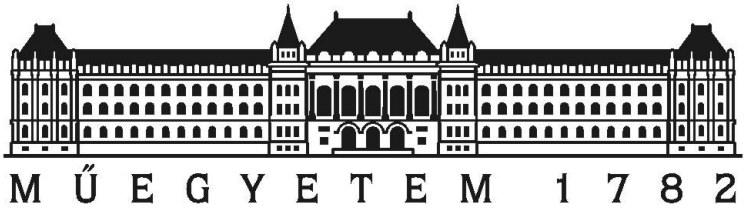


Transzport protokollok szimulációs elemzése

Performance Analysis of Transport Protocols

Készítette:	Abonyi Dániel
E-mail cím:	abonyi.dani@gmail.com
Konzulens:	Dr. Molnár Sándor
E-mail címe:	molnar@tmit.bme.hu

Budapest

2013.

Tartalom

Kivonat

A hagyományos TCP (Transmission Control Protocol) kulcsszerepet játszott az Internet sikertörténetében. Torlódásszabályozási mechanizmusa biztosította a hálózat stabil működését, és megóvta azt az "összeomlástól". Napjainkra azonban elérte hatékony működésének határait, és számos hálózati környezetben nem képes elfogadható teljesítményt nyújtani a felhasználók számára. Ezért a közelmúltban számos javaslat született arra, hogy hogyan lehetne javítani, és a mai hálózati igényekhez igazítani a TCP torlódásvezérlési mechanizmusait. Ezek az új mechanizmusok, illetve TCP verziók megjelentek a legújabb Linux kernelekben és az újabb Windows verziókban is, amelynek eredményeként a mai hálózati forgalom heterogén szabályozási mechanizmusok egymás mellett (illetve "egymás ellen") történő működéséből adódik. A jövő szempontjából természetesen alapvető fontosságú ezen verziók működésének és egymásra hatásának megismerése és széleskörű vizsgálata.

Ebben a dolgozatban a tanszéken folyó ilyen irányú kutatásokba kapcsolódtam be és egy hálózati szimulátor segítségével a fontosabb TCP verziók és egy új transzport mechanizmus, a DFCP (Digital Fountain based Communication protocol) protokoll összehasonlító teljesítményelemzését végezetem el. Az általam elemzésre használt eszköz a széles körben alkalmazott NS-2 hálózati szimulátor. Az új DFCP protokoll kernelkódját NSC kiegészítő segítségével építettem be az NS-2 környezetbe. Az elért eredményeket elemeztem és értékeltem.

Abstract

The traditional TCP (Transmission Control Protocol) played a key role in the success of the Internet. It's congestion control mechanism ensures the stability of the network and prevent it from congestion collapse. As time passed it reached it's operational limits, as it cannot provide acceptable efficiency in numerous network environments.

Therefore several suggestions were made to enhance the capabilities of the congestion control mechanisms used in TCP. These new mechanisms appeared in newer TCP versions, and in the kernels of the commonly used operating systems both in Linux and Windows. These resulted in today's heterogeneous network traffic consisting of competing flows. Regarding the future it is fundamentally important to investigate the influence of these competing flows.

I joined a related research project at the Department of Telecommunications and Media Informatics and executed the performance analysis of the well-known TCP protocols and also a new transport mechanism called DFCP (Digital Fountain based Communication protocol).

I performed the tests in the widely used NS-2 network simulator. In addition, I used the Network Simulation Cradle (NSC) to include the kernel code of the DFCP.

In this paper I presented the results of this simulation research.

1. Bevezetés

Az Internet megjelenése óta a hálózati csomópontok, a hálózatban található entitások közötti kommunikációt megvalósító réteges architektúra alakult ki. Ennek elkülönülő rétegei felelnek elhatárolható feladatokért, funkciókért: a fizikai réteg az adatok fizikai továbbításáért a megfelelő átviteli közegen, az adatkapcsolati réteg két szomszédos pont közötti átvitelért, a hálózati réteg adattovábbításért a hálózat bármely két pontja között, míg a szállítási réteg feladata a megbízható adatküldés két hálózati entitás között.

Az Internet megjelenése óta a legmeghatározóbb szállítási rétegbeli protokoll a Transmission Control Protocol (TCP). A protokoll megalkotásakor az akkori technológiai lehetőségek által meghatározott hálózati elemek képességeit kihasználó alkalmazást kellett tervezni. Külön figyelmet kellett fordítani arra, hogy a hálózati eszközök adta lehetőségek határait ne haladja meg a protokoll által támasztott igény, így kerülve el a hálózati túlterhelést és biztosítva a hálózat megbízható működését.

Az egyre nagyobb mennyiségű adatátviteli igények által támasztott növekvő követelmények maguk után vonták a hálózati elemek fejlesztését, cseréjét. Így lehetővé tették a megbízhatóbb átvitelt, és a nagyobb átviteli kapacitást, adatsebességet. A mára elterjedt hálózati megoldások képességei jelentősen megnövekedtek, az ezekkel felépített hálózaton megváltoztak a korlátozó tényezők.

Bár a régen tervezett transzport protokollok átviteli képessége egyes esetekben megfelel a mai nagysebességű hálózati kiépítettségen is, más esetekben teljesítményük jelentősen elmarad a kívántnál. Ennek oka a tervezésükkor előtérbe helyezett irányelv, hogy meg kell óvniuk a hálózatot az elárasztástól és a túlterheléstől. A mai nagysebességű hálózati szakaszokon fontos szempont a kiszolgálandó forgalom tervezhetősége, amely a TCP protokollok esetén nehéz, a protokoll működéséből adódó változó küldési sebesség miatt.

Egy új kutatási irány a mai hálózati igényeknek megfelelni kívánó protokoll elméleti leírása, amely a torlódásszabályozás elhagyása mellett foglal állást, így aknázva ki a torlódásvezérlés mentes hálózatok nyújtotta előnyöket.

A dolgozatban ismertetett vizsgálataim célja az egyes protokollok tulajdonságainak, viselkedésének, átviteli teljesítményének összehasonlítása a későbbiekben ismertetett topológiákon lefuttatott szimulációs esetsorok segítségével a valós hálózatban jelentkező hatásokra való tekintettel. E tényezők hatását a későbbiekben bemutatott hálózati szimulátor program segítségével modelleztem, majd az egyes szimulációs esetek lefuttatása után kapott eredményeket rendszerezetten mutatom be.

A továbbiakban ismertetem az elterjedt TCP protokollok változatait, majd az új irányelv alapján készült megoldási javaslatot, a torlódásvezérlés nélküli transzport protokollt is.

2. Transzport protokollok

2.1 TCP Protokoll

A TCP az OSI (*Open Systems Interconnection*) architektúra szerint a szállítási rétegben foglal helyet [1]. A TCP egy összeköttetés alapú protokoll, feladata a hálózatban helyet foglaló, egymás partnereiként funkcionáló elemek közti párbeszéd biztosítása, melynek eredménye hibamentes és megbízható adatátvitel bármely két entitás között.

Adatküldés előtt kapcsolatfelépítés szükséges, ezt a három-utas kézfogás valósítja meg. Ezután továbbíthatók a felsőbb rétegek felől érkező továbbítási kérelmek. Ezeket a küldés előtt a protokoll diszkrét méretű csomagokra osztja, majd továbbítja az alatta helyet foglaló rétegnek. Az adatküldés során a küldésre kijelölt csomagok mérete változhat a küldendő információ méretétől, illetve a felsőbb rétegek utasításaitól függően. Az adatküldés befejeztével szükség van a használt kapcsolat bontására.

A TCP forgalom-szabályozást is végez, amelynek szerepe a hálózat túlterhelésének megelőzése mellett a fogadó fél elárasztásának elkerülése is. A TCP-ben a torlódásszabályozás fő eszköze a torlódási ablak, amely azt szabályozza, hogy a küldő fél hány csomagot tarthat a hálózatban a fogadó fél visszajelzése nélkül. Ha a megszabott határt eléri, akkor a küldés a már hálózatban lévő csomagok vételi nyugtájának megérkezéséig szünetel.

A TCP protokoll első verziója 1974-ben jelent meg [2], azóta több továbbfejlesztett változata a már meglévő torlódásvezérlő eljárás javításával, módosításával kísérelt meg nagyobb hatékonyságot elérni [3].

Csomagvesztés bekövetkezése az eltérő protokollverziókból más reakciót vált ki. Másképp kezelik az elveszettnek ítélt csomag újraküldését, és a torlódási ablak megváltoztatását is ilyen esetekben. Az algoritmusok nem tudják elválasztani az átviteli közeg csomagveszési tényezőjéből adódó, illetve a hálózati túlterhelésből adódó csomagvesztést.

A protokoll mára elavult, gyengének ítélt pontja is ez, mert működése miatt nem képes megfelelő, elvárt hatékonyságú megoldást kínálni a gyorsan, elasztikusan változó heterogén hálózati forgalom lebonyolítására az Interneten.

A továbbfejlesztett protokollok teljesítőképessége az eredeti verzióhoz képest javult [4], de az összes verzió ugyanazon a koncepción alapul.

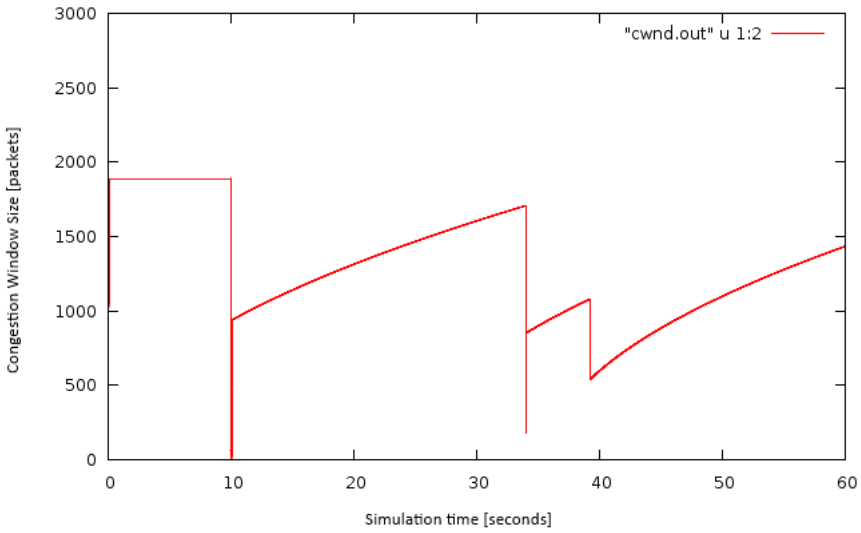
2.1.1 TCP Reno

Ez a változat gyors csomagvesztés detektálást tesz lehetővé. Az eljárás alapötlete, hogy a fogadó azonnal nyugtázza a csomag vételét a küldő felé. Ha egy csomag a sorrendtől eltérő helyen érkezik, akkor a fogadó fél duplikált nyugtát küld a forrásnak. Ez jó eséllyel csomagvesztés esetén történik, hiszen ha a sorban küldött csomagok közül egy vagy több elvész, az utána következő csomag, csomagok már nem a várt sorrendben érkeznek meg. A küldő fél az elveszettnek ítélt csomagot újraküldi.

Az adatküldés kezdetekor először a *slow start* fázisba lép, amelynek során minden elküldött csomagra érkezett nyugta hatására az ablakméretet exponenciálisan növeli. Ha az ablak mérete elért egy korlátot, melyet torlódási küszöbnek nevezünk, akkor a növekedés a továbbiakban lineáris. Csomagvesztés esetén a torlódásvezérlés a torlódási küszöböt az éppen aktuális ablakméret felére korlátozza, a torlódási ablakot a torlódási küszöb méretére állítja be. Ezt a módszert *congestion avoidance* algoritmusnak nevezzük,

melynek alapvető működése az AIMD (*Additive Increase Multiplicative Decrease*) mechanizmus.

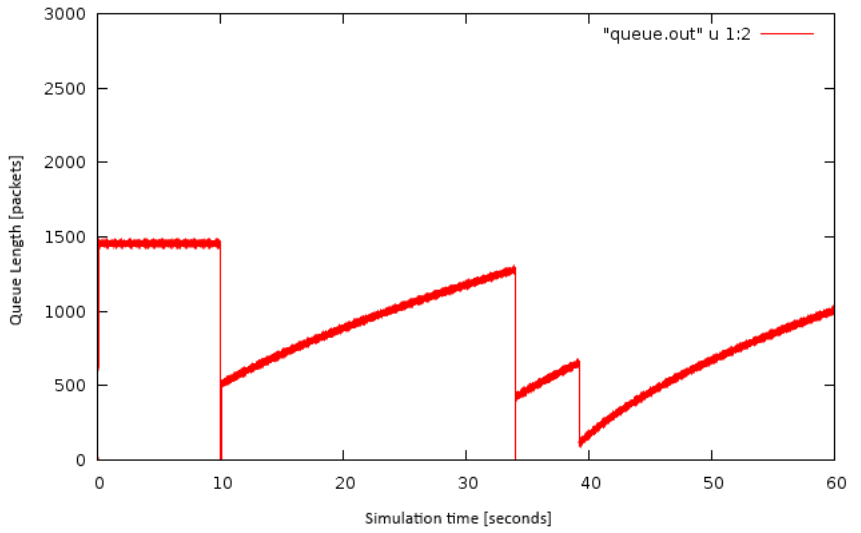
Az alábbi 1. ábrán a torlódásvezérlési eljárásra jellemző torlódási ablakméret látható minimális csomagvesztés mellett. Ablakméret a szimuláció idejének függvényében:



1. ábra: TCP Reno torlódási ablaka átvitel során

Adatküldés kezdetekor a protokoll exponenciálisan növeli a torlódási ablak méretét, amíg eléri a torlódási küszöböt. Ennek lefutása a szimuláció hosszúságához képest igen gyors. Majd a 10. másodpercben bekövetkezett csomagvesztés hatására a *congestion avoidance* eljárásnak megfelelően állítja be az értékét.

A 2. ábrán látható az adatküldés során felhasznált várakozási sor mérete. Ha az adatküldéskor használt linken elérhető maximális sebesség a korlátozó tényező az átvitel során, akkor a sor méretéből következtethetünk a hálózatban lévő, még nyugtázatlan csomagok számára. Sorhosszúság az idő függvényében:



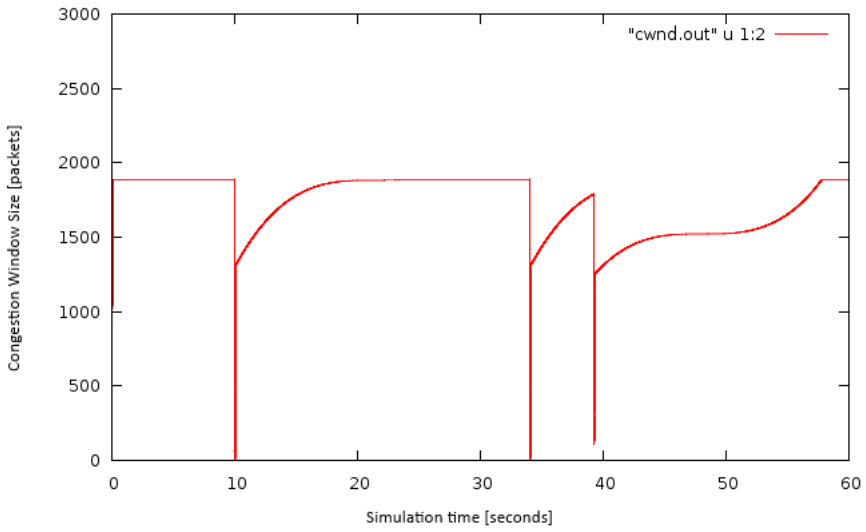
2. ábra: TCP Reno által használt várakozási sor alakulása

2.1.2 TCP Cubic

A TCP Cubic a mai Linux rendszerek kernelverzióiban alapértelmezettként elterjedt szállítási protokoll. A Cubic egy másik TCP verzió, a BIC (*Binary Increase Congestion control*) továbbfejlesztése. A BIC által alkalmazott torlódásvezérlési eljárás nagyon agresszív, de nagy késleltetés mellett is javulást ért el az eredeti protokollverzióhoz képest. Ennek ára a többi verzióval való gyenge együttműködés (*fairness*). A Cubic változat javítja az együttműködési képességet, és egyszerűbb torlódásvezérlést alkalmaz, de megtartja a BIC olyan kedvező jellemzőit, mint a skálázhatóság és a magas stabilitás.

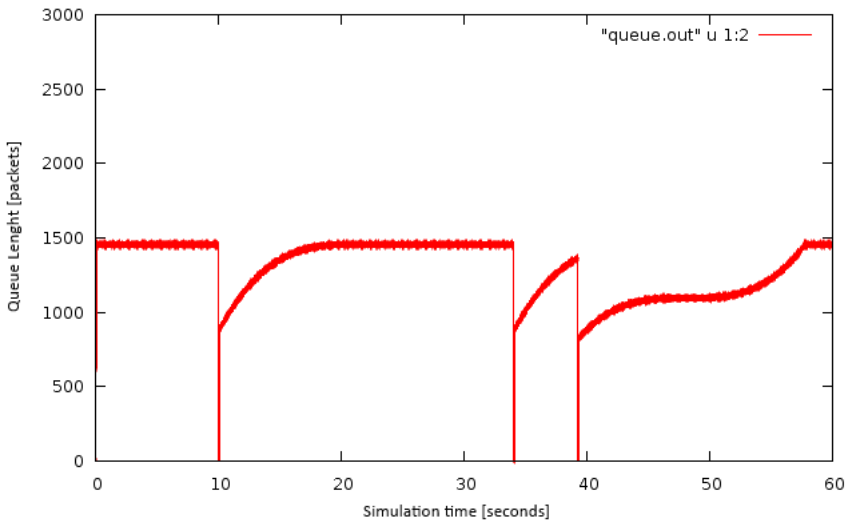
Adatküldés kezdetekor a protokoll exponenciálisan növeli a torlódási ablak méretét, amíg eléri a torlódási küszöböt. Ennek lefutása a szimuláció hosszúságához képest igen gyors. A 3. ábrán látható a 10. másodpercben bekövetkezett csomagvesztés hatására *additív növekedési fázisba* lép. Tehát abban a pillanatban minimális ablakméretnek kapott értékre állítja be a torlódási ablak méretét, majd fokozatosan csökkenő lépésekkel növeli azt a csomagvesztés előtti ablakméretig. Így képes elkerülni a hálózat túlterhelését a hirtelen ablaknövekedés következtében bekövetkező elárasztással szemben.

Ezután *bináris keresési fázisba* lép, amelyben a legutóbbi csomagvesztés előtti ablakmérethez közelíti a legutóbbi csomagvesztés után beállított értéket, eközben az aktuális ablakméretet a két érték között egyenlő távolságra állítja be. Ha a különbség elért egy minimális szintet, akkor *maximum keresési fázisba* lép. Ekkor megkísérli kihasználni a fennmaradó sávszélességet az ablakméret növelésével. Az alábbi 3. ábrán a torlódásvezérlési eljárásra jellemző torlódási ablakméret látható minimális csomagvesztés mellett. Ablakméret a szimuláció idejének függvényében:



3. ábra: TCP Cubic torlódási ablaka átvitel során

A 4. ábrán látható az adatküldés során felhasznált várakozási sor mérete. Sorhosszúság az idő függvényében (az előző esethez hasonlóan itt is a link átviteli kapacitása korlátozta az átvitelt):



4. ábra: TCP Cubic által használt várakozási sor alakulása

2.2 Digital Fountain based Communication Protocol (DFCP)

A mai igényeknek megfelelni kívánó protokoll alapkoncepciója a GENI (*Global Environment for Network Innovations*) által javasolt ötlet [5], amely szerint a hálózatban minden entitás az általa elérhető maximális sebességgel küldhet adatokat. Ha olyan a hálózati kiépítettség, és rendelkezésre áll kellő kapacitás, amely mellett nem jelentkezik torlódás, akkor ez egy igen hatékony megvalósítás. Maximális sebességű adatküldés mellett a hálózat kapacitásának véges volta miatt jelentkezik torlódás. Ez az adatfolyamban jelentős csomagvesztéssel jár, amelyre csomós eloszlás jellemző. A GENI által javasolt elv szerint ennek megoldása a hatékony, ciklikus hibajavító kódok használata.

A tanszéki kutatás céljaként kifejlesztésre került egy ilyen alapelven működő protokol: a Digital Fountain based Communication Protocol (DFCP) [6].

A TCP protokollokkal szemben ennek a megoldásnak az az előnye, hogy a hálózati forgalom könnyebben tervezhető, mert a protokoll működési elvéből adódóan nem alakulnak ki csomósan kiugró megnövekedett forgalmi igények. A hálózati elemekre hárul a torlódás kezelése, és a túlterhelésből adódó összeomlás elkerülése. Erre már meglévő hatékony ütemezési eljárások alkalmazása szükséges.

A protokoll az adatküldés előtti kapcsolat-felépítési fázisban a TCP-hez hasonló elven működő három-utas kézfogást alkalmaz. A felsőbb rétegek által küldési céllal átadott információn két kódolást hajt végre egymás után. Először egy paritásellenőrzést biztosító kódolást, amely a dekódolást segíti redundáns bájtok hozzávételével. Majd ezt követi egy szökőkút kódolás, amely ebből állít elő végtelen folyamot, amit továbbít az alsóbb rétegek felé.

A protokoll korlátozza azon csomagok számát, amelyek már kiküldésre kerültek, de vételükről még nem érkezett nyugta. A kódolás után akkor kerülnek kiküldésre a kódolt csomagok, ha nem ütköznek a csúszóablak méretének korlátozásába. A csúszóablak mérete azt határozza meg, hogy hány nyugtázatlan csomag lehet a hálózatban.

Adatküldés során a vevő a küldőtől beérkezett kódolt csomagokon kísérli meg a dekódolást. Ha ebből elegendő mennyiségű áll rendelkezésre, akkor a dekódolás sikeres, és nyugtával jelzi a küldő felé a sikeres továbbítást. Az adatküldés befejeződése után a TCP-hez hasonlóan le kell bontani a használt kapcsolatot.

3. Hálózati közegek általános jellemzése

Az adatátvitelre használt médián mindig jelen vannak olyan, a fizikai kialakításból adódó jellemzők, amelyek hátrányosan befolyásolják a közeg átvivő képességeit. Ilyen jellemzők az egyes linkek átviteli kapacitása, tehát a rajtuk elérhető adatsebesség, a linkeken jelentkező késleltetés és csomagvesztés, az egyes pontokon használt sorbaállítási eljárások, ezek maximális méretei, valamint a hálózatban szereplő, a forgalom függvényében kialakuló lehetséges szűk keresztmetszetek. Szimulációban lehetséges ideális kapcsolatok létrehozása, amelyeken a késleltetést és csomagvesztést nem vesszük számításba, azonban valós környezetben erre nincs lehetőség.

Ha egy hálózati csomóponton nem csak egy bemeneti kapcsolat van, akkor ezeken a kapcsolatokon keresztül érkező adatfolyamok versenyhelyzetbe kerülnek a hálózati erőforrásokért, ha összesített igényük magasabb, mint amit a hálózat biztosítani képes. Ilyen esetekben a versengő adatfolyamok teljesítményére jelentős hatással van a csomóponton alkalmazott várakozási sor típusa, és ennek eredményeként a csomagok továbbítási sorrendje.

3.1 Várakozási sorok, ütemezési eljárások

Várakozási sor típusok közül három hatását vizsgáltam, ezek: a *Droptail*, a *Stochastic Fairness Queuing* (SFQ) [7], illetve a *Deficit Round Robin* (DRR) [8].

3.1.1 Droptail

Olyan egyszerű ütemezési eljárás, amely egy várakozási sort használ. Telített várakozási sor esetén a további beérkező csomagokat eldobja. A mechanizmus hátránya is ez, mert a telített csomópontban eldobott csomag a TCP számára csomagvesztést jelent a teljes hálózatra nézve a két kommunikációs végpont között. Ennek következtében a torlódásvezérlő eljárás csökkenti a torlódási ablakot, negatívan befolyásolva a hasznos átviteli sebességet.

3.1.2 Stochastic Fairness Queueing

A teljes várakozási sor méretét felosztja több, kisebb méretű FIFO tárolóra [7]. A teljes forgalmat szétosztja ezen tárolók között, melyeket egy hash algoritmus fog össze. A beérkező csomagokat a küldő-fogadó címpár alapján rendeli a tárolókhoz. A csomagok kiküldése ezekből a tárolókból történik, a tárolók összességére nézve ciklikusan.

Az algoritmus akkor képes biztosítani az erőforrások igazságos elosztását, ha minden adatfolyamhoz hozzá tud rendelni egy tárolót. Ez megfelelően nagyszámú tároló esetén lehetséges. Ha két adatfolyam egy tárolóba kerül, akkor nem képes megakadályozni a versengés kialakulását.

Többféle változata is létezik, ezekben eltérő a listák megvalósítása és ezek összefogása. Ennek következtében eltérő mennyiségű művelet szükséges egy csomag behelyezésére a sorba, így csökkentve a számításigényt.

3.1.3 Deficit Round Robin

A *Round Robin* ütemező eljárás továbbfejlesztése [8]. A beérkező csomagok folyamok szerint felosztásra kerülnek, és külön sorokban várakoznak a továbbításra.

Minden sorhoz tartozik egy kvantum, aminek értéke megmutatja, hogy egy

várakozási sorból hány bájtot lehet továbbítani egy körben. Továbbá minden sorhoz hozzárendelünk egy értéket, amely nyilvántartja, hogy hány bájtot továbbíthat még az adott sor. Ez a *deficitszámláló*.

Minden különválasztott sor alapesetben üres. A beérkező csomagot az ütemező hozzárendeli egy várakozási sorhoz. Amelyik sorban legalább egy csomag van, annak sorszáma bekerül az úgynevezett *aktív listába*. Az ütemező minden kiszolgálási körben végighalad az aktív listán szereplő várakozási sorokon, és megkísérli az első csomag továbbítását ezekből.

Ha az elsőnek továbbítandó csomag mérete nagyobb, mint a deficitszámláló értéke, akkor a sort nem szolgálja ki az ütemező az adott kiszolgálási körben. Minden kiszolgálási kör végén az aktív listában szereplő várakozási sorokhoz tartozó deficitszámláló értékét megnöveli a kvantum értékével.

4. Hálózati szimulátor

4.1 Network Simulator 2

A szimulációk lefuttatását a Network Simulator 2 környezetben végeztem el [9]. A hálózati szimulátor *Tool Command Language* (TCL) nyelvű utasításokkal vezérelhető. A szimulációs környezetben megtalálható több hálózati elem (hálózati csomópontok, az ezeket összekötő linkek és átviteli közegek) modellje, várakozási sorok implementációi, többféle adatforgalmat szimuláló ágens programok, továbbá olyan modulok, amelyek lehetőséget biztosítanak a hálózaton áthaladó forgalom mérésére a szimuláció során.

Lehetőség van olyan megfigyelő elemek létrehozására, hálózatba csatolására, amelyek az egyes modulok viselkedésének, és a hálózatra gyakorolt hatásának vizsgálatát teszik lehetővé.

A parancsonkénti bevitel mellett lehetőség van az utasítások TCL/OTCL nyelven történő megadására is. Ekkor a megfelelő parancsok kötegelve kerülnek a szimulátorba, így hozva létre a megfelelő jellemzőkkel rendelkező szimulálni kívánt topológiát, a forgalmat generáló ágenseket, és a megfigyelő elemekhez tartozó naplófájlokat.

Ezután történik a szimuláció végrehajtása, amely során a szimulátor minden, az ágensek által a hálózatba bevitt csomag útját követi, és végigvezeti a hálózati elemeken, figyelembe véve azok hatásait.

4.2 Network Simulation Cradle

A program alapötlete az, hogy a Network Simulator környezeten belül lehessen végrehajtani a Linux operációs rendszer kerneljében implementált, széles körben elterjedt átviteli eljárásokat. Ezzel pontosabbá tehetők a szimulációs eredmények, és a valóság leképezése is pontosabbá tehető.

Ehhez az operációs rendszer kerneljét fel kell készíteni arra, hogy a szimulátor hozzáférhessen a normál használattól eltérő módon. Ennek az összeköttetésnek a biztosítására készült a Network Simulation Cradle (NSC) [10].

A Network Simulator által létrehozott teszteset futtatásakor a hálózati kommunikáció létrehozásához legalább kettő, de számos esetben több példányt is létre kell hozni egy adott típusú ágensből. Erre a kernelt külön fel kellett készíteni, mert a hagyományos implementációkban a csomagokat kezelő programok nem megszakíthatók, és nincsenek felkészítve a párhuzamos hozzáférésre sem [11].

Ennek feloldására az NSC megfelelő számú példányban teszi elérhetővé a kernelben található modulokat. A többszörös hozzáféréssel kapcsolatos problémákat úgy oldja fel, hogy az egyes példányok nem osztoznak adatokon.

4.3 Teszt szkript létrehozása

A szimulációk létrehozásához, futtatásához elkészített kiindulási szkript a függelékben található.

4.3.1 Ágensek beállítása

A mérések során a következő ágens paramétereket tekintem alapbeállításnak. Ha az egyes szimulációs esetekben ettől eltérek, azt külön jelzem.

TCP ágens paraméter	Érték	Magyarázat
window_	2422	A maximális torlódási ablakméret
packetSize_	1468	Az egy csomagban elérhető maximális hasznos adatméret
overhead_	0.000008	Az overhead hányadosa
max_ssthresh_	100	A z első csomagvesztéskor alkalmazott slow start paraméter
maxburst_	2	A maximális burst méret korlátozása
version_	0	A kernelben azonosítja a protokoll verziót

Mindkét ismertetett TCP protokoll esetén ezeket a beállításokat alkalmaztam. A kernelben alapértelmezettként a Cubic verzió szerepel. A Reno TCP-t igénylő szimulációkhoz át kell állítani a torlódásvezérlést is.

Ez a következő paranccsal végezhető el:

sysctl net.ipv4.tcp_congestion_control reno

Figyelni kell arra, hogy a torlódás-szabályozás átállítása **Reno TCP esetén csak az ágens hálózati csomóponthoz kötése után** lehetséges, ellenkező esetben ”segmentation fault” hibát kaptam.

DFCP ágens paraméter	Érték	Magyarázat
win_	100	Alpértelmezett ablakméret
ack_off_	0	Nyugták küldésének kikapcsolása
resend_	49	Csomagvesztés nélküli esetre számolt alapértelmezett érték
bandwith_	1000	Küldési sebesség értéke
version_	7	A kernelben azonosítja a protokoll verziót

DFCP protokollt használó ágens beállításai között két paramétert emelek ki:

- A *win_* paraméterrel a csúszóablak méretét lehet szabályozni.
- A resend_ paraméterrel a redundánsan hozzávett bájtok száma szabályozható.

Az utóbbi paraméter segítségével a nagyobb csomagveszési tulajdonsággal rendelkező közegen való átvitel esetén az átvitt információ dekódolható.

Az optimális resend_ paraméter mellett végzett átvitel biztosítja a dekódolhatóságot,

és maximális adatátviteli sebesség érhető el. A következő táblázatban az egyes csomagvesztési arányokhoz tartozó optimális redundancia paraméter értékek láthatók:

Packet Loss [%]	Resend
0.0	49
0.001	50
0.01	51
0.1	52
0.5	54
1.0	55
5.0	61
10.0	67
50.0	153

4.3.2 Linkek felépítése

Egy link két megadott csomópont közé hozható létre, olyan további paraméterek megadása mellett, mint az átviteli kapacitás, a linken használt sor fajtája és a késleltetés. A link becsatlakoztatásakor a szimulátor a létrehozott kapcsolatnak megfelelően tölti fel a szimulált topológia routing tábláját, így ennek megadásával nem szükséges foglalkozni.

A Network Simulator környezetben használt egyirányú link megadása a következő módon lehetséges [10]:

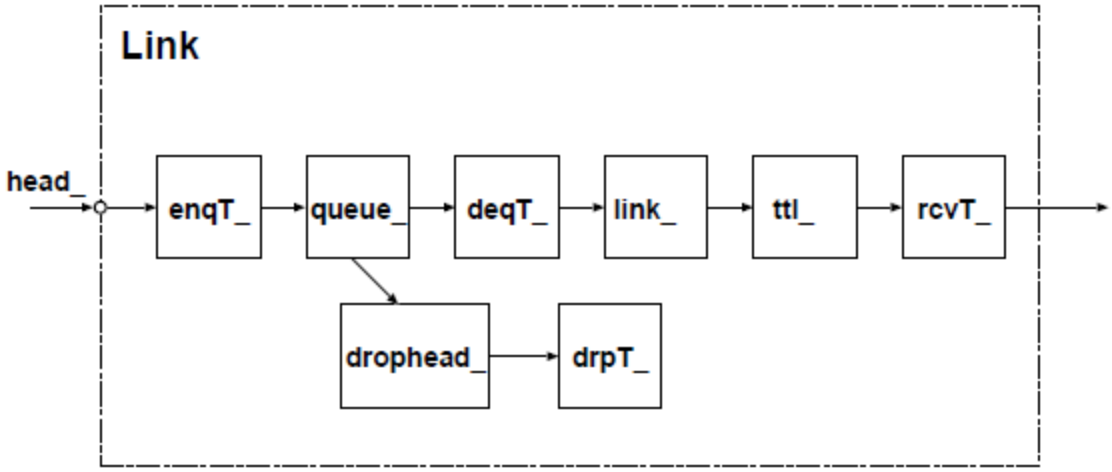
$ns simplex-link $<node1> $<node2> <brandwith [Mb]> <delay [ms]> <Queuetype>

Ahol a *node1* és *node2* jelenti a két, már előzőleg megadott csomópontot. A következő paraméterek pedig sorrendben a sávszélességet, a késleltetést, és a várakozási sort jelentik, a jelzett mértékegységekben.

Tehát egy 1000Mbps sávszélességet, 0 ms késleltetés és 10000 csomagnyi Droptail várakozási sort alkalmazó link megadása a következő módon történik:

```
$ns simplex-link $n1 $n0 1000Mb 0ms DropTail
$ns queue-limit $n0 $n1 10000
```

Az így létrejövő link felépítése az 5. ábrán látható. Az NS működésének és a link elemeinek bővebb leírása [12] találhatók, a *Links* alfejezetben.



5. ábra: Egyszerű link moduláris felépítése

A *head_* a link belépési pontja, a link első elemére mutat, és a beérkező csomagokat továbbítja a következő elemnek. A *queue_* elem a linken szimulált várakozási sornak felel meg. A várakozási sorba az *enqT_* elemen keresztül jutnak be a csomagok, majd a sort a *deqT_* elemen keresztül hagyják el. A link szimulációja ezután következik, erre a *link_* elem szolgál. Itt kerül sor a link késleltetési tényezőjének és sávszélességi kapacitásának szimulációjára. A *ttl_* elem a linken áthaladó csomagok *Time to Live* (TTL) értékét csökkenti a linken való áthaladáskor. A sikeres továbbítást a link *rcvT_* eleme szimulálja, és lépteti ki a linkről a sikeresen megérkezett csomagokat. A *drophead_* az eldobott csomagokat a *drpT_* elemre továbbítja, amelyen keresztül kikerülnek a szimulációból.

Szimuláció futtatása során a csomagok balról jobbra haladnak a linken belüli szimulációs lépésekben. Minden lépés a valós átviteli közegeken jelentkező hatások közül egyet modellez.

A kiindulási csomópontról kiküldött linkek a *head_* egységen keresztül kerülnek be a *queue_*-ba. Ekkor vagy eldobásra kerülnek a sor telítettsége miatt, és az eldobás után egy erre a célra szolgáló nyelőbe kerülnek, ahonnan nem továbbítódnak. Ha nem kerülnek eldobásra, akkor a sávszélesség korlátozás, és a késleltetés szimulációjával kerülnek a fogadó célcsomópontba az *rcvT_* elhagyása után.

Ebbe a struktúrába befűzhetőek további elemek, így lehetőség van további szimulációs lépések közbeiktatására a linken. Ezek között megtalálhatók a forgalmi statisztikák eléréséhez szükséges megfigyelő elemek, de a legfontosabb a csomagvesztést szimuláló modul.

Kétirányú link megadására is van lehetőség: ekkor két azonos tulajdonságú link jön létre ellenkező iránnyal. A két ellentétes irányú link egyesével is megadható egyirányú linkekként. Az eredményekre ez nincs hatással.

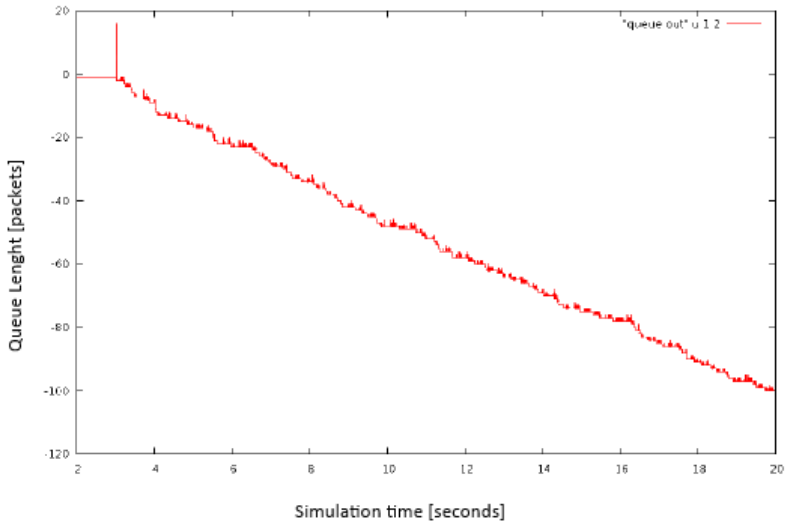
4.3.3 Csomagvesztés szimulációja

Csomagvesztés szimulációja az NS *error module* blokkjával lehetséges. Azért, hogy a modul kifejtse hatását, csatlakoztatni kell egy linkre. Ehhez az NS két lehetőséget kínál:

- Az egyik, hogy a modult a várakozási sor elé csatlakoztatjuk. Ekkor a hibásnak ítélt csomagok már a várakozási sorba kerülés előtt eldobásra kerülnek.

- A másik lehetőség a várakozási sor utáni csatlakoztatás. Ekkor a hibásnak ítélt csomagok csak a linkre küldés után kerülnek eldobásra.

A vizsgálatok során felfedtem az NS beállítási problémáit, ami hibás működésre vezetett. Nevezetesen a kezdetben futtatott szimulációs esetek eredményeinek vizsgálatakor az egyes kimeneti naplófájlok tartalma értelmezhetetlen volt. A 6. ábrán egy várakozási sor hossza látható az eltelt idő függvényében:



6. ábra: Helytelenül felépített link által generált sorhosszúság

A jelenséget az okozta, hogy a linkbe csatolható további elemek linkhez kötési sorrendje csak egyféleképpen helyes. Ha a modulok megfelelő sorrendben kerülnek a linkbe, akkor a naplók tartalma is értelmezhető lesz.

Ha a modulok nem a megfelelő sorrendben kerülnek a linkbe, akkor minden átviteli adat helyesen kerül a naplófájlokba, kivéve a sorméretet. Ami jellemzően tovább csökkenő értéket vesz fel a várakozási sorba bekerülő csomagok számától függő mértékben.

Tehát a jelenség nincs hatással az egész linken sikeresen átjutott csomagokra, így az átviteli kapacitásra sem a szimuláció során.

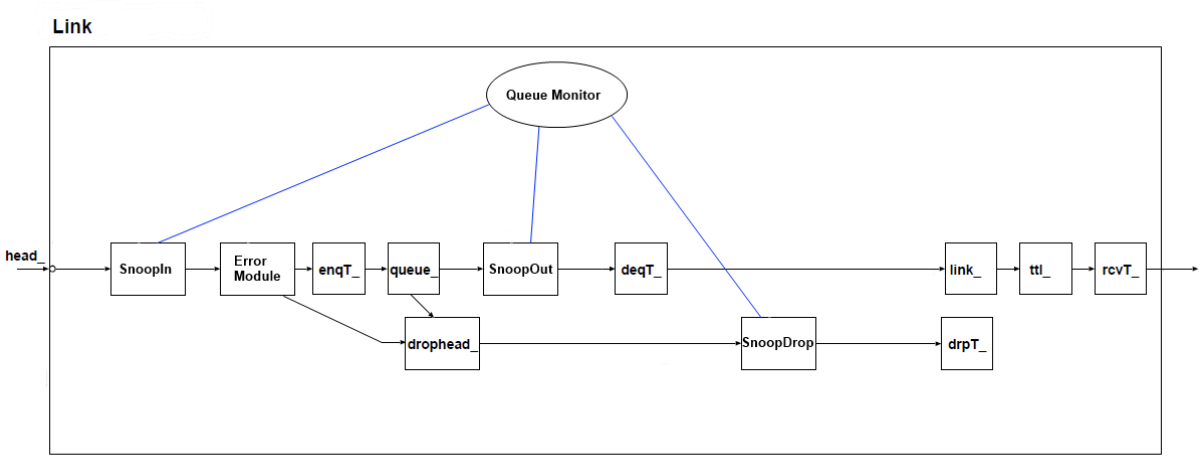
A következőkben ismertetem a helyes működést biztosító beállításokat, amelyek megfelelő használatával a fent ismertetett jelenség elkerülhető, és szimulációs környezet megfelelő eredményeket szolgáltat. A csomagvesztés szimulációjának becsatolására két lehetőség van a szimulátorban: közvetlenül a *head_* mögé, illetve közvetlenül a *link_* elé. Az helyes elrendezéseket is ennek megfelelően tárgyalom.

4.3.3.1 Csomagvesztés szimulációja a várakozási sor előtt

Ebben az esetben a csomagvesztést szimuláló modult közvetlenül a *head_* mögé csatoljuk a linkbe. Az ilyen szimulációk során a következő becsatolási sorrendet használtam a link létrehozásakor:

- Error Module
- Queue Monitor
- Flow Monitor

A következő 7. ábrán a teljes létrehozott link felépítése látható. A naplófájlok előállításáért felelős modulok a *SnoopIn*, *SnoopOut*, és *SnoopDrop* egységeken keresztül mérik a linken szimulált forgalom jellemzőit. Ilyen például a már említett Queue Monitor objektum, ami a sorméretet éri el a várakozási sort a *Snoopout* modulon keresztül.



7. ábra: Helyesen felépített link és csomagvesztés szimulációja a várakozási sor előtt

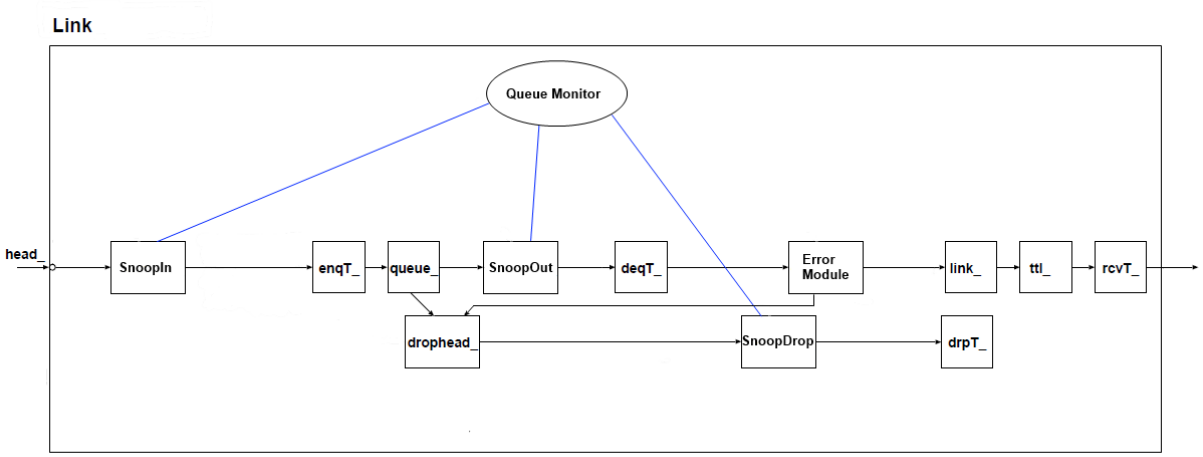
Jól megfigyelhető az eltérés az 5. ábrához képest. Az így létrehozott helyes felépítés megfelelően szimulálja a várakozási sor előtt a csomagveszést és biztosítja a forgalmi adatokat mérő modulok számára a helyes kapcsolódást, így az összes naplófájlba helyes adatok kerülnek be.

4.3.3.2 Csomagvesztés szimulációja a várakozási sor mögött

Ebben az esetben a csomagvesztést szimuláló modult közvetlenül a *link_* modul elé csatoljuk a linkbe. Az ilyen szimulációk során a következő becsatolási sorrendet használtam a link létrehozásakor:

- Queue Monitor
- Flow Monitor
- Error module

A következő 8. ábrán a teljes létrehozott link felépítése látható:



8. ábra Helyesen felépített link és csomagvesztés szimulációja a várakozási sor mögött

Jól megfigyelhető az eltérés az 5. ábrához képest. Az így létrehozott helyes felépítés megfelelően szimulálja a várakozási sor után a csomagveszést és biztosítja a forgalmi adatokat mérő modulok számára a helyes kapcsolódást, így az összes naplófájlba helyes adatok kerülnek be.

4.3.3.3 Csomagvesztés szimulációs lehetőségeinek összegzése

Az előző két pontban ismertetett esetek eltérő hálózati jellemzők szimulációjára alkalmasak:

A *4.3.3.1* pontban ismertetett felépítés olyan szimulációk elvégzésére alkalmas, amelyben a csomagvesztési paraméter a szimulált link előtti hálózat sajátossága, és ezt az összesített hatást egy lépésben vesszük figyelembe; a hibásnak ítélt csomagok még a várakozási sorba kerülés előtt eldobásra kerülnek. Majd a várakozási sorba bekerült csomagok ezután továbbítódnak az éppen szimulált szakaszon. Így a már hibásnak ítélt csomagok nem kerülnek kiküldésre és ezért nem foglalják a link átviteli kapacitását.

A *4.3.3.2* pontban ismertetett felépítés olyan szimulációk elvégzésére alkalmas, amellyel egy átviteli közeg hatását kívánjuk szimulálni, és ennek a közegnek a saját csomagvesztési, késleltetési tulajdonságainak hatását vizsgálni. Ilyen átviteli közeg lehet egy rádiós csatorna, ahol jellemző a magas csomagvesztési arány, illetve a változó, esetenként különösen magas késleltetés. Ugyanis a csomagvesztés a linkre küldés után kerül szimulálásra, így az ott elküldött olyan csomagok is foglalják a hasznos átviteli kapacitást, amelyeket az átvitel során hibásnak ítél majd a szimulátor.

4.3.4 Vizsgált ütemezési eljárások és várakozási sorok megadása

A következő példák 1000Mbps sávszélességet, 0 ms késleltetést állítanak be az *n1* és *n2* csomópontok között létrehozni kívánt linken, továbbá 10000 csomagnyi maximális sorhosszúságot állítanak be.

4.3.4.1 Egy adatfolyamos mérésekhez használt sorok

Droptail

```
$ns simplex-link $n1 $n0 1000Mb 0ms DropTail
$ns queue-limit $n0 $n1 10000
```

4.3.4.2 Két adatfolyamos mérésekhez használt sorok

SFQ

```
$ns simplex-link $n1 $n0 1000Mb 0ms SFQ
```

Az így létrejövő sor mérete azonban a C osztályokban jelenlévő default konstruktorok által beállított értékekkel jön létre. A sor méretét külön kell beállítani egy csak erre a sortípusra érvényes érték megváltoztatásával [13]:

```
Queue/SFQ set maxqueue_ 10000
```

A szimulációs mérések ezzel a sormérettel elakadtak, valamint a várakozási sor méretének naplófájljaiból kiderült, hogy a sor mérete ténylegesen csak 5000 méretű, mert két adatfolyam osztozik rajta.

Így a paramétert a kétszeresére emeltem, de eddig még nem foglalkoztam részletesen a várakozási sor korlátozó hatásával többfolyamos esetben.

Ezután a naplófájlokban is megfelelő sorméret jelentkezett (max. 10000), és a szimulációk is eredményesen lefutottak. Tehát SFQ sor alkalmazása mellett lefuttatott szimulációkhoz a következő beállítást alkalmaztam:

```
Queue/SFQ set maxqueue_ 20000
$ns simplex-link $n1 $n0 1000Mb 0ms SFQ
```

DRR

```
$ns simplex-link $n1 $n0 1000Mb 0ms DRR
```

Az így létrejövő sor mérete azonban a C osztályokban jelenlévő default konstruktorok által beállított értékekkel jön létre. A sor méretét külön kell beállítani egy csak erre a sortípusra érvényes érték megváltoztatásával [14]:

```
Queue/DRR set blimit_
```

Ez a beállítás azonban nem tesz lehetővé csomagszám szerinti megadást. Az így beállított érték bájtokban értendő. A maximális csomagméret (*MTU*) 1500 bájt lehet. Ezért a maximális sor méretét 15 millióban határoztam meg, és így kaptam a tízezer csomag méretű maximális méretet:

```
Queue/DRR set blimit_ 15000000
$ns simplex-link $n1 $n0 1000Mb 0ms DRR
```

4.3.5 Kimeneti fájlok meghatározása, és naplók készítése

Naplófájlok készítésére a szimulációs környezet széles lehetőségeket kínál. Az egyes hálózati elemek állapotát időbélyeges formátumban követhetjük nyomon a szimuláció egész időtartamára, az általunk megadott fájlokban. Figyelni kell arra, hogy az ilyen eljárások a szimuláció során igen gyakran futnak le, így az ezekben elvégezni kívánt aritmetikai műveleteket érdemes minimalizálni, majd eszerint implementálni. Ezzel igen számottevő számítási kapacitás takarítható meg, ez jelentősen (akár harmadával) rövidebb futási időben jelentkezik egy szkriptre nézve. Ily módon jelentős processzoridő takarítható meg nagy volumenű tesztelés esetén.

4.3.6 Adatátviteli sebesség meghatározása

Az adatátviteli sebesség meghatározását a következőképpen végeztem el: Minden teszt 60 másodpercig tartott, azonban az első 15 másodpercben jelentkező tranziens hatások miatt csak az utolsó 45 másodperc alatt átvitt adatmennyiséget mértem. Ehhez a nyelő ágens *bytes_* változóját használtam, amelyet az NSC szolgáltat. Ebben az overhead nélkül számított, tisztán átvitt adatmennyiség található. Így ezt az értéket normáltam a vizsgált időtartamra, tehát 45 másodperce. Ezzel megkaptam a későbbi összehasonlítás alapjául szolgáló, úgynevezett goodput adatsebességet.

Ebben a fejezetben leírtam a szimuláció sikeres elvégzéséhez szükséges beállításokat. Kitértem mindhárom, már ismertetett transzport protokollt használó ágens beállítására és helyes paraméterezésére. Továbbá várakozási sorok beállítására, SFQ és DRR ütemezők használata esetén a méret helyes megadására. Naplófájlok elkészítésére és a mérések során használt adatátviteli sebesség meghatározására, amely alapján az átviteli teljesítőképességet hasonlítottam össze. Továbbá külön figyelmet fordítottam az NS belső sajátosságából adódó problémára, és feltártam annak okát és pontos működését.

5. Vizsgált topológiák

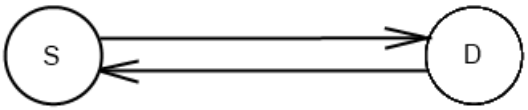
5.1. Egy adatfolyamot szimuláló topológia

Egy adatfolyam vizsgálatánál a topológia a küldő (forrás, source) és fogadó (nyelő, destination) csomópont párból, valamint az ezeket összekötő linkből áll. A két csomópont közötti kétirányú, duplex összeköttetést irányonként szimuláltam. Ehhez két, ellentétes irányú *simplex-link*-et alkalmaztam, így a mérésekhez szükséges paraméterek változtatása irányonként is lehetségessé vált.

Az egy folyamot tartalmazó méréseknél várakozási sornak droptail típust állítottam be, maximális méretét tízezer csomagban határoztam meg:

```
$ns simplex-link $n1 $n0 1000Mb 0ms DropTail
$ns queue-limit $n1 $n0 10000
```

Az így létrehozott, szimulálni kívánt hálózat vázlatos rajza az 9. ábrán látható:



9. ábra: Egy adatfolyamot tartalmazó topológia vázlatos rajza

Az így létrehozott egyszerű hálózaton vizsgáltam a már ismertetett (2. fejezet) három protokoll viselkedését a csomópontok közötti link késleltetés (RTT), csomagvesztés paramétere, és a várakozási sor hosszúságának változtatása hatására.

Minden paramétert a forrás csomópontból a nyelő felé irányuló linken alkalmaztam. Az ellenkező irányú linken ezektől a hatásoktól eltekintettem, mert a DFCP protokoll jelenleg nem képes az elveszett nyugták kezelésére.

5.2. Két adatfolyamot szimuláló topológia

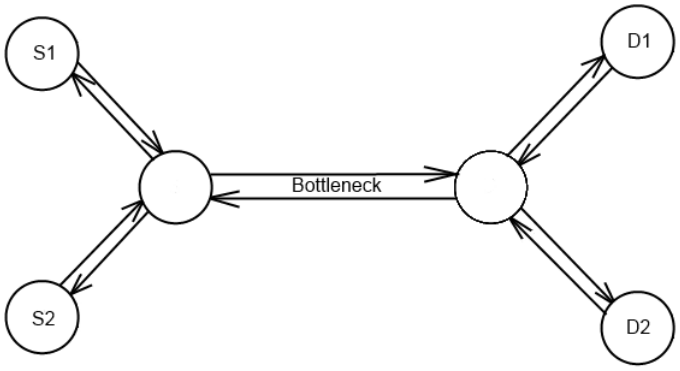
Ennek a szimulációs esetsornak a célja az, hogy két adatfolyam versenyhelyzetbe kerüljön egy szűkös hálózati erőforrásért. Ehhez két küldő-fogadó párt alkalmaztam, egy jelentősen alacsonyabb áteresztőképességű közös linken. Ez az úgynevezett *bottleneck*.

A bottleneck link szimulációjához az egyfolyamos esethez hasonlóan két ellentétes irányú simplex-linket alkalmaztam. A források és nyelők a bottleneck két végéhez két egyirányú simplex-linken keresztül csatlakoznak. A nyelő csomópontokban végződő linkeket ideálisra állítottam, tehát 0 ms RTT paramétert állítottam be ezeken, és csomagvesztést sem alkalmaztam.

A forrásból irányuló linkek közül az egyiken fix 10 ms RTT paramétert állítottam be, a másikon ezt a paramétert változtattam 0 és 100 ms között.

A várakozási sorok típusát az összes linken droptail-re állítottam, kivéve akkor, amikor a bottleneck linken az eltérő várakozási sor alkalmazásának hatását vizsgáltam.

Továbbá maximális méretüket tízezer csomagban határoztam meg. Az így létrehozott szimulálni kívánt topológia vázlatos rajza a 10. ábrán látható:



10. ábra: Két adatfolyamot tartalmazó topológia vázlatos rajza

A hálózaton azt vizsgáltam meg, hogy a fentebb ismertetett három protokoll hogyan viselkedik a forrás csomópontokból induló linkek késleltetéseinek változtatására, különböző várakozási sorok alkalmazása mellett.

A fent ismertetett két topológián lefuttatott szimulációs esetsorok segítségével összehasonlítottam az egyes protokollok viselkedését és átviteli teljesítményét a valós hálózatban jelentkező hatásokra való tekintettel. E tényezők hatását a szimulátor segítségével modelleztem, majd az egyes szimulációs esetek lefuttatása után kapott eredményeket rendszereztem.

Egyfolyamos esetekben megvizsgáltam a késleltetési tényező, és a csomagvesztési arány hatását az egyes protokollok átviteli teljesítményére. Kitértem e két tényező együttes hatására is az egy adatfolyamot tartalmazó szimulációs esetekben. Ezen felül egyfolyamos esetekben megvizsgáltam még a korlátozott sorméret által gyakorolt hatást is az átvitelre, továbbá a késleltetési paraméter hatását a várakozási sorok méretére az egyfolyamos topológián.

Két adatfolyamot tartalmazó szimulációs esetekben annak hatását vizsgáltam, hogy a szűkös átviteli kapacitás birtoklásáért kialakult versenyhelyzetben hogyan viselkednek az egyes protokollok több ütemezési eljárás használata mellett.

6. Eredmények

6.1 Egy adatfolyamot tartalmazó szimulációk

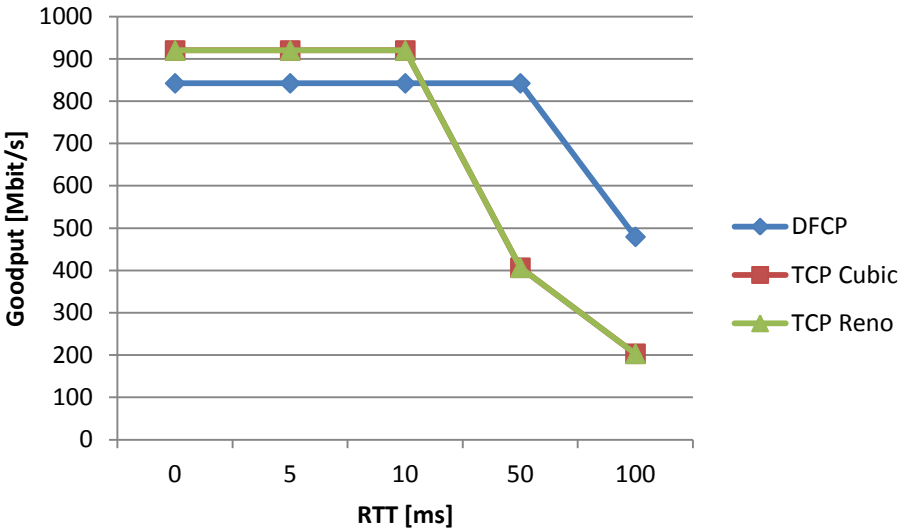
A továbbiakban az előző fejezetben megfogalmazott célok elérése érdekében elvégzett szimulációk rendszerezett eredményét tárgyalom. Az eredmények az egyes hatások szerint kerültek csoportosításra és összefoglalásra. Előbb az egy adatfolyamot tartalmazó eseteket, majd a két adatfolyamot tartalmazó eseteket mutatom be.

6.1.1 Késleltetés növekedésének hatása az adatátviteli sebességre

A szimulációkhoz a következő jellemzőkkel rendelkező topológiát hoztam létre:

- Egy folyamot szimuláló topológiát állítottam össze.
- A link átviteli kapacitását 1000 Mbit/s sebességre állítottam.
- A várakozási sor méretét 10 ezer csomagra állítottam.
- Nem alkalmaztam csomagvesztést.
- Droptail várakozási sort alkalmaztam a linken.

Ebben az esetsorban mindhárom, a már részletesen ismertetett protokoll átviteli teljesítményét mértem, miközben a késleltetés paramétert növeltem. A 11. ábrán láthatók az általam lefuttatott szimulációk összesített eredményei, az átviteli sebesség az RTT paraméter függvényében:



11. ábra

Jól látható, hogy a DFCP protokoll a nagyon magas késleltetéseket is jól kezeli, sokkal magasabb átviteli sebességet érve el. A TCP változatok eredményei megegyeznek.

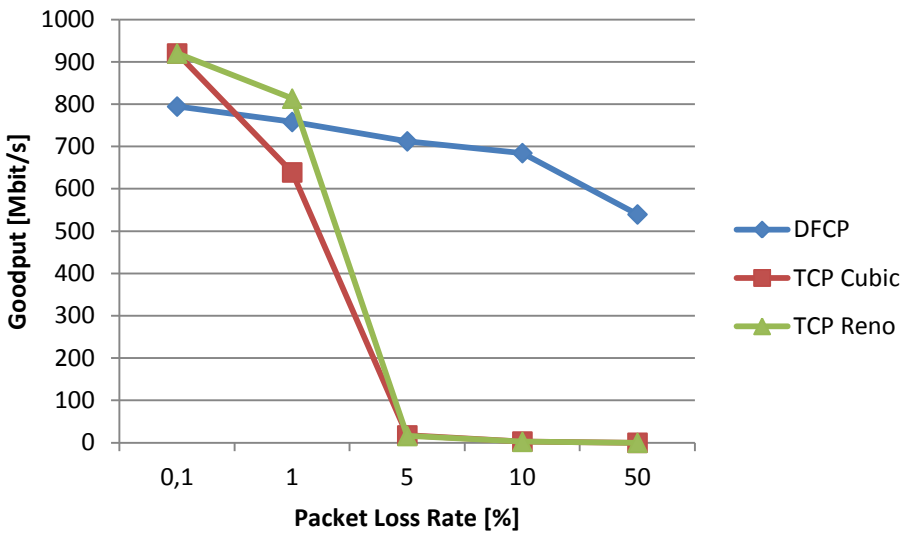
6.1.2 Csomagvesztés növekedésének hatása az adatátviteli sebességre

6.1.2.1 Várakozási sor előtt szimulált csomagvesztés

A szimulációkhoz a következő jellemzőkkel rendelkező topológiát hoztam létre:

- Egy folyamot szimuláló topológiát állítottam össze.
- A link átviteli kapacitását 1000 Mbit/s sebességre állítottam.
- A várakozási sor méretét 10 ezer csomagra állítottam.
- Késleltetésnek 0ms-ot állítottam be.
- Droptail várakozási sort alkalmaztam a linken.
- A csomagvesztés szimulációját a várakozási sor elé csatoltam be.

Ebben az esetsorban mindhárom, a már részletesen ismertetett protokoll átviteli teljesítményét mértem, miközben a csomagvesztés paramétert növeltem a fenti feltételek mellett. A 12. ábrán láthatók az általam lefuttatott szimulációk összesített eredményei, az átviteli sebesség a csomagvesztési arány növekedésének függvényében:



12. ábra

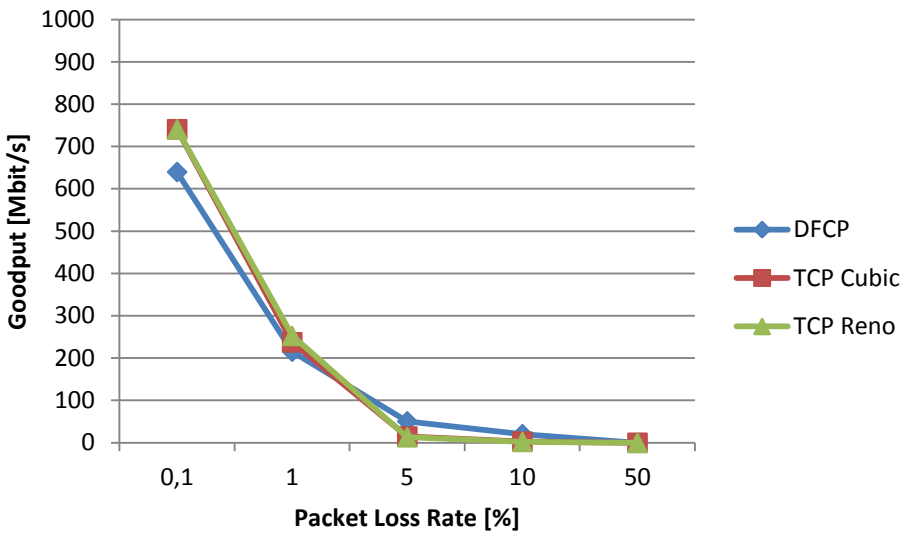
Kedvező jellemzőkkel rendelkező átviteli közeg esetén a TCP verziók jobban teljesítenek. Magas csomagvesztési arány esetén a DFCP *goodput* teljesítménygörbéje nem törik le úgy, ahogy a TCP verzióké.

6.1.2.2 Várakozási sor mögött szimulált csomagvesztés

A szimulációkhoz a következő jellemzőkkel rendelkező topológiát hoztam létre:

- Egy folyamot szimuláló topológiát állítottam össze.
- A link átviteli kapacitását 1000 Mbit/s sebességre állítottam.
- A várakozási sor méretét 10 ezer csomagra állítottam.
- Késleltetésnek 0ms-ot állítottam be.
- Droptail várakozási sort alkalmaztam a linken.
- A csomagvesztés szimulációját a várakozási sor mögé csatoltam be.

Ebben az esetsorban mindhárom, a már részletesen ismertetett protokoll átviteli teljesítményét mértem, miközben a csomagvesztés paramétert növeltem a fenti feltételek mellett. A 13. ábrán láthatók az általam lefuttatott szimulációk összesített eredményei, az átviteli sebesség a csomagvesztési arány növekedésének függvényében:



13. ábra

Ebben az esetben mindhárom transzport protokoll viselkedése igen hasonló. Az új protokoll teljesítménye is az eddig elterjedtekhez hasonlóan letörik. A TCP verziók teljesítménye közel azonos.

A két szimulációs beállítás eredményei közötti eltérés azzal magyarázható, hogy ha a csomagok közül a hibásakat még a várakozási sorba kerülés előtt eldobjuk, azok nem foglalják a hálózati erőforrásokat (a helyet a sorban, a sávszélességet a linken). Így az első beállítás mellett az alapvetően elárasztó viselkedésű DFCP protokoll jobban teljesít.

A csomagvesztés szimulációjának eltérő helyéből adódó különbségek alátámasztják a *4.3.3.3* pontban megfogalmazottakat.

6.1.3 Késleltetés és csomagvesztés együttes hatása

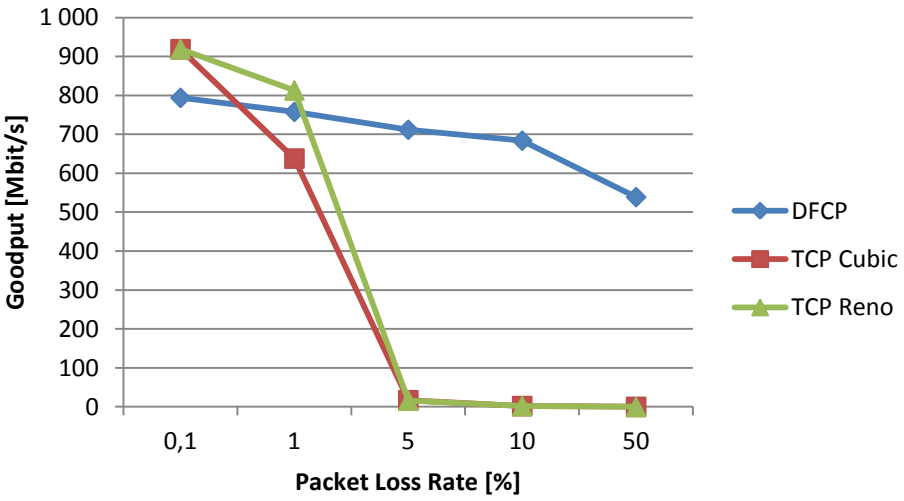
6.1.3.1 Csomagvesztés szimulációja a várakozási sor előtt

A szimulációkhoz a következő jellemzőkkel rendelkező topológiát hoztam létre:

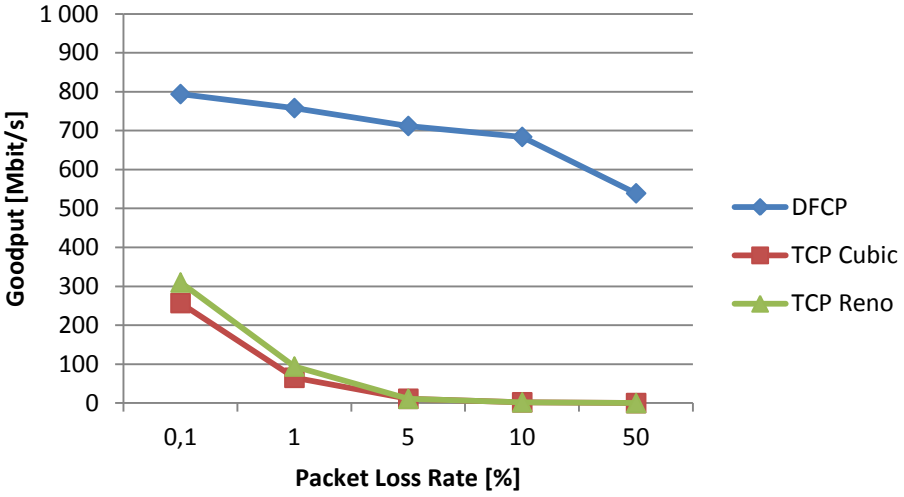
- Egy folyamot szimuláló topológiát állítottam össze.
- A link átviteli kapacitását 1000 Mbit/s sebességre állítottam.
- A várakozási sor méretét 10 ezer csomagra állítottam.
- Droptail várakozási sort alkalmaztam a linken.
- A csomagvesztés szimulációját a várakozási sor elé csatoltam be.

Ebben az esetsorban mindhárom, a már részletesen ismertetett protokoll átviteli teljesítményét mértem, miközben a csomagvesztés és késleltetés paramétereket növeltem a fenti feltételek mellett.

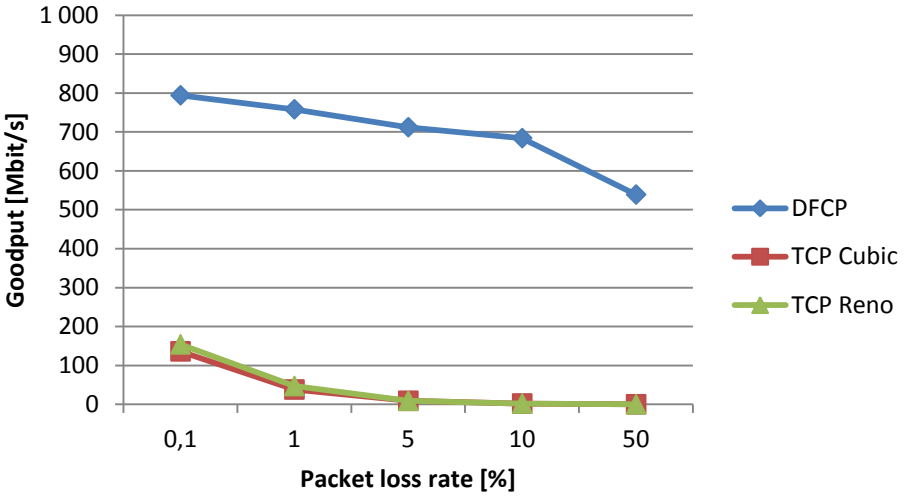
Az alábbi (14,15,16) ábrákon láthatók az általam lefuttatott szimulációk összesített eredményei, az átviteli sebesség a csomagvesztési arány növekedésének függvényében egyre magasabb késleltetés mellett.



14. ábra: 0 ms késleltetés mellett csomagvesztés növekedésének hatása



15. ábra: 1 ms késleltetés mellett csomagvesztés növekedésének hatása



16. ábra: 2 ms késleltetés mellett csomagvesztés növekedésének hatása

A TCP verziók által biztosított átviteli sebesség jelentős csökkenése látható a 14. és a 15. ábra adatai között alacsony csomagvesztési arányokra, alacsony késleltetés mellett is. A TCP protokoll ilyen feltételek mellett érzékeny a késleltetés kismértékű megnövekedésére is. Az összesített eredményekből látható, hogy a TCP verziók teljesítménye a két tényező együttes hatására jelentősen lecsökken.

Ezzel ellentétben a DFCP protokoll teljesítménye nem csökken olyan mértékben megfelelően beállított redundancia paraméter használatával.

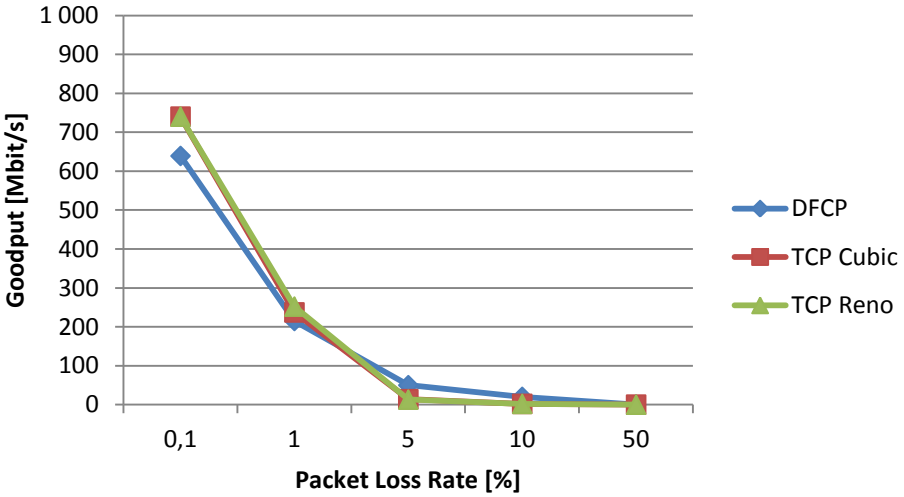
6.1.3.2 Csomagvesztés szimulációja a várakozási sor mögött

A szimulációkhoz a következő jellemzőkkel rendelkező topológiát hoztam létre:

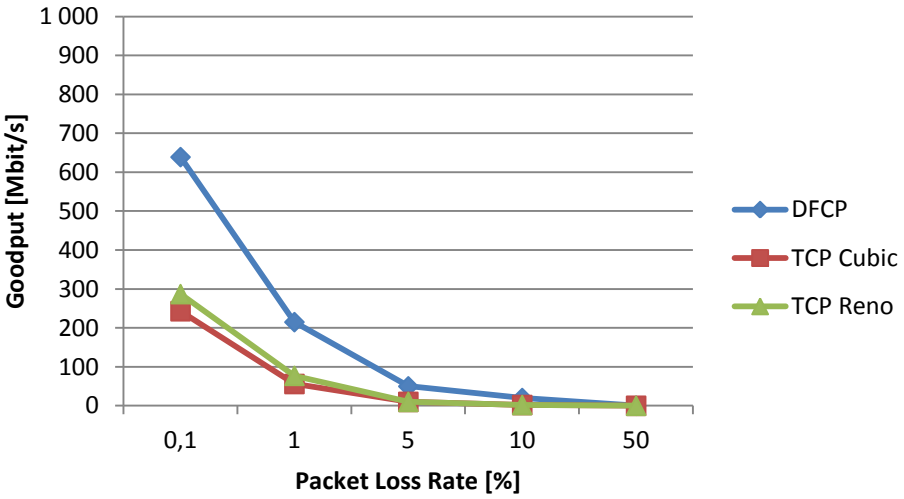
- Egy folyamot szimuláló topológiát állítottam össze.
- A link átviteli kapacitását 1000 Mbit/s sebességre állítottam.
- A várakozási sor méretét 10 ezer csomagra állítottam.
- Droptail várakozási sort alkalmaztam a linken.
- A csomagvesztés szimulációját a várakozási sor mögé csatoltam be.

Ebben az esetsorban mindhárom, a már részletesen ismertetett protokoll átviteli teljesítményét mértem, miközben a csomagvesztés és késleltetés paramétereket növeltem a fenti feltételek mellett.

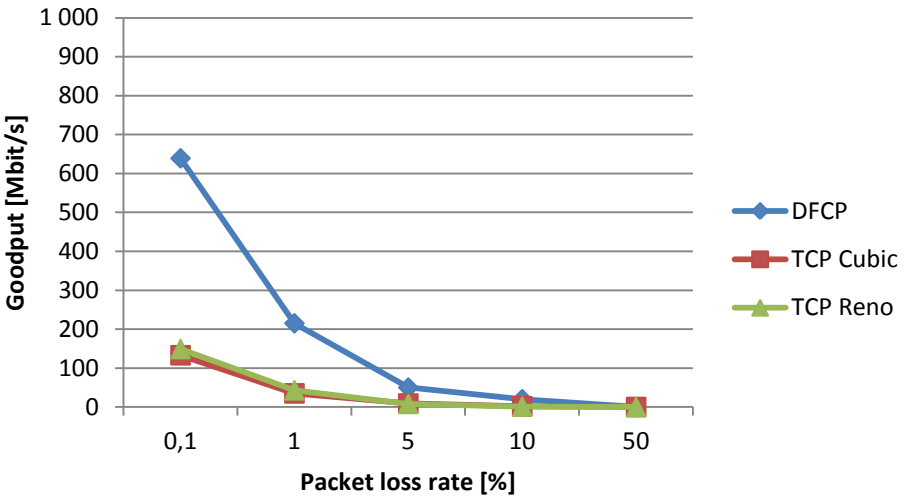
Az alábbi (17,18,19) ábrákon láthatók az általam lefuttatott szimulációk összesített eredményei, az átviteli sebesség a csomagvesztési arány növekedésének függvényében egyre magasabb késleltetés mellett:



17. ábra: 0 ms késleltetés mellett csomagvesztés növekedésének hatása



18. ábra: 1 ms késleltetés mellett csomagvesztés növekedésének hatása



19. ábra: 2 ms késleltetés mellett csomagvesztés növekedésének hatása

Magas csomagvesztési arány mellett jelentkező egyre növekedő késleltetés miatt a TCP protokollok teljesítménye erősen lecsökken. Magasabb késleltetés mellett még a vizsgált, alacsony csomagvesztési arányok mellett is egyre kisebb átviteli sebesség érhető el.

A DFCP protokoll teljesítménye az egyre magasabb késleltetés mellett jelentkező csomagvesztés hatására sem csökken jelentősen. Ez magyarázható a protokoll késleltetésére való kisebb érzékenységével. A sikeres átvitelhez szükséges a redundancia paraméter helyes beállítása is a csomagvesztési aránynak megfelelően.

Együttes hatás szimulációja esetén is jól megfigyelhető a link strukturális felépítéséből adódó eltérő viselkedés, amely szerint a várakozási sor mögött alkalmazott csomagvesztés sokkal jelentősebben befolyásolja az átviteli teljesítményt, mint a várakozási sor előtt szimulált csomagvesztés. Ezek a *4.3.3.3* pontban megfogalmazottaknak megfelelnek.

6.1.4 Várakozási sorméret változtatásának hatása az átviteli teljesítményre

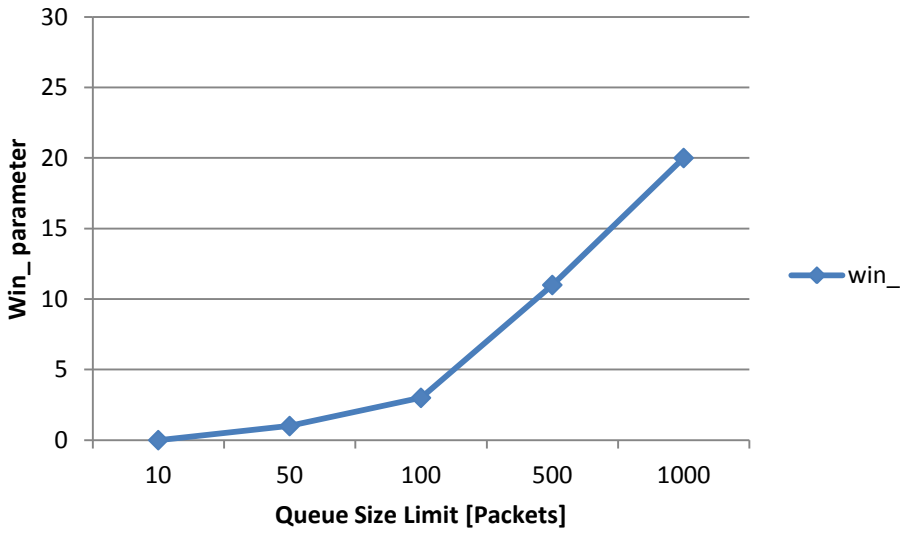
A szimulációkhoz a következő jellemzőkkel rendelkező topológiát hoztam létre:

- Egy folyamot szimuláló topológiát állítottam össze.
- A link átviteli kapacitását 1000 Mbit/s sebességre állítottam.
- Késleltetésnek 10ms-ot állítottam be.
- Droptail várakozási sort alkalmaztam a linken.
- Nem alkalmaztam csomagvesztést.

Ebben az esetsorban mindhárom, a már részletesen ismertetett protokoll átviteli teljesítményét mértem a fenti feltételek mellett, miközben a várakozási sor méretét változtattam. Továbbá statisztikát készítettem a ténylegesen felhasznált átlagos sorméretről.

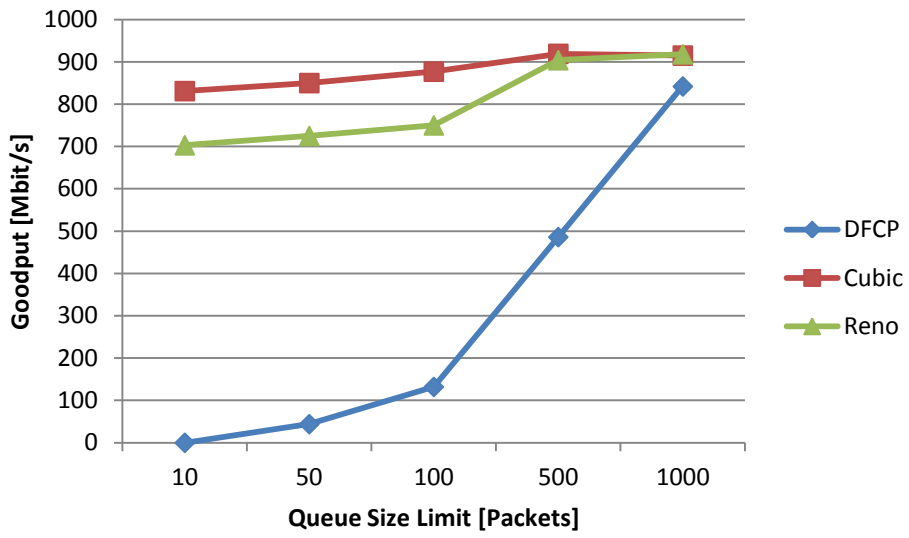
A szimulációk helyes elvégzéséhez szükséges volt meghatározni DFCP esetén az optimális ablakméretet. Az optimális ablakméret értelmezése ebben az esetben azt jelenti, hogy melyik az a legnagyobb ablakméret, amelyre az adatküldés még megtörténik. Ennél nagyobb ablakméret esetén az adatküldés megszakadna, mert a várakozási sor telítődne az adatküldés elején.

Az optimális ablakméretek a 20. ábrán láthatóak a várakozási sor hosszának függvényében:



20. ábra

A 21. ábrán látható az átviteli sebesség a várakozási sor hosszúságának függvényében:

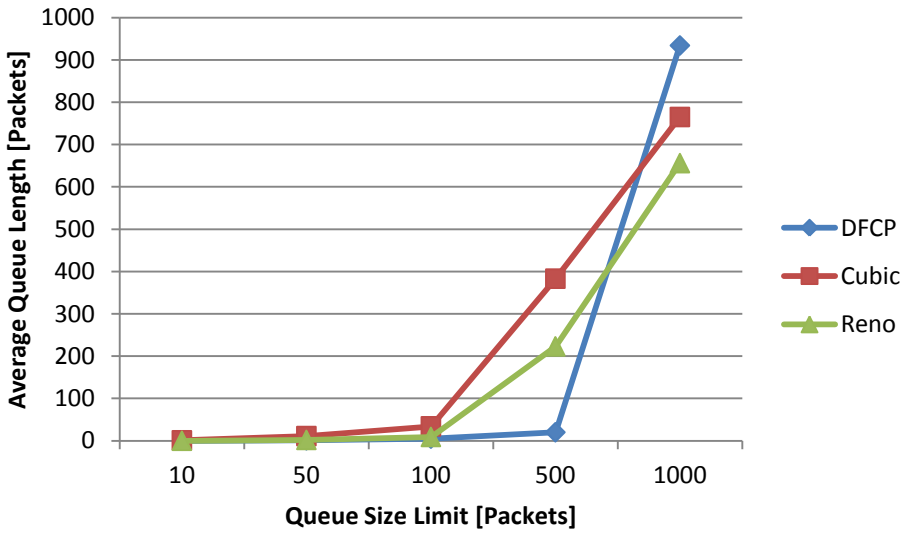


21. ábra

A TCP protokollok alacsonyra korlátozott várakozási sorméret mellett is megfelelő adatsebességet biztosítanak, megközelítve az elérhető maximális sebességet a linken. Megalkotásukkor fontos szempont volt az alacsony kapacitású hálózatokban való működés biztosítása.

A DFCP protokoll magasabb várakozási sorhosszúság mellett éri csak el a TCP verziók hatékonyságát ilyen feltételek mellett.

A 22. ábrán látható az egyes protokollok által használt átlagos sorhosszúság az átvitel során:



22. ábra

6.1.5 Késleltetés hatása a sorméretre

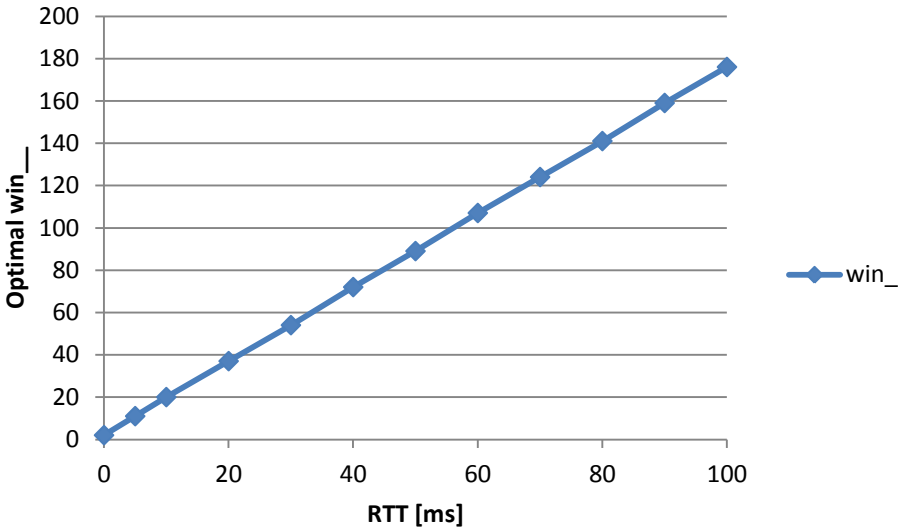
A szimulációkhoz a következő jellemzőkkel rendelkező topológiát hoztam létre:

- Egy folyamot szimuláló topológiát állítottam össze.
- A link átviteli kapacitását 1000 Mbit/s sebességre állítottam.
- A várakozási sor méretét 10 ezer csomagra állítottam.
- Droptail várakozási sort alkalmaztam a linken.
- Nem alkalmaztam csomagvesztést.

Ebben az esetsorban mindhárom, a már részletesen ismertetett protokoll átviteli teljesítményét mértem, miközben a linken jelentkező késleltetést változtattam. Továbbá statisztikát készítettem a ténylegesen felhasznált átlagos sorméretről.

A szimulációk helyes elvégzéséhez szükséges volt meghatározni DFCP esetén az optimális ablakméretet. Az optimális ablakméret értelmezése ebben az esetben azt jelenti, hogy melyik az a legkisebb ablakméret, amelyre az adatküldés során az átviteli sebesség eléri a linken elérhető maximális sebességet, és minimális sorhosszúságot használ fel ehhez. Tehát ennél nagyobb ablakméret beállításával már csak a sorhosszúság növekedne, a *goodput* már nem.

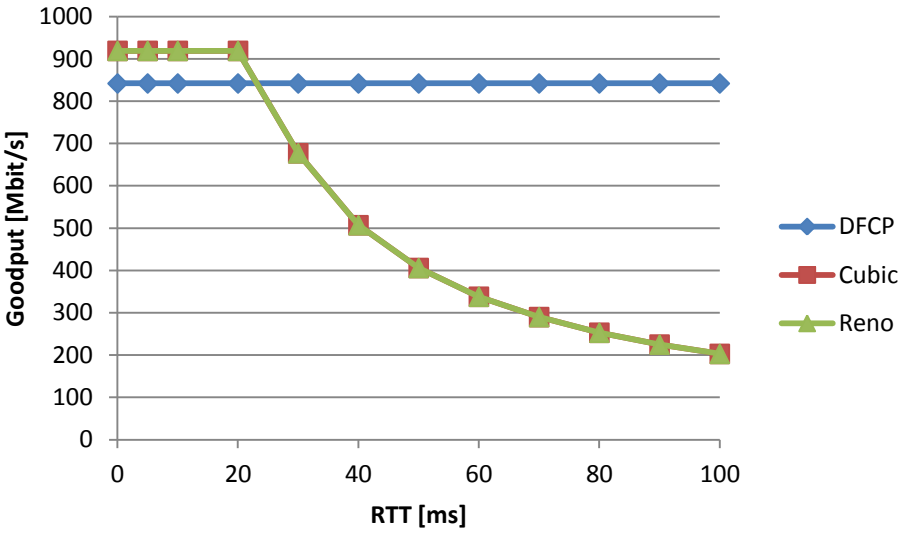
A 23. ábrán láthatóak az optimális ablakméretek a késleltetés függvényében:



23. ábra

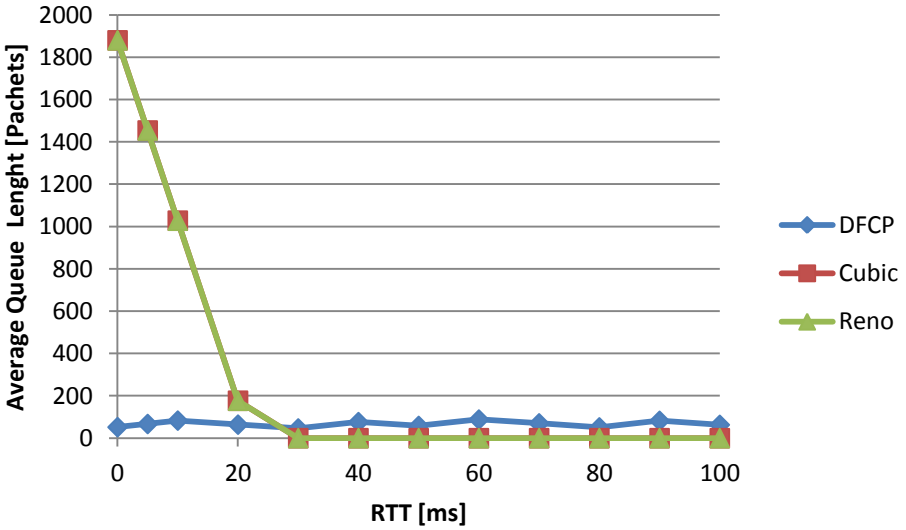
Az 1000 Mbit/s sávszélességű linken elérhető maximális *goodput* sebesség DFCP esetén 842 Mbit/s. Míg a TCP verziókra nézve a maximális *goodput* 919 Mbit/s.

A 24. ábrán látható az átviteli sebesség a késleltetés függvényében:



24. ábra

A 25. ábrán látható az átvitel során használt átlagos sorhosszúság az egyes protokollokra:



25. ábra

Az eredményekből jól látható, hogy a DFCP protokoll a mért késleltetési paraméterekre nem érzékeny annyira, mint a vizsgált TCP változatok.

A várakozási sor telítettségét is alacsonyan lehet tartani, ha a DFCP *win_* paraméterére a fent megfogalmazott optimális beállítást alkalmazzuk.

Ebben a fejezetben összefoglaltam az egy adatfolyamot tartalmazó topológián elvégzett szimulációim eredményét. Az egyes szimulációs esetsorok összesített eredménye azt mutatták, hogy a helyesen paraméterezett DFCP protokoll jól tűri a magas csomagvesztési arányt a hálózat egészére nézve, továbbá magas késleltetés mellett is magasabb átviteli sebességre képes, mint a vizsgált TCP változatok.

A DFCP protokoll megfelelő paraméterbeállításával a teljes vizsgált késleltetési tartomány alatt alacsony sorhosszúságot igényelt. Ez kedvező tulajdonság például optikai

átviteli közegekre nézve, ahol nincs lehetőség nagyon magas várakozási sorok kialakítására.

Ugyanakkor a DFCP hátrányaként a vizsgált esetekben a nagyon alacsonyra korlátozott várakozási sorkapacitás melletti alacsony átviteli teljesítményt figyeltem meg.

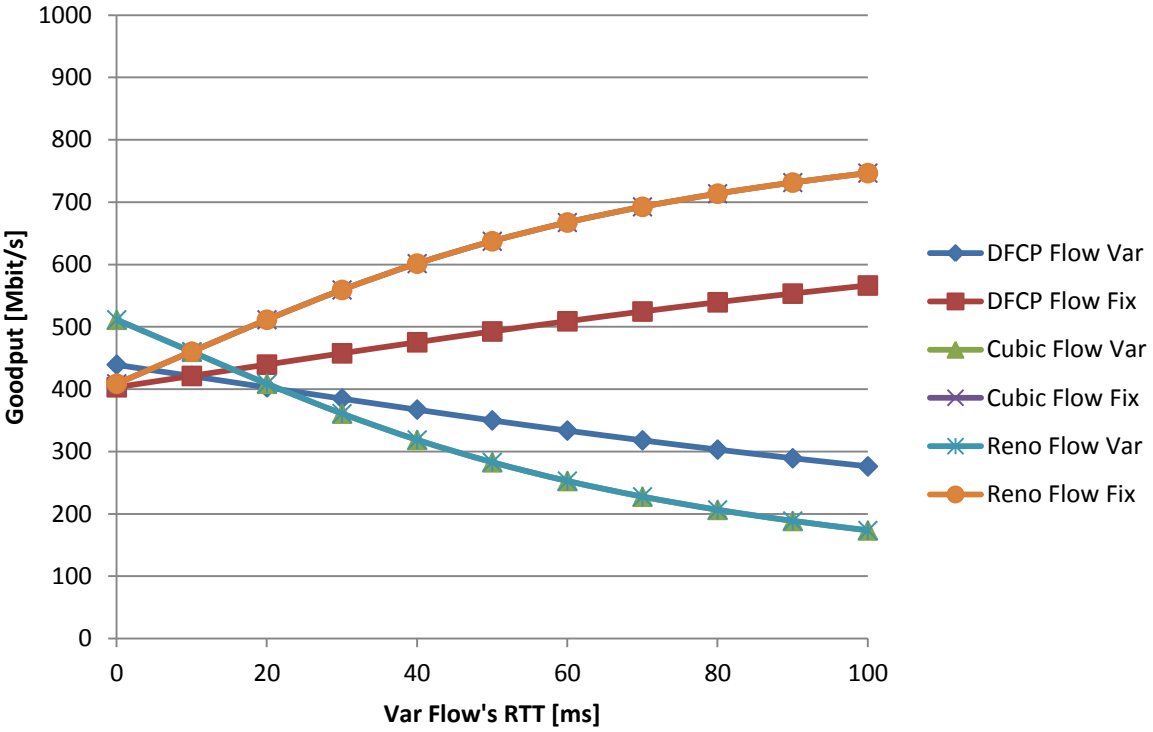
6.2 Két adatfolyamot tartalmazó szimulációk

6.2.1 Szimuláció Droptail várakozási sorral

A szimulációkhoz a következő jellemzőkkel rendelkező topológiát hoztam létre:

- Két folyamot szimuláló topológiát állítottam össze.
- A bottleneck link átviteli kapacitását 1000 Mbit/s sebességre állítottam.
- A linkeken alkalmazott várakozási sorok méretét 10 ezer csomagra állítottam.
- A két küldő felől csatlakozó linkek kapacitása egyenként 1000 Mbit/s.
- A bottleneck linken Droptail várakozási sor használatát állítottam be.
- Egyik linken sem alkalmaztam csomagvesztést.

A 26. ábrán láthatók az általam futtatott tesztek összesített eredményei, mind a három protokollra. A grafikonon a ”*Flow Fix*” jelöli a konstans 10ms RTT ágon csatlakozó folyamot, míg a ”*Flow Var*” jelöli a másik folyamot, amelynek a bottleneck-re csatlakozó ágán az RTT paramétert változtattam.



26. ábra

Jól látszik a protokollok azon tulajdonsága, hogy a növekvő késleltetésű folyamot kiszorítja az állandó, alacsonyabb késleltetésű vetélytársa. Az új protokoll is hasonlóan

viselkedik droptail sor mellett, azonban ez a hatás kevésbé erős. A TCP változatok teljesítménye közel azonos.

Általánosságban kijelenthető, hogy a hagyományos TCP protokollok azonos késleltetés mellett osztoznak igazságosan a kapacitáson, tehát ekkor fair-ek egymással szemben. Eltérő késleltetés mellett az alacsonyabb késleltetésű adatfolyam kerül kedvezőbb helyzetbe, és kiszorítja a magasabb késleltetéssel rendelkező vetélytársát.

6.2.2 Tesztkörnyezet determinisztikusságának meghatározása

A két adatfolyamot tartalmazó szimulációs méréseknél felmerült az a kérdés, hogy a tesztkörnyezet érzékeny-e a hálózati topológia egyes elemeinek megadási sorrendjére, valamint más objektumok létrehozásának sorrendjére. Ezért megismételtem a teljes droptail sorral elvégzett szimulációs sort egyenként a következő módosításokkal:

- Felcseréltem a küldő ágens párok becsatolását a hálózatba.
- Felcseréltem a küldő ágensek létrehozásának sorrendjét.
- Összevontam a két fogadó ágenst a bottleneck fogadó oldali végére, és elhagytam azon az oldalon a fogadó leágazásokat a topológiából.

Ezen módosítások közül egyik sem volt hatással a mérések eredményére. Ezek alapján kijelenthető, hogy az NS szimulációs környezet determinisztikusan ugyanazt az eredményt szolgáltatja, ha routing szempontból azonos hálózaton kerül lefuttatásra az azonos paraméterekkel megadott szimulációs eset.

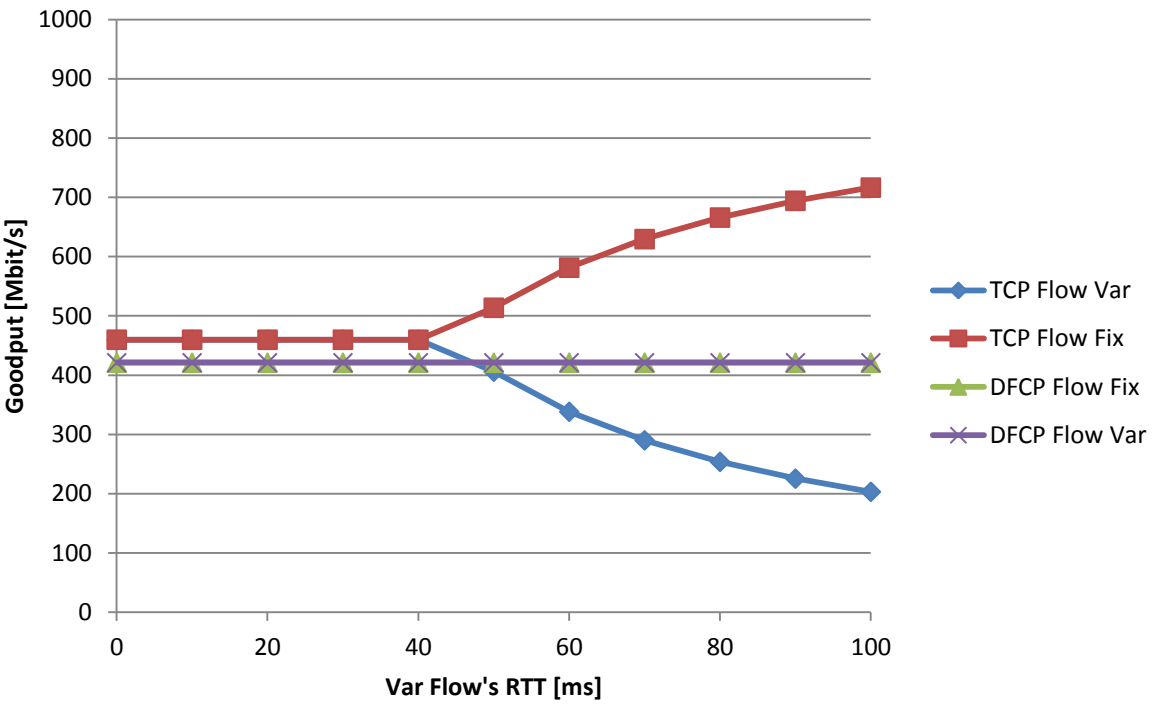
Ebből következik, hogy a hálózat megadásának bármely permutációjára ugyanazok az eredmények születnek.

6.2.3 Szimuláció DRR várakozási sorral

A szimulációkhoz a következő jellemzőkkel rendelkező topológiát hoztam létre:

- Két folyamot szimuláló topológiát állítottam össze.
- A bottleneck link átviteli kapacitását 1000 Mbit/s sebességre állítottam.
- A linkeken alkalmazott várakozási sorok méretét 10 ezer csomagra állítottam.
- A két küldő felől csatlakozó linkek kapacitása egyenként 1000 Mbit/s.
- A bottleneck linken DRR várakozási sor használatát állítottam be.
- Egyik linken sem alkalmaztam csomagvesztést.

A 28. ábrán láthatók az általam futtatott tesztek összesített eredményei, mind a három protokollra. A grafikonon a ”*Flow Fix*” jelöli a konstans 10ms RTT ágon csatlakozó folyamot, míg a ”*Flow Var*” jelöli a másik folyamot, amelynek a bottleneck-re csatlakozó ágán az RTT paramétert változtattam:



28. ábra

Az eredmények alapján a TCP változatok átviteli teljesítménye azonos, ezért a grafikonon közös jelöléssel ábrázoltam.

Látható, a TCP protokollok esetén az DRR sor alkalmazása ellensúlyozza a nem túl nagy késleltetésbeli eltérést. Magas RTT eltérés mellett azonban a droptail sor használata esetén tapasztalt jelenség figyelhető meg ebben az esetben is.

A DFCP protokoll DRR sor esetén is érzéketlen a késleltetésre a fairness tekintetében. Tehát két versengő adatfolyam magas késleltetésbeli jellemzőkkel is azonos mértékben, egyenlően használja a bottleneck szűk keresztmetszetű kapacitását.

A teljes DRR sorral futtatott esetsort megismételtem SFQ ütemező megadásával is. Az így kapott eredmények megegyeztek a fent láthatókkal.

7. Összefoglalás

Részletesen tanulmányoztam a TCP protokoll alverzióit, valamint a tanszéken fejlesztett torlódásszabályozás nélküli transzport protokollt (DFCP). Szimulációs környezetben vizsgáltam a TCP Reno, TCP Cubic, DFCP protokollok által szolgáltatott adatátviteli sebességet, egy- és kétfolyamos esetekre.

Az egyfolyamos esetekben a linken jelentkező késleltetés és csomagvesztés hatásait vizsgáltam. Az eredmények összesítése után kijelenthető, hogy a DFCP protokoll magasabb átviteli sebességet biztosít rossz tulajdonságokkal rendelkező átviteli közegekben, mint a TCP változatok. Továbbá a DFCP protokollt megfelelően paraméterezve alacsony sorhosszúság mellett is képes megfelelően működni, amely kedvező az optikai átviteli közegekre, ahol nincs lehetőség nagyon nagy kapacitású várakozási sorok kialakítására.

Kétfolyamos eseteken pedig azt vizsgáltam, hogy az eltérő késleltetéssel rendelkező adatfolyamok hogyan versengenek a szűk átviteli kapacitással rendelkező hálózati szakaszért. A szimulációs eredmények alapján kijelenthető, hogy a DFCP protokoll fairness tekintetében sokkal kedvezőbb, még az igen nagy késleltetési különbségekkel rendelkező adatfolyamok esetén is.

Összefoglalva a szimulációs vizsgálataim eredményeiből megállapítottam, hogy a DFCP protokoll jobb működési alternatívát biztosíthat a jövő hálózatainak kiszolgálására a jelenlegi TCP verziókhoz képest.

További vizsgálatok célja lehet a szimulációs esetek kibővítése a már említett kétfolyamos topológián úgy, hogy a versengő adatfolyamok eltérő időben kerülnek indításra. Így a szűk keresztmetszet sávszélességét már kihasználó adatfolyamot szorítana ki egy később indított másik adatfolyam. További kutatási lehetőség annak felderítése, hogy az egyes protokollok hogyan viselkednek az átviteli közeg paramétereinek változására, különös tekintettel az átviteli kapacitására.

A vizsgálatok további kibővítési pontja a kétfolyamos esetek bővítése az úgynevezett ”*parking lot*” topológiával.

Irodalomjegyzék

[1] H.Zimmermann, OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection, April, 1980

[2] V.Cerf, Y.Dalal, C.Sunshine, Specification of Internet Transmission Control Program, Dec, 1974

[3] A. Afanasyev, N. Tilley, P. Reiher, L. Kleinrock, “Host-to-Host Congestion Control for TCP”, IEEE Communications Surveys and Tutorials,vol. 12, no. 3, pp. 304–342, 2010

[4] S. Molnar, B. Sonkoly, T. A. Trinh, “A Comprehensive TCP Fairness Analysis in High Speed Networks”, Computer Communications, Elsevier, vol. 32, no. 13–14, pp. 1460–1484, 2009

[5] D. Clark, S. Shenker and A. Falk, ”GENI Research Plan”, Version 4.5 of April 23, 2007.

[6] S. Molnár, Z. Móczár, A. Temesváry, B. Sonkoly, S. Solymos, T. Csicsics, Data Transfer Paradigms for Future Networks: Fountain Coding or Congestion Control?, IFIP NETWORKING 2013, New York, USA, 22-24 May, 2013.

[7] Paul E.McKenny, Stochastic Fairness Queuing, Jan 24,1991

[8] M. Shreedar and G. Varghese, Efficient Fair Queuing Using Deficit Round-Robin, June, 1996

[9] Network Simulator 2
http://www.isi.edu/nsnam/ns/ (2013-10-20)

[10] Network Simulation Cradle
http://www.wand.net.nz/~stj2/nsc/ (2013-10-20)

[11] Sam Jansen, Network Simulation Cradle, The University of Waikato, 2008

[12] Kevin Fall, Kannan Varadhan, The NS Manual Nov 4, 2011

[13] ns-2.33 Documentation
http://www.cs.mun.ca/~yzchen/code/ns-2.33/class_s_f_q-members.html
(2013-10-20)

[14] ns-2.33 Documentation
http://www.cs.mun.ca/~yzchen/code/ns-2.33/class_d_r_r-members.html
(2013-10-20)

Függelék

Az alábbiakban látható a szimulációkhoz használt kiinduló szkript, melyet az egyes esetekhez a külön jelzetteknek megfelelően módosítottam. A szkript elvégzi a várakozási sor hosszúságának, az összesen átvitt adatmennyiségnek, az adatfolyam által elért *goodput* sebességnek és a torlódási ablakok méretének naplózását is. Továbbá összesített eredményeket tartalmazó naplót is készít, amelyben megtalálhatók a következő adatok: összesen átvitt hasznos adatmennyiségek, több jelzett intervallumra elért átlagos *goodput* sebességek és a várakozási sor átlagos hosszúsága.

```
#----myfile vars
set A 0
set B 0
set SentBytes 0
set ReceBytes 0
set Link_Bw 0
set Link_Loss 0
set Link_Qsize 0
#----queue telem full interval
set Qsum 0
set Qpts 0
set Qavg 0
#----queue telem 60s interval
set Qsum1 0
set Qpts1 0
set Qavg1 0

proc print-queue {interval fp} {
	global ns qmon agent1 Qsum Qpts Qsum1 Qpts1

	set now [$ns now]
	set Qnow [$qmon set pkts_]
	set Qsum [expr $Qsum+$Qnow]
	set Qpts [expr $Qpts+1]
#-------qavg for 15-60s
	if {$now >= 15.0 && $now <=60.0} {
	set Qsum1 [expr $Qsum1+$Qnow]
	set Qpts1 [expr $Qpts1+1]
	}
	puts $fp "[format %.3f $now] $Qnow"
	$ns after $interval "print-queue $interval $fp"
}
proc print-cwnd {interval fp} {
	global ns agent0

	set now [$ns now]
	puts $fp "[format %.3f $now] [$agent0 set cwnd_]"
	$ns after $interval "print-cwnd $interval $fp"
}
set lastGput 0
proc print-gput {interval fp intervalfp} {
	global ns agent1 lastGput diffp
	set now [$ns now]
	set recieved [$agent1 set bytes_]
	set bytesDbl [ns-int64todbl $recieved]
	set Kbytes [expr $bytesDbl / 1024.0]
	puts $fp "[format %.3f $now] [format %.3f $recieved]"
	set diff [expr (($Kbytes-$lastGput))/($interval*128.0)]
	puts $intervalfp "[format %.3f $now] [format %.3f $diff]"
		set lastGput $Kbytes
```

```
        $ns after $interval "print-gput $interval $fp $intervalfp"
}
set bw 1000
set ns [new Simulator]

# Common TCP parameters
Agent/TCP set window_ 2422
Agent/TCP set packetSize_ 1468
Agent/TCP set overhead_ 0.000008
Agent/TCP set max_ssthresh_ 100
Agent/TCP set maxburst_ 2

#Application/Traffic/CBR set packetSize_ 1468
Application/Traffic/CBR set packetSize_ 65536
#---------------------------------
#DFCP parameters
Agent/TCP/NSC set version_ 7
Agent/TCP/NSC set win_ 100
Agent/TCP/NSC set ack_off_ 0

#DFCP optimal resend 49
Agent/TCP/NSC set resend_ 55
Agent/TCP/NSC set code_off_ 1
Agent/TCP/NSC set decode_off_ 1
Agent/TCP/NSC set maxtokens_ 0
Agent/TCP/NSC set inctokens_ 507145 ;# $bw * 524.288 * 0.96730245
Agent/TCP/NSC set inttokens_ 1
Agent/TCP/NSC set bandwidth_ 1000

set n0 [$ns node]
set n1 [$ns node]
#RTT
$ns simplex-link $n0 $n1 1000Mb 0ms DropTail
$ns simplex-link $n1 $n0 1000Mb 0ms DropTail
$ns queue-limit $n0 $n1 10000;
$ns queue-limit $n1 $n0 10000;

set agent0 [new Agent/TCP/NSC/Linux2626]
set agent1 [new Agent/TCP/NSC/Linux2626]
$agent0 set fid_ 1
$agent1 set fid_ 1
$ns attach-agent $n0 $agent0
$ns attach-agent $n1 $agent1

#Congestion Control setting for TCP
#$agent0 sysctl net.ipv4.tcp_congestion_control reno
#$agent1 sysctl net.ipv4.tcp_congestion_control reno

$ns connect $agent0 $agent1
set app [new Application/Traffic/CBR]
$app set rate_ 1100Mb
$app attach-agent $agent0
#---------------------------------
#error module
set loss_module [new ErrorModel]
$loss_module set rate_ 0.01
#loss unit
$loss_module unit pkt
$loss_module drop-target [new Agent/Null]
#attach
#$ns link-lossmodel $loss_module $n0 $n1 ;# after queue
$ns lossmodel $loss_module $n0 $n1 ;#before queue
```

```
#--------------------------------
set qfp [open queue.out w]
set qmon [$ns monitor-queue $n0 $n1 ""]
print-queue 0.001 $qfp

set cwndfp [open cwnd.out w]
print-cwnd 0.001 $cwndfp

set gputfp [open gput.out w]
set diffp [open gputintervals.out w]
print-gput 0.1 $gputfp $diffp

set thputfp [open thput.out w]
set mf [open myfile.out w]

#--------------------------------
#flow monitor

set flowmon [$ns makeflowmon Fid]
set fLink [$ns link $n0 $n1]

$ns attach-fmon $fLink $flowmon
set fclass [$flowmon classifier]
set flowstats [$fclass lookup auto 0 0 1]
if {$flowstats != "" } {
#    set bytes [$flowstats set bdepartures_]
}
#-----------------------------
#Throughput tracing of a flow in a link
set lastKBytes 0
proc print-thput {fid fmon interval fp} {
        global lastKBytes ns
        set now [$ns now]
        set fcl [$fmon classifier]; # flow classifier
        set flow [$fcl lookup auto 0 0 $fid]
        if {$flow != "" } {
                set bytes [$flow set bdepartures_]
                set bytesDbl [ns-int64todbl $bytes]
                set Kbytes [expr $bytesDbl / 1024.0 ]
                set thruLastPeriod [ expr (( $Kbytes - $lastKBytes ) )/($interval*128.0)]
                set lastKBytes $Kbytes
                puts $fp "[format %.3f $now] [format %.3f $thruLastPeriod]"
        }

        $ns after $interval "print-thput $fid $fmon $interval $fp"
}
proc saveAgent {} {
        global ns agent0 agent1 mf SentBytes ReceBytes A B
        puts $mf "agent-vars-------------------------"
        set a [Agent/TCP/NSC set win_]
        set b [Agent/TCP/NSC set resend_]
        puts $mf "win_        : $a"
              puts $mf "resend_      : $b"
}
```

```
proc saveMS {} {
        global ns agent0 agent1 mf SentBytes ReceBytes A B
        set now [$ns now]
        set SentBytes [$agent0 set bytes_sent_]
        set ReceBytes [$agent1 set bytes_]
        set A $ReceBytes
        puts $mf "gput-bytes-------------------------"
        puts $mf "Sent     at     $now: $SentBytes"
              puts $mf "Recieved at     $now: $ReceBytes"
    }
    proc saveMF {} {
        global ns agent0 agent1 mf SentBytes ReceBytes A B
        set now [$ns now]
              set SentBytes [$agent0 set bytes_sent_]
              set ReceBytes [$agent1 set bytes_]
        set B $ReceBytes
        puts $mf "gput-byte-------------------------"
        puts $mf "Sent     at     $now: $SentBytes"
              puts $mf "Recieved at     $now: $ReceBytes"
    }

    $ns at 0.0 "$app start"
    $ns at 0.0 "print-thput 1 $flowmon 0.1 $thputfp"
    $ns at 0.0 "saveAgent"
    $ns at 15.0 "saveMS"
    $ns at 60.0 "saveMF"
    $ns at 60.0 "$app stop"
    $ns after 60.1 "finish; exit"

    proc finish {} {
        global ns agent0 agent1 SentBytes ReceBytes A B qfp cwndfp gputfp diffp thputfp mf Qpts Qsum Qavg Qpts1
Qsum1 Qavg1
        catch {
                set SentBytes [$agent0 set bytes_sent_]
                set ReceBytes [$agent1 set bytes_]
                puts $mf "gput-byte-------------------------"
                uts $mf "sent     at finish: $SentBytes"
                puts $mf "recieved at finish: $ReceBytes"
                puts $mf "gput-byte-------------------------"
                puts $mf "Bytes &60.0-&15.0 : [expr $B-$A]"
                puts $mf "speed-Mbps-------------------------"
                set Measure60 [expr (($B-$A))/(45.0*1024.0*128.0)]
                set Measuref [expr (($ReceBytes-$A))/60.0*1024.0*128.0)]
                set MeasureAll [expr ($ReceBytes)/(60.0*1024.0*128.0)]
                set MeasureAlli [expr int($MeasureAll)]
                set Measure60i [expr int($Measure60)]
                set Measurefi [expr int($Measuref)]
                puts $mf "Goodput  0.0 to 60.0: $MeasureAlli ($MeasureAll)"
                puts $mf "Goodput 15.0 to 60.0: $Measure60i ($Measure60)"
                puts $mf "Goodput 15.0->finish: $Measurefi ($Measuref)"
                puts $mf "Queue-telemetry--------------------"
                set Qavg [expr $Qsum/$Qpts]
                set Qavg1 [expr $Qsum1/$Qpts1]
                puts $mf "Average Queue Size 15.0 to 60.0  : $Qavg1"
                puts $mf "Average Queue Size  0.0 to finish: $Qavg"
                }
        flush $qfp
        lose $qfp
        flush $cwndfp
        lose $cwndfp
        flush $gputfp
        close $gputfp
        flush $diffp
```

```
        close $diffp
        flush $thputfp
        close $thputfp
        flush $mf
        lose $mf
}

$ns run
```