \quad (1.1)$$

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

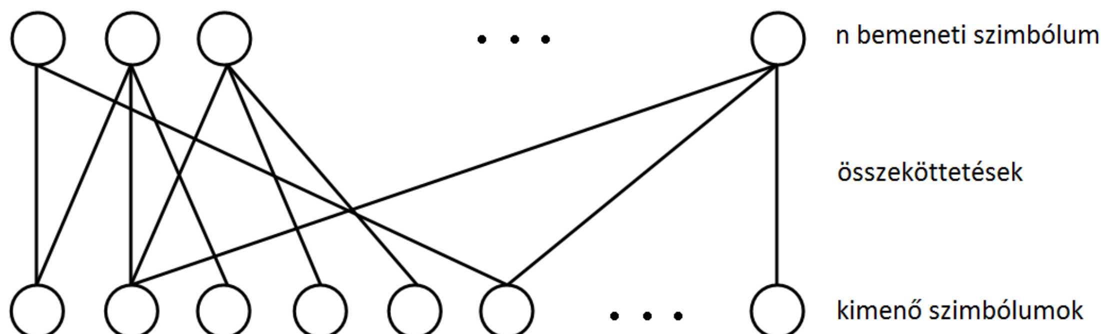
lesz. Ez a 65536 bájt képezi az LT kódolás bemenetét.



1.9. ábra. Az LDPC kódolás folyamata

1.6.3. 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 1.10. 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.



1.10. ábra. A szimbólumok közti összeköttetések

Ekkor $\Omega_i = P(D = i)$. A 1.1. 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 1.2. képlet szerint számítható ki:

$$c = \frac{2^{32} - 1}{\sum_{i=0}^{n} \Omega_i} \tag{1.2}$$

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

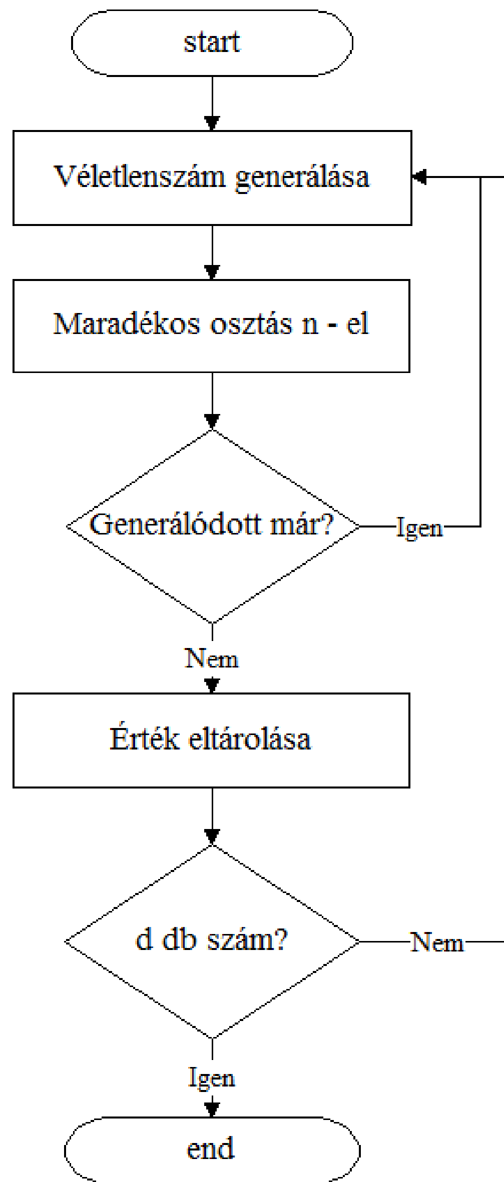
1.1. táblázat. Az elvégzett transzformáció

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* [12] 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 1.11. ábra szemlélteti.



1.11. 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

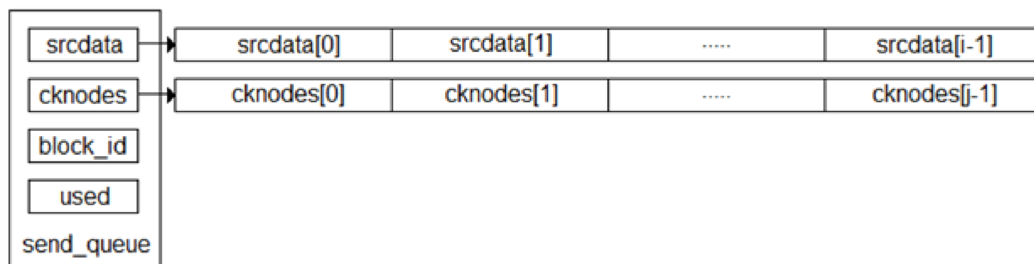
keletkezik az Y kimenő bájt, azaz $Y = X_{s1} \oplus X_{s2} \oplus \ldots \oplus X_{sd}$, ahol X_{si} jelenti az i - edik kiválasztott bájtot. Bájtok esetén ez bitenkénti *kizáró vagy* művelet elvégzését jelenti, tehát $\{u_1, u_2, \ldots, u_n\} \oplus \{v_1, v_2, \ldots, v_n\} = \{u_1 \oplus v_1, u_2 \oplus v_2, \ldots, u_n \oplus v_n\}$. 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.

1.7. Az adatokküldés

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.

1.7.1. 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 1.12. ábrán láthatjuk.



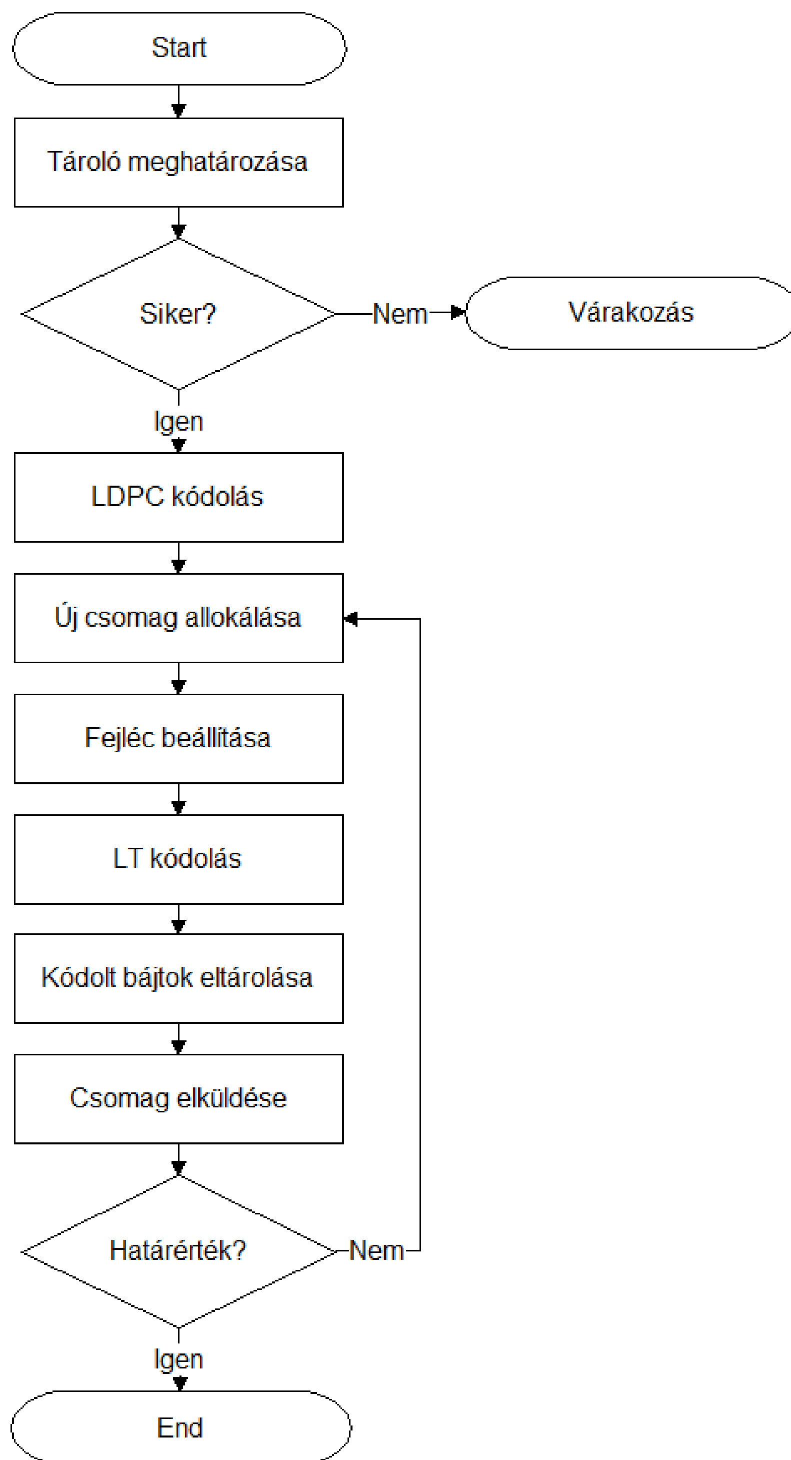
1.12. ábra. Egy küldési tároló felépítése

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.

1.7.2. 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 1.13. 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 új 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.



1.13. ábra. Az adatküldés folyamata

1.7.3. 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 1.13. 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.

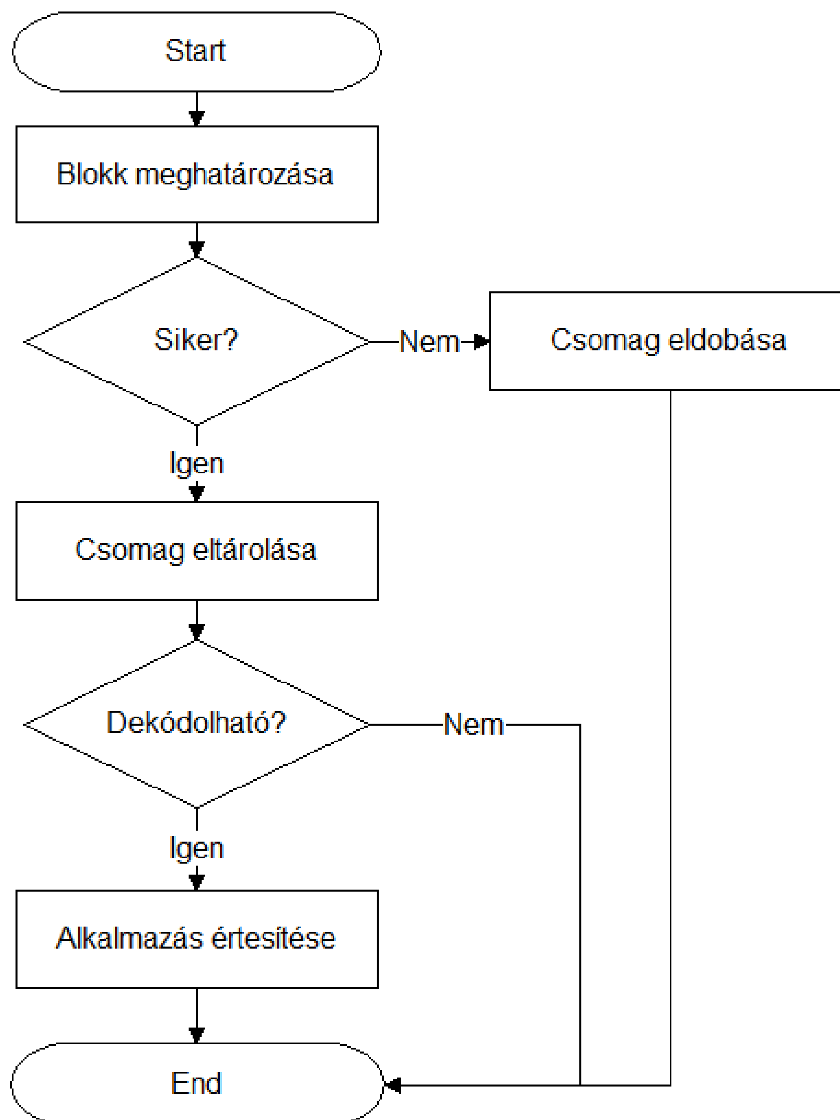
1.8. Az adatfogadás

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 1.14. 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.



1.14. 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 találtható 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.

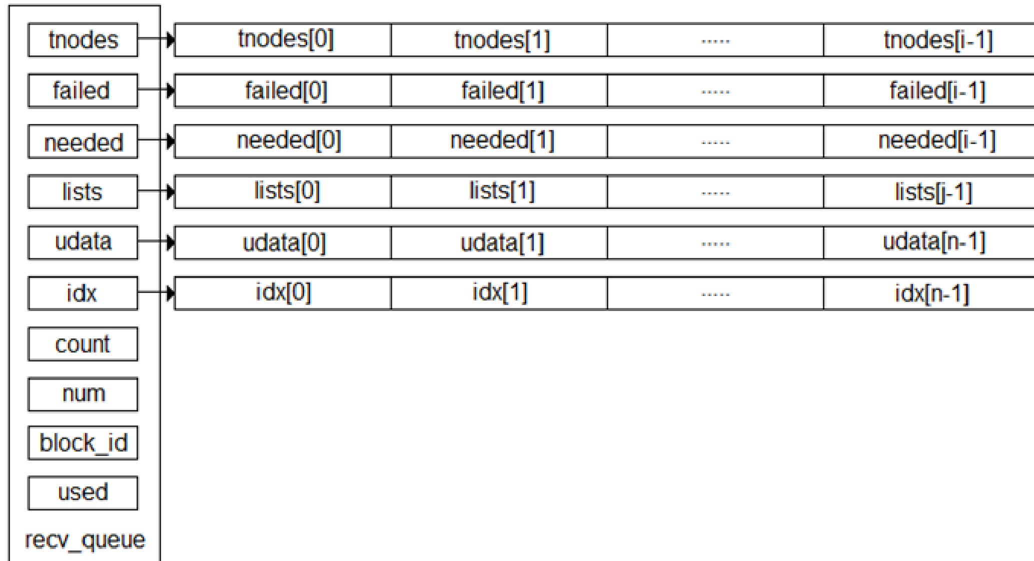
1.9. Az adatok dekódolása

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.

1.9.1. 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 1.15. 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.



1.15. ábra. Egy vételi tároló felépítése

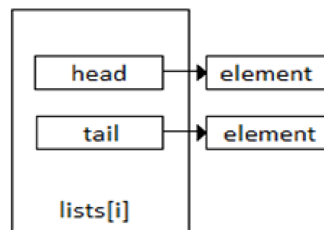
1.9.2. Az LT dekódolás

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 felszabadíthassa

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.

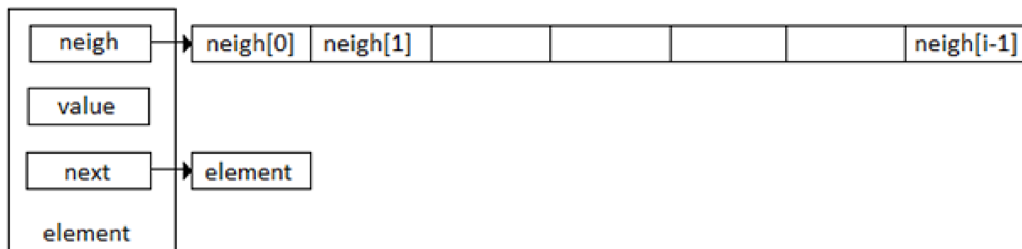
Az LT dekódolás esetén a 1.15. ábrán látható második három mezőt (*lists*, *udata*, *idx*) használjuk. A *lists* mező egy olyan mutatót tartalmaz, amely egy listákat tartalmazó összetett adatszerkezetre mutat. Az ábrán látható j értéke a jelenlegi implementáció esetén 10, tehát 10 listát használunk. Ezeket a listákat az összeköttetésben levő bájtok száma alapján különítem el. A lehetséges értékek a 1.1. 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 1.16. ábrán egy adott lista felépítését láthatjuk.



1.16. 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 1.17. ábrán mutatom be.



1.17. ábra. Egy listaelem felépítése

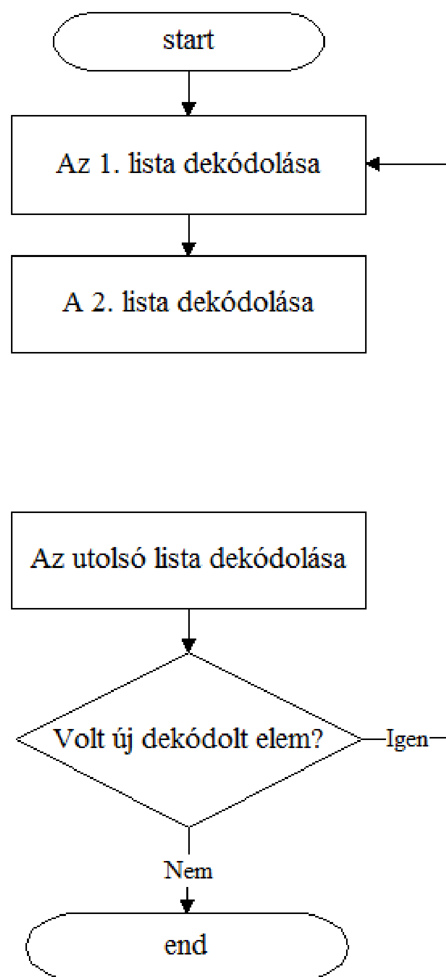
Minden listaelem tartalmaz egy mutatót (*next*), amely az adott lista következő elemére mutat. A *value* mező tartalmazza egyetlen kódolt bájt értékét azon bájtok közül amelyeket a bejövő csomag tartalmazott. A *neigh* tartalmazza azon szomszédok sorszámát, amelyek részt vettek a *value* kódolt bájt 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 1.18. 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.



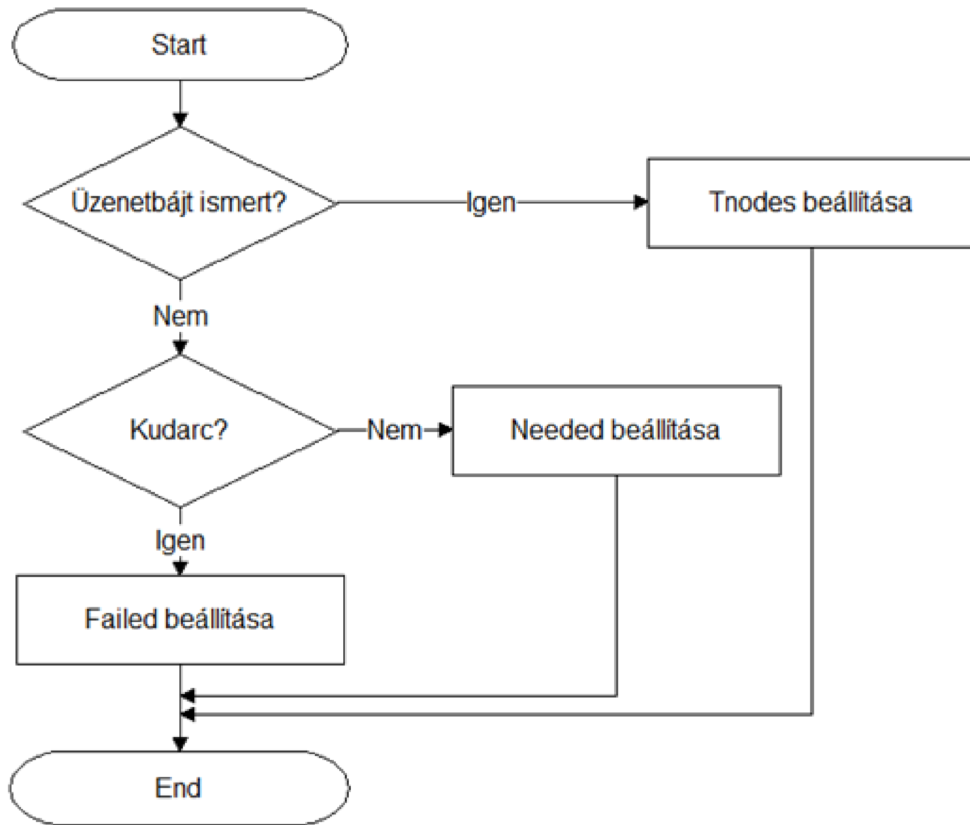
1.18. ábra. Az LT dekódolás folyamata

1.9.3. Az LDPC dekódolás

Az LDPC dekódolás esetén a 1.15. 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.

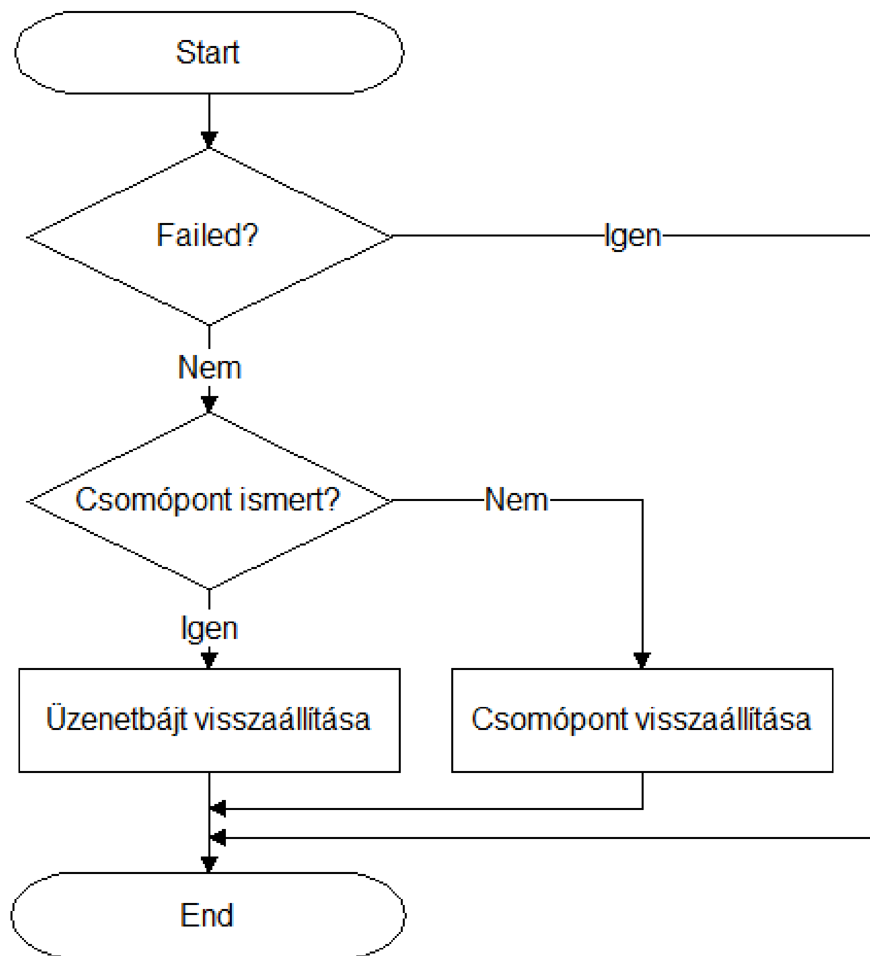
Az első lépés az inicializálás. Ennek során minden *tnodes[i]* értéket, és *failed[i]* értéket 0-ra állítunk be, *needed[i]* kezdeti értéke pedig 65536 lesz, mert ez egy olyan érték, amelyet egy sorszám már nem vehet fel, és így kezdeti értékként felhasználható annak jelzésére, hogy jelenleg érvénytelen 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 1.9. 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 1.19. 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.



1.19. 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 1.20. 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.



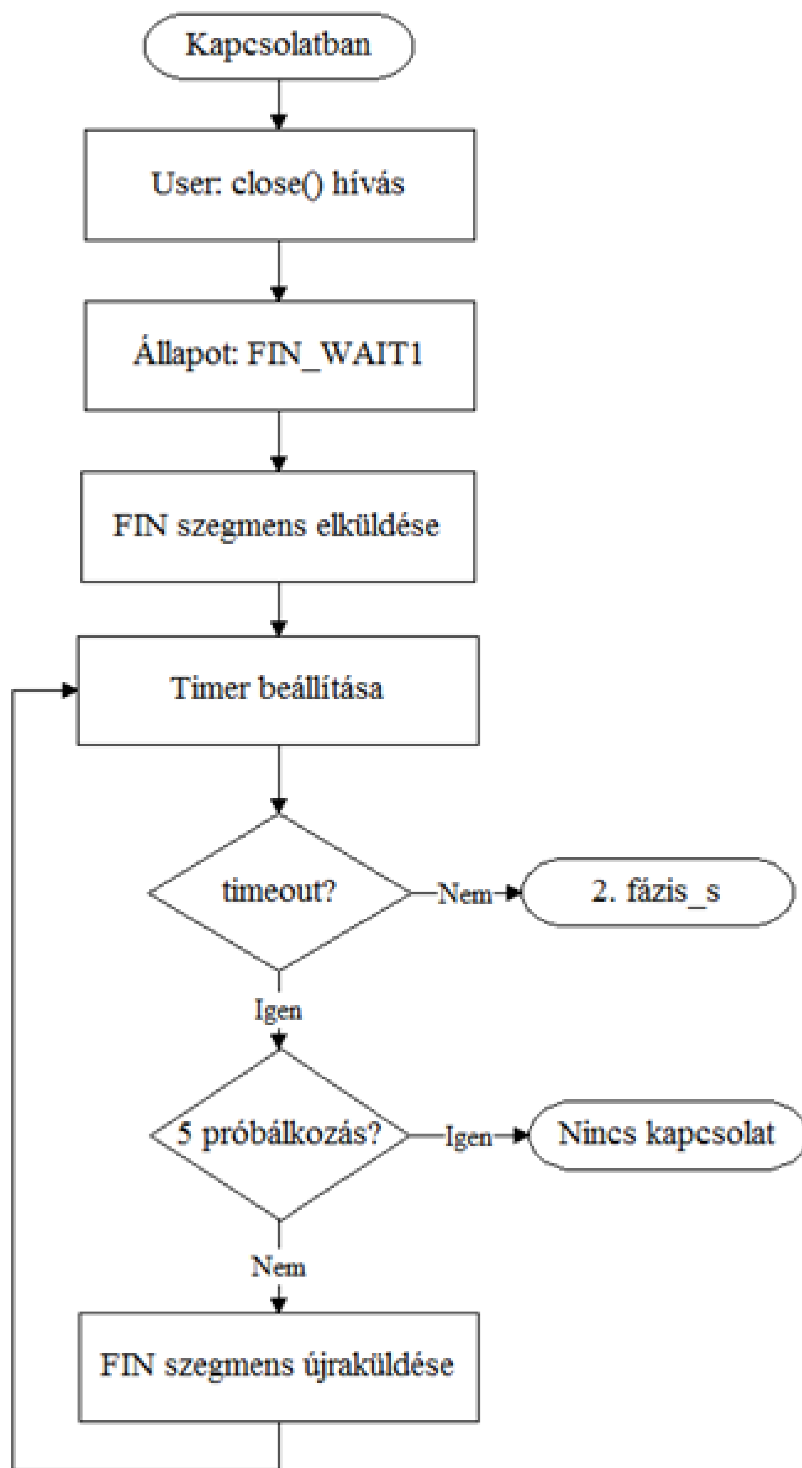
1.20. ábra. A 3. lépés, egyetlen ellenőrző csomópontra vonatkoztatva

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.

1.10. 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.

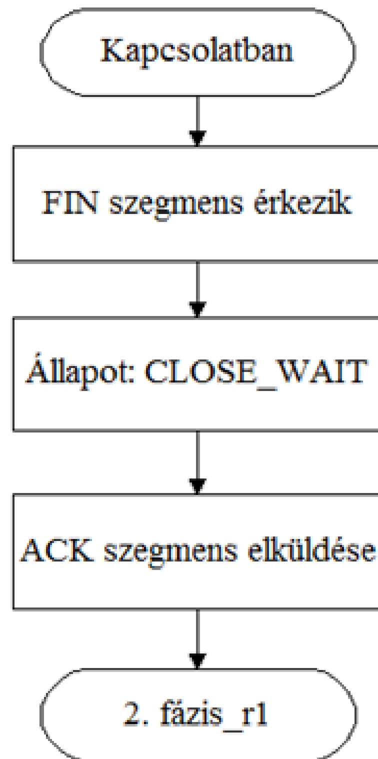
A kapcsolatbontás első lépése a 1.21. ábrán látható módon történik. Ennek során az oldal, amely bontani szeretné a kapcsolatot egy *FIN* szegmenst küld a másik oldalnak.



1.21. ábra. A kapcsolatbontás 1. lépése

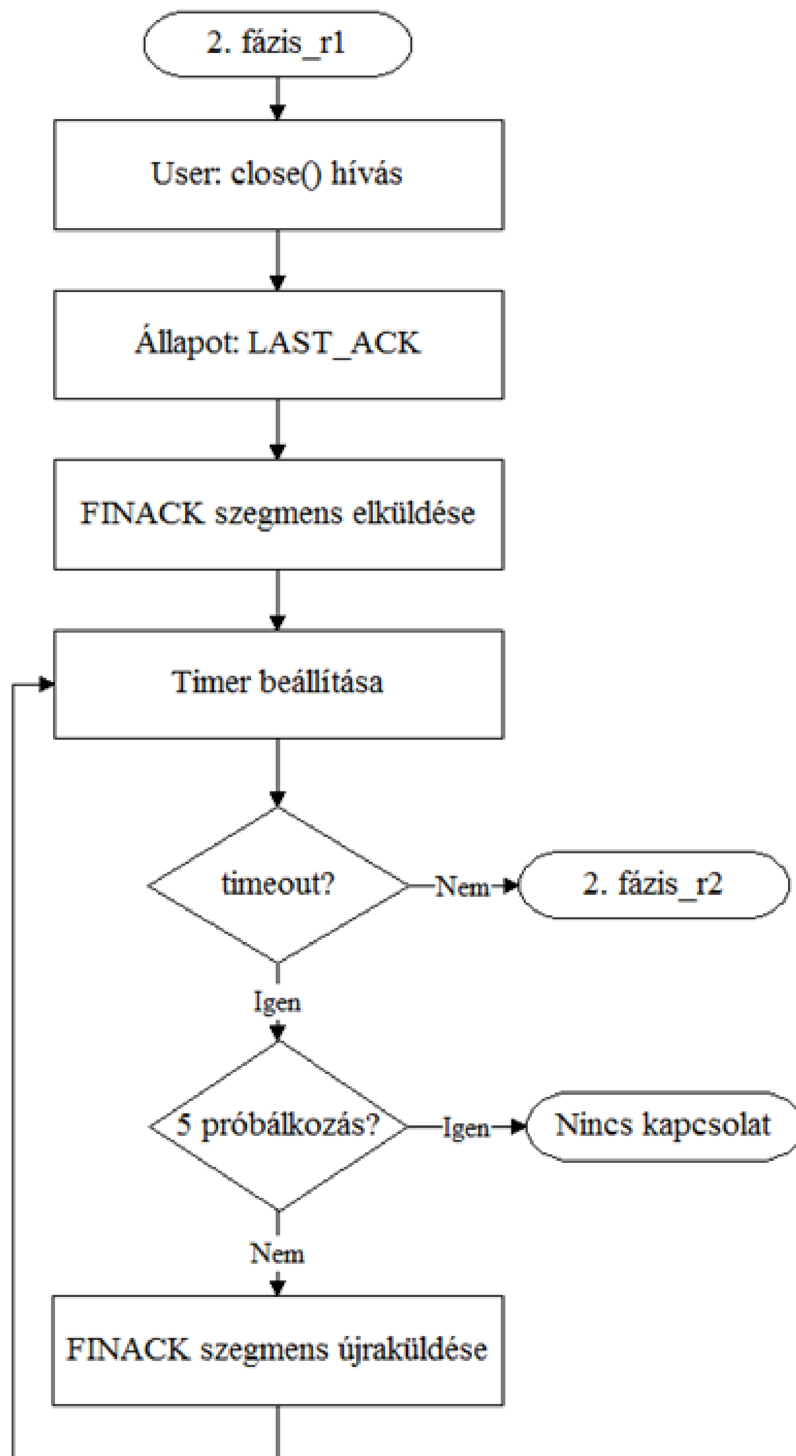
Az első lépés során a küldő állapota *FIN_ WAIT1* állapotra változik. Arra az esetre, ha a szegmens elveszne, időzítőt használok. A *FIN* szegmens esetén is maximum 5 alkalommal történik újraküldés. Az ábrán feltételezem, hogy ha nem történik időtúllépés, akkor megérkezett a nyugta a *FIN* szegmensre.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

A második lépést a 1.22. 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 1.22. ábrán látható módon.



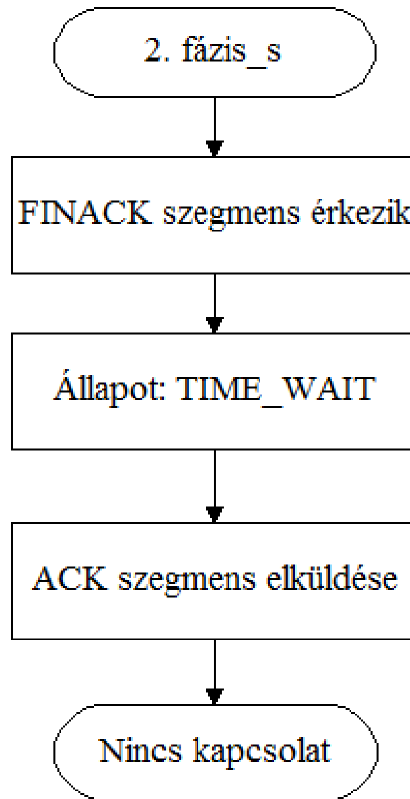
1.22. ábra. A kapcsolatbontás 2.a. lépése

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 1.23. ábrán láthatjuk.



1.23. ábra. A kapcsolatbontás 2.b. lépése

A harmadik lépést a 1.24. 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.



1.24. 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.

1.11. A protokoll paraméterei

Ebben a fejezetben a új 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 hatóköre 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 1.2. táblázatban foglalom össze. Az első paraméterrel szabályozhatjuk a kernel által lefoglalt küldési, és vételi tárolók számát. A második paraméterrel megadhatjuk az adatok küldésénél alkalmazandó határértéket

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

1.2. táblázat. A paraméterek jelentése

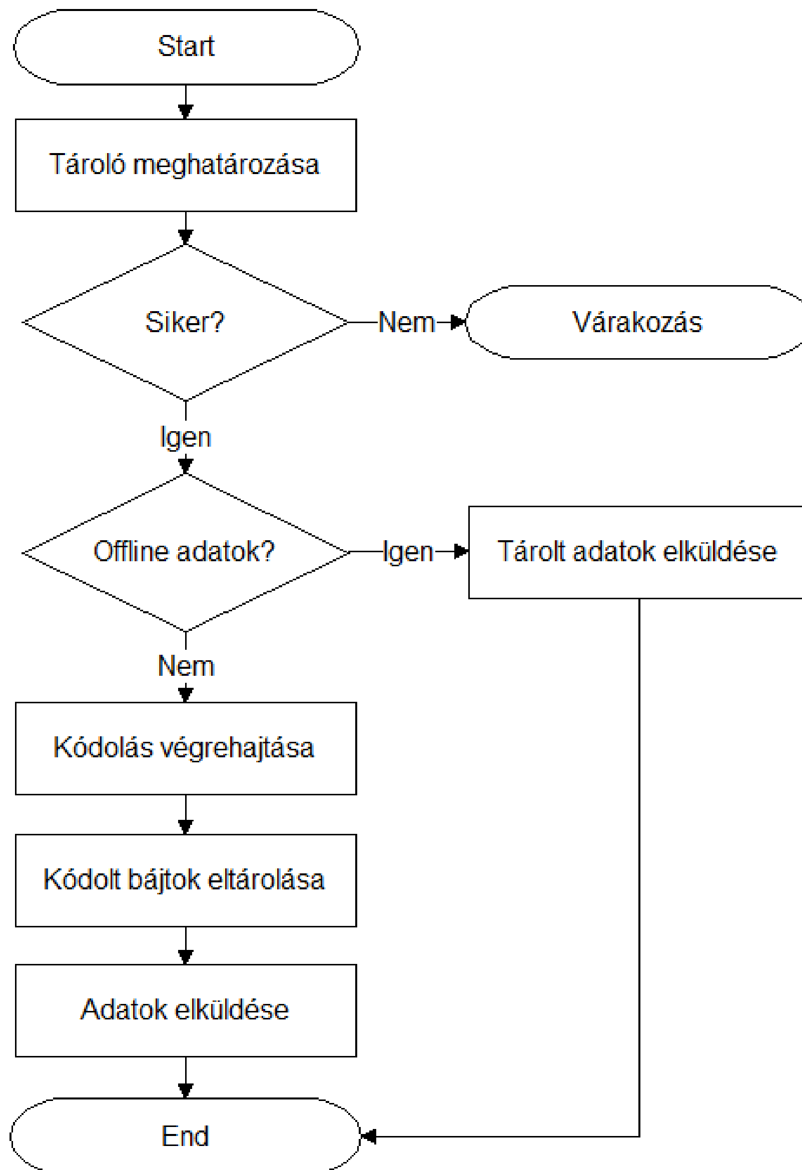
A paraméter neve	A paraméter funkciója
Maximális ablakméret	Az adatküldés során használt csúszóablak maximális méretének beállítása.
Redundancia	A blokkok küldése esetén használt határérték meghatározása.
Nyugtázás	A nyugtázás kikapcsolása/bekapcsolása a vevőnél.
Kódolás	Az adatok kódolásának kikapcsolása/bekapcsolása 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 1.13. 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 1.25. ábrán láthatjuk.



1.25. 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 1.14. 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.

2. fejezet

Összefoglalás

A tanulmányban 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 tanulmányban bemutatott protokoll egy lehetséges alternatívája lehet a jelenlegi TCP-nek. A protokoll jelen állpapotban egy folyamatban lévő kutatás eredménye. A protokoll Linux kernelben került implementálásra és jelenleg a mérési tesztelését végezzük.

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] 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.

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

[4] 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.

[5] 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.

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

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

[8] Solymos Szilárd. Robusztus, torlódásszabályozás nélkuli transzport protokoll tervezése és fejlesztése, 2011. BSc szakdolgozat.

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

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

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

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