EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

INFORMÁCIÓS RENDSZEREK TANSZÉK

TORLÓDÁSVÉDELMI MÓDSZEREK ÉS TECHNIKÁK ELEMZÉSE

ÍRTA: TUSKA BALÁZS

(btuska@elte.hu)

SZAK: PROGRAMTERVEZŐ MATEMATIKUS

TÉMAVEZETŐ: VINCELLÉR ZOLTÁN

BUDAPEST, 2004.05.20

Köszönetnyilvánítás

Köszönetemet fejezem ki témavezetőmnek, Vincellér Zoltán tanár úrnak, aki széleskörű hálózati ismereteket nyújtva lehetővé tette a diplomamunka megszületését, és értékes tanácsaival nagyban hozzájárult munkám elkészüléséhez.

Tartalomjegyzék

1. Bevezetés

1.1. Mi az a torlódás?

Talán a címben szereplő kifejezés megtévesztő lehet, mert torlódás sok helyen előfordulhat akár a hétköznapi életben is. Ebben a dolgozatban a számítógép-hálózatok esetében és környezetében vizsgálódunk. Mindennapos lehet ez a jelenség általános használat esetén is, de előfordulhat szándékos rongálás, mely esetben a hálózat teljesen lelassul, megbénul. A célunk az, hogy ezeket az eseteket megelőzzük, vagy a probléma fennállása esetén megszüntessük azt. Erre rengeteg lehetőség van, melyek több-kevesebb sikerrel alkalmazhatóak. Mi megpróbáltuk azokat az eszközöket felsorakoztatni, melyeket egy átlagos felhasználó is üzembe tud helyezni ezen leírás elolvasása után. Mint látni fogjuk az alábbi módszerek nem igényelnek semmilyen speciális eszközt, mégis szép számú lehetőség közül lehet választani. Egyes módszerek nem alkalmazhatóak mindenhol eredményesen, de erre külön ki fogunk térni, hogy mikor mit érdemes alkalmazni.
Dolgozatomban közelebbről vizsgálom meg ezt a problémakört. Megnézem a jelenleg megtalálható módszereket, ezek hatékonyságát, valamint azt, hogy hol használhatóak ezek eredményesen. Célom, hogy átfogó képet adjak erről a témáról úgy, hogy azok is felhasználhassák ezeket az ismereteket, abban a környezetben, ahol hasonló jelenséget tapasztaltak, de nem tudják, mit tehetnének. Különböző körülmények eltérő megoldásokat követelnek, de a végeredmény ugyanaz, mégpedig a maximum kihozása a legkevesebb anyagi befektetéssel. Bár a mai nagy sebességű hálózatok mellett ez nem tűnhet fontosnak, mert a forrásokat kimeríthetetlennek hisszük, de torlódás védelem és megelőzés nélkül nem sokáig lennének használhatóak a jelenlegi kapacitások sem.

1.1.1. A torlódásról általában

Nos eddig még nem igazán határoztuk meg mi is az a torlódás valójában. Ezt egyenlőre még nem is szeretném megtenni, inkább bemutatok egy hétköznapi példát.
Nyilván mindenkivel előfordult az az eset miközben oldalakat nézegetett, esetleg nagyobb

file-t töltött le az Internetről, hogy az oldal nem jött le olyan sebességgel, vagy hosszú ideig kellett várni míg az oldal egyáltalán válaszolt. Netalántán még ssh-zni is próbáltunk, de a túloldali gép lassan reagált a mi üzeneteinkre. Ezek a hálózati jelenségek jó eséllyel a torlódásoknak köszönhetőek. A hálózat egy olyan láncnak tekinthető, amelyen a különböző alhálózatok és az egyes hálózati eszközök alkotják a láncszemeket. És mint minden lánc esetében itt is a leggyengébb láncszem fogja meghatározni a rendszerünk maximális teljesítőképességét. Tehát hiába van nekünk 100 Mbites belső hálózatunk, ha az egész egy ADSL routerrel csatlakozik az internetre egy ADSL kapcsolaton keresztül, akkor a letöltési sebesség nem fogja meghaladni a 512 Kbitet. De ami ennél is fontosabb az a feltöltési sebesség, mert itt van a mi oldalunkról a torlódás egyik forrása. Mégpedig a kapcsolat a lassú hálózat és a gyors belső hálózat között. Legtöbbször ezen a határon egy modem helyezkedik el, ami hardveresen nagy várakozási sorral lett felszerelve, ami az interaktivitásnak nem előnyös, és semmilyen algoritmust nem tartalmaz a túlterhelés megelőzésére vagy kiküszöbölésére ezen egyetlen soron kívül. Azért rossz ez az egyetlen sor, mert ha kevés a forgalom, akkor nem történik semmi, de ha a kimenő sebesség meghaladja a külső hálózat sebességének a határát, akkor a modem nem tudja elküldeni a csomagokat, és kénytelen őket berakni a várakozási sorába. Ha a sor betelik a hardvereszköz eldobja a további csomagokat. Mindezt úgy teszi, hogy az egyes forgalmak között nem tesz különbséget. Így az agresszívabb forgalomnak több esélye van kijutni, és a ritka de gyors reakciót kívánó forgalom, ha be is kerül a sorba, akkor is ki kell várnia míg sorrakerül, ami a hosszú sor miatt nem rövid idő. Ezzel megöli az interaktivitást.

Ebből a példából is látszik, hogy többféle forgalmat lehet megkülönböztetni. Azon kívül, hogy létezik kimenő és bejövő forgalom, a forgalmaknak számtalan típusa van. Lehetne őket osztályozni a csomag mérete, célja, forrása, protokollja alapján. Mivel a lassabb hálózat nem képes fogadni a gyorsabb hálózatról érkező forgalmat ezért tenni kell valamit ennek a kettőnek a határán, azaz a torlódást még a határ előtt kell enyhíteni, hogy a lassabb hálózat a gyorsabbtól már megfelelő sebességgel kapja a csomagokat, azaz a lassabb hálózat maximális teljesítőképességénél valamivel alacsonyabban.

Legtöbbször arról sem szabad megfeledkezni, hogy egy hálózatot nem csak egyetlen ember használ, hanem több. Előfordulhat, hogy valaki nem olyan célból használja, ami mások szempontjából előnyös, és „elnyom” másokat. Ezért célszerű olyan módszert választani ilyen környezetben, ahol a különböző felhasználóktól érkező forgalom között egyenlőséget tudunk biztosítani.

1.1.2. A torlódás technikai megközelítésből

A torlódás egy olyan jelenség ami akkor jön létre, amikor túl sok csomag van jelen a hálózat egy részében, ezzel akadályozva a csomagok továbbhaladását. Így jelentős lassulást idéz elő, sőt csomagok elvesztéséhez vezethet, amik újraküldése tovább terheli a hálózatot.
Ezeknek az okainak a mélyebb magyarázatához szükség van egy részletesebb betekintésre a számítógépes hálózatok világában.

1.1.2.1. Hálózati rétegek

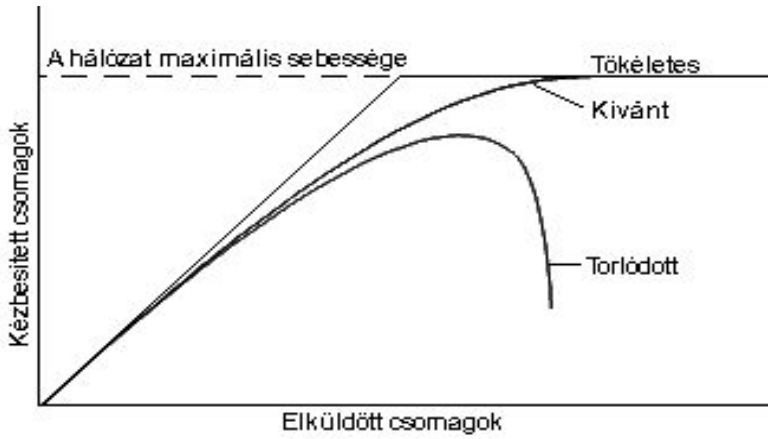
A számítógépes hálózat rétegekből épül fel, melyek közül két hivatkozási modell a legelterjedtebb: az OSI és a TCP/IP. Ezen hivatkozási modellek tárgyalása nem része a dolgozatnak, de a témát érintő részekről beszélünk, és a könnyebb érthetőség miatt a többiről is nagyvonalakban.
Először az OSI modellről ejtsünk néhány szót. Az OSI (Open System Interconnection) hét rétegből áll: alkalmazási, megjelenítési, viszony, szállítási, hálózati, adatkapcsolati, fizikai réteg. Ha találóan akarom megfogalmazni, akkor: a felhasználó az alkalmazási réteggel, a programozó a szállítási réteggel, a rendszergazda a hálózati réteggel a hardver-gyártó az adatkapcsolati és fizikai réteggel találkozik. Az ahol a torlódásokkal leginkább foglalkoznak, az a hálózati réteg, de előfordul a szállítási réteg protokolljaiban is erre vonatkozó algoritmus.
A hálózati réteg (network layer) az alhálózatok működését irányítja. Ez a réteg felel a csomagok eljuttatásáért a forrástól a célig egy bizonyos útvonalon, sőt itt dől el a megfelelő út meghatározása. Itt találkozunk alhálózatokkal és a közöttük levő gatewayekkel, routerekkel. Tehát itt nem csak két gép összekapcsolásáról van szó, mint szállítási réteg esetén, hanem több különböző tulajdonságú hálózati eszköz láncolatáról. Éppen ezek a kapcsolódások lesznek a torlódások kialakulásának okai, mivel ezek az eszközök ritkán vannak egymáshoz igazítva, hangolva, lassabb illetve gyorsabb eszközök követik egymást. Itt nem lesz garantálva az egyes csomagok biztos célbajutása.
A szállítási réteg (transport layer) feladata az, hogy felsőbb réteg adatait feldarabolja, majd biztosan eljuttassa azt a célig, majd ott helyes sorrendben összerakja (van ettől eltérő

protokoll is). A manapság legnépszerűbb protokoll a TCP is ebbe az osztályba tartozik. A TCP négy módszert is kínál a torlódások kiküszöbölésére (ls. később). Ezentúl ez a réteg rendelkezik egy olyan módszerrel is, amely a lassabb hosztok elárasztását próbálja megakadályozni a gyorsabb hosztokról. Ez a mechanizmus a forgalomszabályozás (flow control). A szállítási réteg processz-processz közötti tulajdonságából adódóan nem tud kitérni az alhálózatok specifikus tulajdonságain alapuló szabályozásra, tehát ez koránt sem egyezik meg a hálózati rétegnél található szabályozással illetve politikával.
A másik hivatkozási modell, a TCP/IP már leegyszerűsítettebben csak 4 réteket tart nyilván: alkalmazási, szállítási, internet és hoszt és hálózat közötti kommunikációs rétegek. (RFC793) Innen a torlódások szempontjából az internet illetve a szállítási réteg a legfontosabb. Ezen rétegek egyikében sem jelenik meg a különböző hálózatok kapcsolata, már az internet rétegben is cél és forrásról beszélünk, ezeket viszont mindegy, hogy a csomag útján hol nézzük, mert mindig ugyanaz lesz. Majd látni fogjuk, hogy ezt hol használjuk ki az egyes módszerek esetén.
Az internet réteg meghatároz egy egységes protokollt, amit internet protokollnak (IP) hívnak. Ennek a réteknek az elsődleges célja a csomagok kézbesítése és a torlódások elkerülése. Ebből a szempontból hasonlít az OSI modell hálózati rétegéhez (úgy gondolom a hasonlóságok itt ki is merülnek). Én igazából az OSI rétegek közül nem igazan tudok megfeleltetni semmit, valahol a szállítási és hálózati réteg között, de a szállításihoz közelebb (tekintve a pont-pont tulajdonságot). Az IP protokollhoz (RFC791) hozzátartozik a jól definiált IP fejléc is (ls. jelmagyarázat). A szűrések során igen fontos szerepet játszik, mert ez az első fejléc-héj, ami alapján a csomagokat osztályozni lehet.
A következő IP alatti fejléc már a szállítási réteghez kapcsolódik. Ennek elemzése elsősorban a részletesebb ellenőrzés esetén szükséges. A két jellemző protokoll a TCP és az UDP. Ezek fejléce a jelmagyarázatnál található, ezeknek a szerepe fontos, amit később a szűrésnél látni fogunk. Rengeteg kis apró részletre lehet nézni az illeszkedést: csomag méret, kiindulási-, cél tulajdonságok, kezdet, vég illetve ellenőrző csomag tulajdonság.

1.1.2.2. A torlódás

Tehát ha a hálózat egy részében túl sok csomag van jelen, akkor az torlódáshoz vezet. Ez történhet úgy, hogy az egy alhálózaton levő gépek egyszerre hirtelen sok csomagot

bocsátanak a hálózatba, és ezek a csomagok egy kimenő vonalon akarnak távozni. Vagy az egyik közbenső router processzora nem megfelő gyorsasággal dolgozza fel a csomagokat stb. A következő ábra szemlélteti mi történik ilyen helyzetben:



Látható, hogy amíg a küldött csomagok, a szállítási kapacitáson belül vannak, az elküldött és a kézbesített csomagok egyenesen arányosan viszonyulnak egymáshoz. De ha a hálózaton található csomagok száma meghaladja a maximális kapacitást a hálózati eszközök már nem képesek feldolgozni az összes csomagot,így azok elvesznek. Ha a forgalom jelentősen meghaladja a határt, akkor már az eddigi kézbesítések is leállhatnak, és a hálózat teljesen összeomlik.

Elsődleges megoldásként jöhet az a triviális ötlet, hogy a torlódás helyére egy sort rakjunk, hogy a pillanatnyilag elküldeni nem tudott csomagokat belerakjuk, majd szabad kapacitás esetén azokat is elküldjük. De előbb utóbb a sor megtelik, ha túl sok a csomag, és az újaknak nem jut hely így azok megint csak eldobódnak. Ha viszont túl hosszú a sor, akkor esetleg minden csomagnak lesz helye, de miközben várakoznak, az idejük is lejárhat, így a magasabb szintű protokoll újra fogja küldeni, ami tovább ronthatja a helyzetet.

Torlódás védelem módszereivel a helyzeten sokat lehet javítani különösebb ráfordítás nélkül, de ez sem képes csodákra, a fizikai korlátokat nem lehet áthágni vele, tudni kell hol a határ, és nem kell többre számítani, mint amire képes. Az is fontos, hogy nem megfelelően választott algoritmus sok esetben ronthat a helyzeten.

1.1.2.3. A torlódásvédelem alapelvei

Itt kétféle megközelítés a jellemző. Az egyik eleve úgy próbál építkezni, hogy torlódás ne alakulhasson ki. Ez a rendszer elindítása után statikus, menet közben nem tud változtatni. A másik megközelítés szerint a forgalom torlódási szempontok szabályai szerint szabadon engedélyezett, majd torlódás esetén felderíti annak helyét, és ott vagy előtte megpróbálja feloldani azt. Ez a módszer erősen hagyatkozik arra, hogy az egyes hálózati elemek között jó a kommunikáció, és ha valaki kér valamit, azt a másik be is tartja. A valóság valahol a kettő egyvelegeként mutatkozik.

1.2. A TCP torlódásvédelmi algoritmusai

Mint korábban is említettem a TCP is kínál négyféle lehetőséget a torlódások elkerülésére. A Tanenbaum könyv úgy hangsúlyozza, hogy a torlódás megelőzésében itt történnek meg a fő lépések, és ez az elsődleges hely a megelőzésre. Mellékesen megjegyzi, hogy a hálózati rétegben is vannak lehetőségek. De ez éppen ellenkezőleg van, az okokat mindjárt elmagyarázom.

Az Interneten nem csak a TCP van jelen, hanem a programok számtalan protokollt használhatnak, így nem lehet csak ennek az egy protokollnak a védelmére hagyatkozni, mikor az UDP vagy ICMP stb. protokollok nem kínálnak erre lehetőséget. Persze a legelterjedtebb a TCP, de akkor sem az egyetlen. Számos program amellett, hogy TCP-re épül mégis meglehetősen „agresszív” viselkedést mutat. Azaz bár a TCP fejléc komponenseket használ, de az egyéb szabályokat figyelmen kívül hagyja. Így a hálózati rendszergazdák joggal nem is hagyatkoznak eme algoritmusokra, mert ezt nem ők felügyelik. A TCP szintjén már nem is avatkozhatnak be a szabályozásban, ezért egy alacsonyabb szint szükséges, amit még szoftveresen lehet szabályozni, ez innentől fogva egyértelműen csak a hálózati réteg lehet.

De most nézzük a TCP algoritmusait. Ezek az algoritmusok nem elkülönülve vannak, mint a hálózati rétegnél, hanem szorosan összekapcsolódnak, egymásra épülnek. A modell a rendszert három fő részre bontja: a küldő, a fogadó és a közöttük lévő kapcsolat. Az egész azon alapul, hogy a gyors küldő csak olyan mértékben tudjon küldeni csomagokat, hogy az a hálózaton is átmenjen és a fogadó is tudja fogadni azt. Úgy tudja vizsgálni a teljesítés

határainak elértét, hogy nézi, mikor kezdenek csomagok elveszni. Ezt hívják torlódás detektálásnak. Manapság a megbízható hálózatok idején legtöbbször, ha csomageldobás történik, akkor az torlódás következménye. Bár az „új” wireless technikák is hordoznak önmagukban ilyen veszélyeket.

1.2.1. Alapfogalmak

Definiáljunk néhány fogalmat melyek a fejezet továbbtárgyalása során fontos lehet.

- **Szegmens:** ez egy akármilyen TCP/IP vagy nyugta csomag (vagy mindkettő).
- **A küldő maximális szegmensmérete (SMSS):** ez a legnagyobb szegmens mérete, amit a küldő el tud küldeni. Ez az érték a hálózat MTU (maximum transmission unit) értékén alapul. Ez a méret nem tartalmazza a TCP/IP fejlécet és az opciókat sem.
- **A fogadó maximális szegmensmérete (RMSS):** a legnagyobb szegmens mérete, amit a fogadó még fogadni képes. Ezt a kapcsolat felépítése során a fogadó határozza meg és küldi el a küldőnek. Az MSS opcióban van beállítva, ha nem akkor ez 536 byte. Ez sem tartalmazza a fejlécet és opciókat.
- **Teljes méretű szegmens:** a lehetséges legnagyobb számú byte-ot tartalmazó szegmens, ami még megengedett.
- **Fogadó ablak (rwnd):** ez a mostanában használt ablak mérete.
- **Torlódási ablak (cwnd):** ez egy TCP állapot változó, ami a TCP által küldhető adatmennyiséget korlátozza le egy érték alá. A szabály a következő: bármely időpillanatban a TCP nem küldhet olyan csomagot, melynek a sorszáma nagyobb a legnagyobb nyugta sorszámának és a cwnd és rwnd minimumának az összegénél.
- **Kezdeti ablak (IW):** a kezdeti érték a küldő torlódási ablakára a háromutas kézfogás után.
- **Veszteség ablak (LW):** a torlódási ablak mérete abban a pillanatban, amikor a TCP érzékelte, hogy csomagvesztés történt az ismétlési újraidőzítő (retransmission timer) segítségével
- **újrakezdési ablak (RW):** ez a torlódási ablak mérete az újraküldés után egy bizonyos idő elteltével (bővebben később).
- **Repülő méret:** az az adatmennyiség amire még nem érkezett nyugta.

Most nézzük a már sokat emlegetett négy lehetőséget: lassú kezdés (slow start), torlódás elkerülés (congestion avoidance), gyors újraküldés (fast retransmit) és gyors helyre állítás (fast recovery). Egyes esetekben a TCP nem is használja ki még az algoritmusok által megengedetteket sem. De azokat átlépnie mindenféleképpen tilos lenne.
(RFC 2001, RFC 2581)

1.2.2. A lassú kezdés és a torlódás elkerülés

Ez a két algoritmus azt az adatmennyiséget szabályozza, hogy a küldő a hálózatba mennyi adatot küldhet. Az algoritmusok implementálásához két változóra lesz szükség a TCP állapotainak jellemzéséhez. A torlódási ablak (cwnd) a küldő oldali maximálisan küldendő adatmennyiség behatárolásához, és a fogadó ablak (rwnd) a fogadó oldali korlát a küldendő adatmennyiségre. Ennek a kettőnek a minimuma határozza meg a ténylegesen küldendő adatot.
Egy másik állapot változó a lassú kezdés határértéke (ssthresh). Ezt a változót az adatátvitel ellenőrzésére használja majd a két algoritmus.
Az adatforgalom kezdetén a hálózat állapotáról nincsenek információink, ezért a TCP lassan próbálja a hálózatot, hogy meghatározza annak kapacitását, és nem kergeti azt toródásba a hirtelen küldött nagy adatmennyiségekkel. A lassú kezdés algoritmust fogja ebből a célból használni az átvitel elején, vagy egy az ismétlési időzítő által detektált adatvesztés után.
Az IW, a kezdeti torlódási ablak, értéke kisebb vagy egyenlő mint 2*SMSS érték byte-ban, és semmiféleképpen sem nagyobb két szegmensnél. Néhány nem szabványos TCP kiterjesztés megengedi, hogy az IW érték nagyobb legyen, mint ez. Például:

IW = min (4*SMSS , max (2*SMSS , 4380 byte))

Az ssthresh kezdeti értéke néha túlzóan nagy lehet (például néhány megvalósításban a hirdetett ablak mérete), de ez később csökkenni fog a torlódás hatására. A lassú kezdés algoritmus akkor használatos, amikor cwnd értéke kisebb ssthresh-nél. Míg a torlódás elkerülés a cwnd nagyobb ssthresh esetben. Egyenlőség esetén bármelyik használható lehet.
A lassú kezdés alatt a TCP a cwnd éréket növeli legfeljebb SMSS byte-tal minden egyes ACK fogadása esetén (nyugta). Akkor fog leállni, ha a cwnd elérte (túlhaladt) az ssthresh értéken, vagy amikor torlódást észlel.
A torlódás elkerülés alatt a cwnd egy teljes méretű szegmens per körülfordulási idővel (rtt –

round-trip time) nő. Ez addig folytatódik, amíg a torlódás nem áll fenn. A következő képlet igen népszerű a cwnd új értékének a meghatározására: cwnd += SMSS*SMSS/cwnd.
Ez a növelés minden bejövő nem-duplikált nyugta esetén megtörténik. Ez a képlet elfogadható közelítést ad arra, ami a cwnd növeléséhez kell. (Ez a képlet agresszívabb, mint egy szegmens per RTT szerinti nyugtázás, de kevésbé agresszív, mint minden egyes csomag utáni nyugtázás.)
Egyes régebbi implementációkban a jobb oldalon még található egy plusz konstans, de ez helytelen, és rosszabb hatásfokot eredményez.
Egy másik elfogadott útja a cwnd növelésének torlódás elkerülés alatt az, hogy egy újabb állapot változó bevezetésével figyeljük a nyugtázott byte-ok számát. Amikor ez a szám eléri a cwnd-t, akkor a cwnt-t megnöveljük egy legfeljebb SMSS-nyi byte-tal. De ez a növelés sem haladhatja meg az egy teljes-méretű szegmens per RTT értéket.
Amikor a TCP torlódást detektált az ismétlési időzítővel, akkor az ssthresh értéket be kell állítani egy az alábbinál nem nagyobb értékre: ssthresh = max (repülő méret / 2 , 2*SMSS) (3), ahol a repülő méret, a hálózatban kint lévő adat mennyiség. Továbbá időtúllépés esetén a cwnd beállított értéke, nem haladhatja meg az elveszett ablak méretét (ami egy teljes méretű szegmens). Ezért az újraküldés esetén a TCP küldő a lassú kezdés algoritmus alatt használt növelést használja a cwnd-re a teljes méretű szegmenstől az ssthresh új értékéig.

1.2.3. A gyors újraküldés és gyors helyreállítás

A TCP fogadó egy azonnali duplikált nyugtát küld, amikor egy rendellenes szegmens érkezik. A célja ennek a nyugtának, hogy tájékoztassa a küldőt arról, hogy egy rendellenes szegmenst fogadott, aminek a sorszáma nem jó. A küldő szemszögéből ez azt jelenti, hogy a hálózatban történt valami hiba, ami ezt a duplikált nyugtát okozta. Ez a hiba származhat eldobott szegmenstől, ebben az esetben, az eldobott szegmens utáni további szegmensek mindegyike duplikált nyugtát küld vissza. A második ok az lehet, hogy a hálózatban a csomagok sorrendje összekeveredett (gyakori eset). Végül keletkezhetnek duplikált nyugták úgy is, hogy a hálózat készít egy másolatot vagy az adatcsomagról vagy a nyugtáról. A TCP fogadó azonnali nyugtát küld abban az esetben, amikor a bejövő szegmensek megtöltötték az üres sorokat. Ez már bővebb információval szolgál a küldő számára, így eredményesebben tudja alkalmazni az alábbi algoritmusokat.

A TCP küldő a gyors újraküldés algoritmust használja a veszteségek detektálására és javítására a duplikált nyugtákon alapulva. Ehhez három érkező duplikált nyugtát használ (4 azonosat közbenső csomagok nélkül), ezzel az azonossággal határozva meg az elveszett szegmenst. Ha megérkezett ez a három nyugta, akkor az ismétlését a hiányzó szegmensnek, az időzítő lejárta nélkül is megkezdi.

Miután a gyors újraküldés algoritmus elküldi a hiányzónak vélt szegmenst, a gyors helyreállító algoritmus fogja felügyelni az új adat átvitelét, amíg egy nem duplikált nyugta nem érkezik. Az ok, hogy miért nem a lassú kezdés algoritmust használja az az, hogy a duplikálás nem csak csomagvesztésre utalhat, mert a csomag elhagyhatta a hálózatot. Persze a probléma tartós fennállása esetén ez már nem lesz igaz. Másképpen a fogadó csak akkor generál duplikált nyugtát ha hozzá szegmens érkezett, azaz ez a szegmens elhagyta a hálózatot és bekerült a fogadó pufferébe, és már nem a hálózat erőforrásait fogyasztja. De mivel az ACK „óra” megmaradt, ezért a TCP folytathatja az újabb szegmensek küldését.

Ezen algoritmusok leggyakrabban a következő megvalósítás szerint történnek.

1. Amikor az első duplikált nyugta-hármas megérkezik, akkor a ssthresh értékét a (3)-as egyenletben meghatározott értéknél nem nagyobbra állítja.
2. Újra küldi az elveszett szegmenst, és a cwnd-t beállítja: cwnd = ssthresh + 3*SMSS. Ez mesterségesen „felpuffasztja” a torlódási ablakot annyival, amennyivel a hálózatba kiküldött szegmensek száma (három) és amennyit a fogadó elraktározott a pufferében.
3. Minden további duplikált nyugta esetén a cwnd értéke SMSS-nyivel megnövelődik. Ez a további kiküldött szegmensek miatt szükséges.
4. Egy szegmens küldése, abban az esetben, amikor azt a cwnd és a fogadó ablaka is lehetővé teszi.
5. Amikor megérkezik a következő nyugta, mely az új szegmenshez tartozik, akkor a cwnd értékét az ssthresh-re állítjuk be. Ez „lefogyasztja” az ablakot. Ezután újra az 1-es lépés jön.

Ez a megoldás sok veszteség esetén nem túl hatásos.

Ezután lépjünk egy réteggel lejjebb a hálózati réteghez. A routolás szintjén is valamennyire képesek lehetünk elkerülni, feloldani a torlódásokat. Hiszen egy torlódásos kapcsolat esetén a többi kapcsolatnak, más útvonalat választunk arra, amerre a vonalak nincsenek leterhelve. De a routing feladata elsősorban nem a torlódások kivédése. Nézzük ennek a rétegnek a további lehetőségeit.

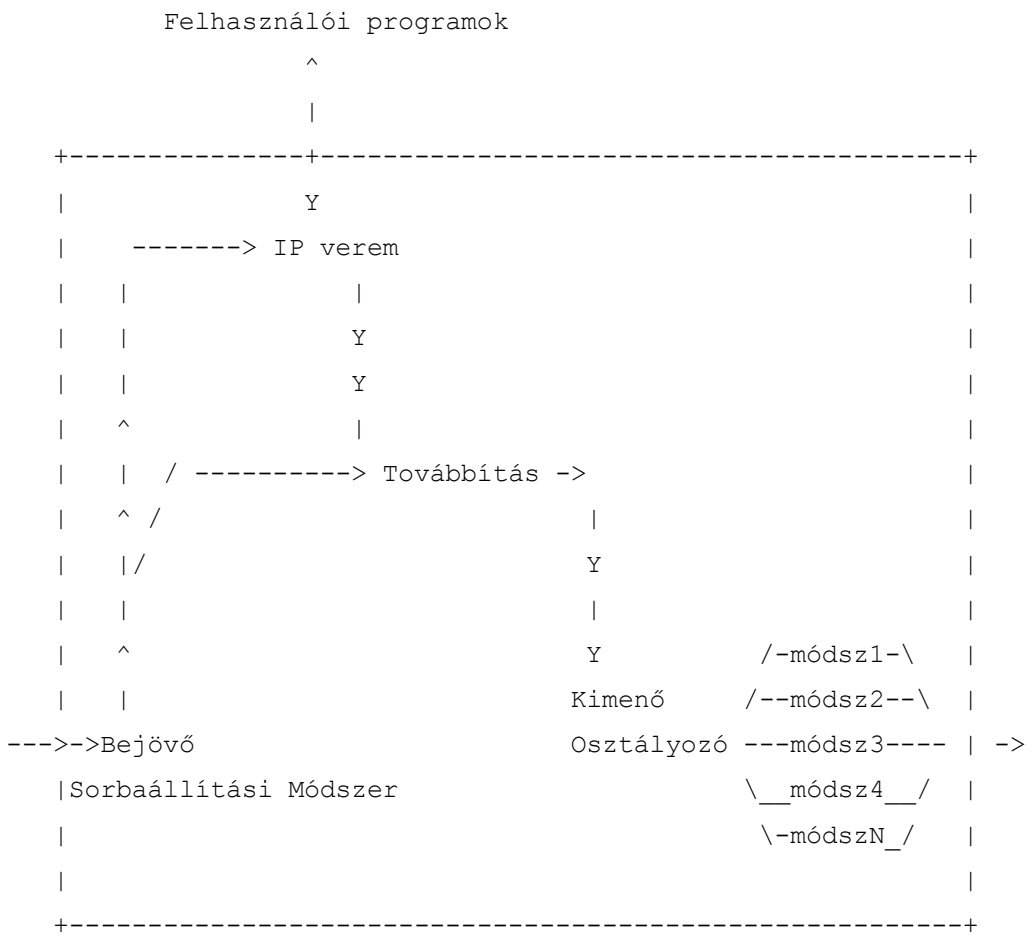
2. Sorbaállítási módszerek (Queueing Disciplines)

A sorokkal meghatározzuk az adatok küldési módját. De mi csak azokat a forgalmakat tudjuk alakítani melyek rajtunk keresztül mennek. Közvetlenül nem tudjuk szabályozni azt, hogy milyen csomagok érkezzenek hozzánk. Az Interneten szélesebb körben a TCP/IP van elterjedve, ami egy két képességében segít nekünk. Egyből nem tudja megállapítani a hálózat kapacitását, de a lassú kezdéssel fel tudja térképezni azt.

A sorbaállítási módszerekkel meg lehet akadályozni azt, hogy egy adott alhálózatban levő gép domináns szerephez jusson a teljes hálózati sávszélesség lefoglalásában. Ekkor a „belső" hálózati interfészen kell szabályozni, hogy ne legyen túl sok adat küldve egyes gépek felé.

De ha, teszem azt egy router a szűk keresztmetszet, akkor az is képes lekezelni a problémát, de esetleg nem úgy, ahogyan mi szeretnénk. Ezért arra törekszünk, hogy saját sort alakítsunk ki saját szabályrendszerrel, amit teljesen mi irányítunk. Ehhez azt kell elérni, hogy a mi ellenőrzésünk alatt legyen a leggyengébb láncszeme az adott láncnak.

Ha tekintjük a sorbaállítási módszereket, akkor ez az alapvető felépítésük:



A nagy kocka a kernelt jelenti, a legbaloldalibb nyíl a hálózatról a gépbe bejövő forgalom, amit a bejövő sorbaállítási módszer (ingress qdisc) fogad. Ez dönt a szűrője alapján, hogy a csomag eldobásra kerül-e (politika). Ez egy nagyon korai stádium, még mielőtt a csomag túl sokat látott volna a kernelből. Azért jó még itt eldobni a csomagot, mert nem igényel sok processzor erőforrást.
Ha a csomag továbbjutott ezen, akkor lehet, hogy hozzá van rendelve egy program. Ebben az esetben a csomag az IP verembe kerül, hogy onnan a felhasználói területre jusson feldolgozásra. De a csomag program érintése nélkül is továbbításra kerülhet, egyből a kijárat fele. A felhasználói térben levő programok is küldhetnek a kijárat fele csomagokat.
Itt megvizsgálják, majd sorbaállítják a csomagot valamelyik módszer sorába. Beállítatlan alapesetben csak egy kimenő sorbaállítási módszer van, ez pedig a pfifo_fast, ami mindig fogadja a csomagokat. Ez a rész a sorbaállítás (enqueueing). A csomag addig van itt, amíg a kernel át nem engedi a hálózati eszközön, ez a sorból kivétel (dequeueing).
Ebben az esetben éppen egy hálózati eszköz van, de ebből lehet több is. Ezek közül mindegyik rendelkezik saját bejövő és kimenő kezelővel.

2.1. Egyszerű, osztálytalan sorbaállítási módszerek (Simple, classless Queueing Disciplines)

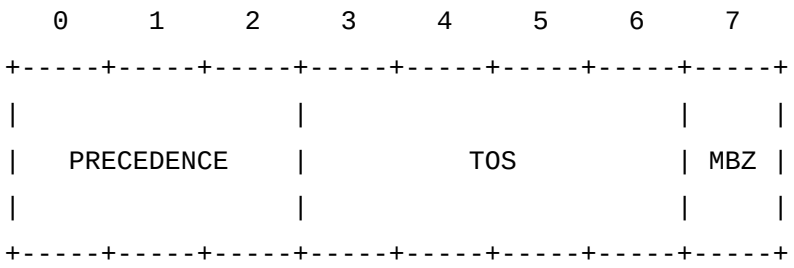
A sorbaállítási módszerekkel megváltoztatjuk az adatküldés módját, azaz nem ész nélkül küldünk csomagokat. Az osztálytalan sorok csak elfogadják, újraütemezik, várakoztatják illetve eldobják a csomagokat.
A teljes interfészen szabályozzák a forgalmat alosztályok nélkül. A továbbiaknak ez az alapja, így megértése nélkülözhetetlen a későbbi módszerek tárgyalása során.
A legszélesebb körben használt módszer a pfifo_fast, talán azért is, mert mikor létrehozunk egy definiálatlan módszert, akkor az pfifo_fast lesz.
Minden ilyen sornak van hossza és szélessége (azaz áteresztő képessége)

2.1.1. A pfifo_fast

Ez a sor, mint ahogyan a neve is mutatja first in first out (első be első ki), azaz a csomagok fogadásának nincsen különleges módja. A megszokott FIFO-tól azért mutat néhány különbséget, ennek a sornak 3 kötege (band) van, minden kötegen egy-egy különálló FIFO sornak felel meg. Amíg a 0. sorban várakozik csomag, addig az 1-es sorban levő csomagok nem kerülnek feldolgozásra, és amíg az 1-esben van csomag, addig a 2-eshez nem nyúl hozzá.
A csomagoknak van egy Type of Service flagje, mely alapján a kernel be tudja sorolni őket (pl.: az 'minimum delay' (minimális várakozású) csomagokat a 0-ás kötegbe helyezi.
De ennek ellenére ne tévesszük össze az osztályos PRIO-val. Bár hasonló tulajdonságokkal bír, de fontos, hogy a pfifo_fast nem osztályos, tehát a kötegeihez nem adhatók további módszerek.

2.1.1.1. Paraméterezés és használat

Nézzük meg a módszer paramétereit:
priomap
A csomag prioritását határozza meg, kernel hozzárendelés alapján, hogy melyik csomag melyik kötegbe tartozik. Ezt a csomag TOS nyolcasa alapján térképezi fel. Íme a TOS nyolcas:



A négy TOS bit a következő:

```
Bináris Decimális Jelentés
----------------------------------------
1000    8         Minimális várakozás (Minimize delay - md)
0100    4         Maximális áteresztés (Maximize throughput - mt)
0010    2         Maximális megbízhatóság (Maximize reliability - mr)
0001    1         Minimális költség (Minimize monetary cost - mmc)
0000    0         Normális szolgáltatás (Normal Service)
```

A TOS mező értéke a duplája a TOS bitek értékének, a tcpdump -v -v megmutatja a teljes TOS mezőt (nem csak a 4 bitet). Ezt láthatjuk az első oszlopban ebben a táblázatban:

```
TOS     Bitek Jelentés                Linux Prioritás   Köteg
-------------------------------------------------------------
0x0     0     Normális szolgáltatás   0 Legjobb hatás   1
0x2     1     Minimális költség       1 Kitöltés        2
0x4     2     Maximális megbízhtóság  0 Legjobb hatás   1
0x6     3     mmc+mr                  0 Legjobb hatás   1
0x8     4     Maximális áteresztés    2 Nagy mennyiség  2
0xa     5     mmc+mt                  2 Nagy mennyiség  2
0xc     6     mr+mt                   2 Nagy mennyiség  2
0xe     7     mmc+mr+mt               2 Nagy mennyiség  2
0x10    8     Minimális várakozás     6 Interaktív      0
0x12    9     mmc+md                  6 Interaktív      0
0x14    10    mr+md                   6 Interaktív      0
0x16    11    mmc+mr+md               6 Interaktív      0
0x18    12    mt+md                   4 Int. Nagy menny.1
0x1a    13    mmc+mt+md               4 Int. Nagy menny.1
0x1c    14    mr+mt+md                4 Int. Nagy menny.1
0x1e    15    mmc+mr+mt+md            4 Int. Nagy menny.1
```

A második oszlop a TOS bitek értéke, a következő oszlopban a magyarázatukkal.
A következő oszlop az ahogyan a Linux kernel értelmezi ezeket a csomagokat, azaz milyen prioritást rendel hozzájuk. Végül a priomap eredménye:

```
1, 2, 2, 2, 1, 2, 0, 0 , 1, 1, 1, 1, 1, 1, 1, 1
```

Ez azt jelenti, hogy a 4-es prioritás az 1. kötegre képződik le. A priomap lehetőséget ad arra is, hogy a magasabb (>7) prioritásokat is listázzunk, amik nem felelnek meg a TOS leképezésnek, mert más célból lettek beállítva. A TOS-ról bővebben az RFC1349-ben

olvashatunk.

txqueuelen

Ez a sor hossza az eszköz beállításaiból kinézve, ezt az ifconfig és ip parancsokkal lehet ellenőrizni és beállítani (pl.: a sor hosszának beállítása 10-re, „ifconfig eth0 txqueuelen 10")

Megjegyzendő, hogy ezt a paramétert tc-vel nem lehet beállítani.

2.1.2. A vezérjeles vödör algoritmus (Token Bucket Filter)

Ez egy egyszerű sorbanállási módszer, amelyben az érkező csomagokat egy előre meghatározott mértékben engedi át, de lehetőséget ad rövid tüskékre, amikor az előre meghatározott sebességet (rate) át lehet lépni.

A TBF meglehetősen pontos, emellett hálózat- és processzor barát. Ez lehet az első szóba jöhető választás, amikor le szeretnénk lassítani egy hálózati eszköz kimenő forgalmát.

A TBF megvalósítása: puffer (vödör), állandóan kitöltve néhány információs darabbal (vezérjel – token), ez egy bizonyos sebességgel (rate) rendelkezik. A puffer másik legfontosabb paramétere a mérete, ennyi jelet tud eltárolni.

Minden egyes vezérjel egy bejövő csomagot gyűjt be, és ezután törlődik a vödörből. Három lehetséges eset lehet:

- az adat ugyanolyan sebességgel jön, mint a TBF mértéke. Ekkor minden bejövő csomagnak megvan a megfelelő jele, így várakozás nélkül tovább haladhat
- lassabban jönnek a csomagok, mint a TBF áteresztő képessége. Ekkor csak egy része törlődik a vezérjeleknek, így ezek felhalmozódnak egészen a vödör méretéig. Ilyenkor lehetővé válik, hogy a nem használt jeleket egy a TBF áteresztőképességénél nagyobb sebességben érkező csomagokra használjuk fel, ekkor beszélünk tüskéről (burst).
- ha a csomagok a TBF áteresztőképességénél gyorsabban jönnek, akkor hamar elfogyhatnak a vödör jelei, ami a TBF saját elfojtásához vezet egy időre. Ha huzamosabb ideig ez a jelenség áll fenn, akkor a csomagok eldobásra fognak kerülni

A harmadik lehetőség a gyakorlatban fontos, mert ez ad lehetőséget a sávszélesség szabályozására.

A jelek felhalmozása lehetőséget biztosít bizonyos sebesség megugrásra (tüske), de ha huzamosabb ideig többlet csomag jön, akkor a csomagoknak várakozniuk kell, ezután el is dobódhatnak.
Gyakran a megvalósításban nem jelek vannak, hanem byte-ok.

2.1.2.1. Paraméterek és használat

Annak ellenére, hogy nem szükséges ezeket megváltoztatni, a TBF rendelkezik párral.

limit vagy *latency*

A limit azoknak a byte-oknak a száma, amelyek sorban állhatnak miközben a érvényes vezérjelre várnak. Ezt egy másik úton is be lehet állítani, mégpedig a latency paraméterrel, ami a csomagok TBF-ben eltöltött maximális idejét határozza meg.

burst/buffer/maxburst

Ez a vödör mérete, byte-okban. Ennyi csomagot képesek a vezérjelek azonnal átvinni. Általában a nagyobb forgalomarányhoz nagyobb pufferméret tartozik. Ha a puffer túl kicsi, akkor csomagok dobódhatnak el, mert több vezérjel érkezik egy időegység alatt, mint amekkora a puffer mérete.

mpu

A kisméretű csomagok nem, annyi sávszélességet használnak el, mint amekkorák (a nulla méretű csomag nem 0-át használ el). Általában ez az ethernetnél 60 byte, ennél kevesebbet nem használnak a csomagok. Tehát a Minimum Packet Unit a minimálisan felhasznált vezérjelek száma.

rate

Ez a sebesség, amellyel a csomagok általános esetben haladhatnak.

Ha a vödrünk több vezérjelet tartalmaz és megengedett a kiürülése, akkor ezt alapból végtelen sebességgel teheti (tehát a jeleink azonnal elfogyhatnak). Ha ezt nem szeretnénk megengedni, akkor a következő paraméterekre lesz szükségünk:

peakrate

Ha vannak készenálló jeleink, és jönnek a csomagok, akkor azok azonnal továbbításra kerülnek tetszőleges sebességgel. De mi nem ezt szeretnénk, különösen nem akkor, amikor nagy vödrünk van. Ekkor a peakrate paraméter beállításával lehet

megmondani, hogy a vödörből a jelek milyen gyorsan törlődhetnek. Mégpedig úgy, hogy két csomag elküldése között szabályozzuk az eltelt időt.

mtu/minburst

A peakrate értéknek csak úgy van értelme, ha az a rate-nél nagyobb. Ez valójában azt jelenti, hogy egy második vödröt rakunk a vezérjeles vödör után. A maximális peakrate kiszámolása: a beállított mtu szorozva 100 (pontosabban, HZ, 100 egy Intel alapú gépen és 1024 egy Alphán).

2.1.2.2. Példa beállítás

Egy egyszerű, de hasznos beállítás:

```
# tc qdisc add dev ppp0 root tbf rate 220kbit latency 50ms burst 1540
```

Miért is hasznos ez? Ha van egy hálózati eszközünk nagy várakozási sorral (pl.: egy DSL vagy kábel modem), ami egy gyors eszközzel kommunikál (ethernet kártya), akkor a feltöltés teljesen ellehetetleníti az interaktivitást. Ez azért van, mert a feltöltés megtölti a várakozási sort a modemben, ami eléggé nagy, hogy megfelelően nagy áteresztőképességet tudjon biztosítani a feltöltéshez. De ebben az esetben nem ez az, amit mi szeretnénk. Mi a feltöltés mellett még más tevékenységeket is akarunk végezni.
A példában lelassítjuk az adatok küldését annyira, hogy a modemben ne keletkezzen sor, így a sort áthelyeződik a Linuxra, ahol a sor méretét kényelmesen be lehet határolni.

2.1.3. A véletlenszerű egyenlő esélyű sor (Stochastic Fairness Queueing, SFQ)

Ez az egyenlő esélyű sorba állítási (fair queueing) algoritmus család egyik egyszerűen megvalósítható tagja. Kevésbé pontos, mint a többi hasonló algoritmus, de kevesebb számítást igényel, miközben biztosítani tudja az egyenlő esélyeket.
Az SFQ kulcsszava az adatfolyam („beszélgetés”), ami megfelel egy TCP kapcsolatban lebonyolított kommunikációnak vagy egy UDP adatfolyamnak. A forgalmat felbontjuk ezeknek az adatfolyamoknak megfelelő számú FIFO sorra. Majd küldésnél round robin ütemezés szerint vesszük az elemeket minden sorból fordulónként. Így nagyon egyenlő

tulajdonságot kapunk, és egyik adatfolyamnak sem lesz lehetősége, hogy lefogja a kapcsolatot. Azokban az esetekben, amikor a különböző sorokban a csomagok mérete lényegesen eltérő, és minden körben egy-egy csomagot veszünk ki a sorok elejéről, akkor egy nagyobb csomagmérettel rendelkező sor nagyobb sávszélességet használ el a kisebb csomagmérettel rendelkező sorokkal szemben. Ez esetben a körönként nem csomagszámot számítunk, hanem byte-okat.

Az SFQ azért véletlenszerű, mert nem csinál minden egyes adatfolyamhoz egy sort, hanem egy algoritmus alapján felosztja a forgalmat egy korlátozott számú sorra egy hashelő algoritmust használva. A hash miatt több adatfolyam is kerülhet egy részbe, ami megfelezi az egyes adatfolyamok csomagküldési lehetőségét, ezzel megfelezve a hatékony sebességet. Hogy ezt megelőzzük, az SFQ gyakran változtatja a hashelő algoritmusát, így két ütköző adatfolyam csak néhány pillanatra van ebben az állapotban.

Fontos megjegyezni, hogy az SFQ akkor válik hasznossá, amikor a kimenő forgalom nagyon terhelt. Nincs jelentősége, ha a Linuxon nincs sorbanállás. Későbbiekben leírásra kerül, hogy hogyan használjuk az SFQ-t más módszerekkel együtt.

2.1.3.1. Paraméterezés és használat

perturb

Ennyi másodpercenként választ új hash függvényt. Ha nem állítjuk be, akkor soha nem lesz új, ez nem ajánlott. 10 másodperc jó érték.

quantum

Azoknak a byte-oknak a mennyisége, amik kiállhatnak a sorból mielőtt a következő sorra kerül a sor. Az alapértelmezett 1 maximális méretű csomag (MTU méretű). Ezt az értéket nem tanácsos az MTU értéke alá állítani.

limit

A maximális csomagszám, amit az SFQ sorába berakhatunk (e fölött a csomagok eldobásra kerülnek)

2.1.3.2. Példa beállítás

Ott ahol olyan eszköz van, aminél a kapcsolat sebessége megegyezik az aktuális sebességgel, mint például egy telefonmodem, ez a beállítás elősegíti az egyenlőséget:

```
# tc qdisc add dev ppp0 root sfq perturb 10
# tc -s -d qdisc ls
qdisc sfq 800c: dev ppp0 quantum 1514b limit 128p flows 128/1024 perturb
10sec
 Sent 4812 bytes 62 pkts (dropped 0, overlimits 0)
```

A 800c-es szám a qdisc-hez automatikusan hozzárendelt kezelési szám. A limit azt jelenti, hogy 128 csomag várakozhat a sorban. 1024 hash-vödrünk van, amiből 128 lehet egyszerre aktív (ugyanis nem lehet több csomag a sorban). Minden 10 másodpercben a hash függvény újradefiniálódik.

2.2. Haladó és kevésbé népszerű módszerek

2.2.1. bfifo/pfifo

Ezek az osztálytalan sorok egyszerűbbek, mint a pfifo_fast abban, hogy itt már nincsenek belső kötegek, azaz az összes forgalom egyenlő (bejövő csomagok tulajdonsága alapján). Ennek van egy fontos előnye, ad néhány statisztikai adatot. Tehát ha nincs szükségünk a forgalom alakítására vagy a csomagok priorizálására, akkor tudjuk ezt a módszert használni, hogy visszanézzük az eszköz használtságát.
A pfifot csomagszámmal, a bfifot byte-okkal használjuk.

2.2.1.1. Paraméterezés

Limit

Ez a paraméter határozza meg a sor hosszát (pfifonal csomagokban, bfifonal byte-okban). Alapértelmezésben ez az érték az eszköz txqueuelen értékével egyezik meg (lásd pfifo_fast).A bfifonal ez txqueuelen*mtu.

2.2.2. Clark-Shenker-Zhang algoritmus (CSZ)

Ez az algoritmus inkább elméleti, gyakorlatban kevésbé alkalmazott.
Bővebben a „David D. Clark, Scott Shenker and Lixia Zhang Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism” könyvben írnak róla.
Az algoritmus fő gondolata, hogy létrehoz egy WFQ folyamot minden garantált szolgáltatásnak (guaranteed service), és lefoglalja a maradék sávszélességet egy nem valós (dummy) 0-folyamnak. A folyam létrehozása egy szolgáltatási szerződés alapján történik. A szerződés szól a felhasználó felé, hogy nem küld a megegyezettnél több csomagot, és szól a hálózat felé is, hogy az a megengedett határon belül, olyan körülményeket biztosít, mint egy terheletlen hálózat esetén. A 0-folyam magában foglalja a megjósolt szolgáltatásokat és a „legjobb törekvésű forgalmat” (best effort traffic). Ezt a folyamot egy prioritásos ütemező kezeli, a megjósolt szolgáltatásoknak adva a legnagyobb prioritást, a maradék a legjobb törekvésű csomagoké. A folyamok felépítéséhez és annak állapotának megtervezéséhez egy algoritmusra van szükség. Ezt a feladatot látja el az RSVP, azaz felméri az erőforrásokat, majd le is foglalja azokat.
A CSZ folyamok nincsenek sávszélességben korlátozva. Feltételezi, hogy a folyam áteresztésére vonatkozó engedélyek a QoS hálózat határán vannak, és nem szükséges további szabályozás. Bármilyen kísérlet a folyam tökéletesítésére vagy szabályozására, például a vezérjeles vödrös algoritmussal azonnal előidéz nem kívánt várakozásokat és megnöveli az „idegességet”.
Egyenlőre csak a CSZ képes valódi garantált szolgáltatásokat kezelni. Más módszerek nem képesek biztosítani a várakozásokat.

2.2.3. Dsmark

(RFC2474, RFC2475, RFC2597 and RFC2598)
A Dsmark egy olyan sorbaállítási módszer, amely rendelkezik azokkal a képességekkel, ami a „Megkülönböztetett Szolgáltatásokban” kell (Differentiated Services). Ezek a „megkülönböztetett szolgáltatások” egyike a QoS architektúráknak (a másik az „integrált

szolgáltatások”), melyek a csomagok IP-fejléc DS mezőjének az értékén alapulnak. Az első megoldás arra, hogy az IP felajánljon néhány QoS szintet, a Type of Service mező (TOS byte) volt az IP fejlécében. Ennek az értéknek a megváltoztatásával tudunk változtatni a magas/alacsony szintű áteresztés, várakozás és megbízhatóság között. De ez nem lát el minket elegendő rugalmassággal, amikor szükség van új szolgáltatásokra (mint például valós idejű alkalmazások, interaktív alkalmazások stb.). Ez után új architektúrák jelentek meg. Ezek közül az egyik a „megkülönböztetett szolgáltatások”, mely megtartotta a TOS biteket és átnevezte DS mezővé.

2.2.3.1. A „megkülönböztetett szolgáltatások” irányelve

A „megkülönböztetett szolgáltatások” csoport orientált. A folyamok az „integrált szolgáltatások” célja lesz, itt most folyam halmazokkal foglalkozunk, és különböző tulajdonságokat alkalmazunk attól függően, hogy egy csomag melyik halmazba tartozik.
Amikor egy csomag egy határ csomóponthoz (node) érkezik (ez a belépő csomópontja a „megkülönböztetett szolgáltatások” tartománynak), belépve oda a tartomány egy eljárásmódot rendel a csomaghoz illetve megjelöli azt a DS mező alapján. Ez az érték (jel) lesz az, amit a belső csomópontok a tartományban néznek majd és ami alapján meghatározzák milyen QoS szintet kell majd alkalmazniuk rá. Következmény, az összes csomag, ami belép ebbe a tartományba osztályozva lesz. Egyszer belép, ekkor megkapják az osztályozás alapján a szabályokat, ezután minden keresztezett csomópont ugyanazt a QoS szintet alkalmazza. (Valójában lehet alkalmazni saját eljárást a tartományon belül is, de néha figyelembe kell venni, amikor másik tartományhoz csatlakozunk)

2.2.3.2. Munka a Dsmarkkal

Ahogy eddig is kitűnt, kétféle csomópontot különböztetünk meg, a határon lévőket és a belsőket. Ez a forgalom útjának két fontos pontja. Minkét típusnál a csomag osztályozásra kerül érkezésükkor. Ez sok helyen használva lesz a DS feldolgozás során mielőtt a csomag ténylegesen belép a hálózatba. Ez a módszer biztosít egy struktúrát, amit sk_buff-nak hívnak, ez rendelkezik egy skb->tc_indx mezővel, ami eltárolja nekünk a kezdeti osztályt, amit a DS

eljárás során többször is használni fogunk.
Ezt a kezdeti értéket a DSMARK qdisc-kel lehet beállítani, ezt az IP fejléc DS mezőjéből lehet kinyerni minden egyes érkező csomagnál. Nézzük a DSMARK qdisc parancsot és paramétereit:

```
... dsmark indices INDICES [ default_index DEFAULT_INDEX ] [ set_tc_index
]
```

paraméterek:

indices

A (maszk,érték) párból álló tábla mérete.

default_index

Az alapértelmezett érték, ha az osztályzó nem talált egyezést

set_tc_index

ez egy utasítás a dsmark módszernek, hogy a DS mező alapján tárolja el az értéket a skb->tc_index-be

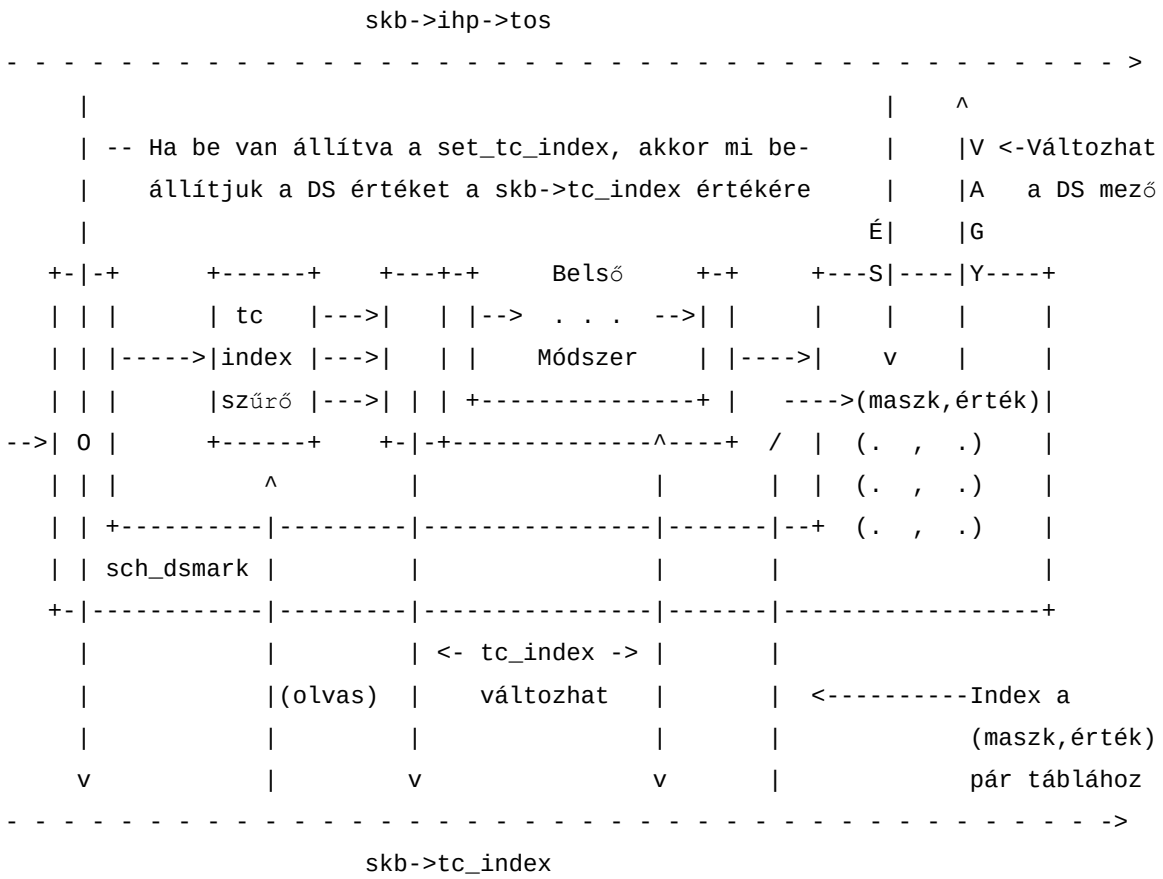
Nézzük a DSMARK működési folyamatát.

2.2.3.3. Hogyan működik az SCH_DSMARK

A qdisc a következő lépésekben kerül alkalmazásra.

- Ha a set_tc_index opció be van állítva, akkor az IP fejlécből kinyerődik a DS mező és eltárolásra kerül az skb->tc_index változóban
- Ezután az osztályozó fut le, ami egy osztályazonosítóval tér vissza. Ez az skb->tc_index-ben tárolódik el. Ha nem talált az osztályozó egyezést, akkor a default_index érték alapján határozódik meg a tárolandó érték. Ha a set_tc_index és a default index sincs beállítva, akkor a keletkező érték megjósolhatatlan.
- Miután ez el lett küldve a belső sorbaállító módszernek (qdiscs), ahol újra lehet használni, ez is egy osztályazonosítóval tér vissza az skb->tc_index-ben tárolva. Ezt az értéket használjuk a maszk-érték táblázat indexelésére. A végső eredményt az alábbi képlet adja: Új_Ds_mező = (Régi_DS_mező & maszk) | érték

A folyamat:



Hogyan jelölünk meg egy csomagot? Csak változtassuk meg az osztályhoz tartozó maszkot és értéket:

```
tc class change dev eth0 classid 1:1 dsmark mask 0x3 value 0xb8
```

Ez megváltoztatja a (maszk,érték) párt a táblázatban, ezután a 1:1 osztályazonosítóhoz ez fog tartozni. Ezt az értéket szinte kötelező megváltoztatni, mert az alapértelmezettje nem lehet jó.

A TC_INDEX szűrő még egy kis magyarázatra szorul, így essen néhány szó arról, hogyan is illeszkedik ez bele a modellünkbe. Emellett a, TC_INDEX szűrő használható más beállításoknál is.

2.2.3.4. A TC_INDEX szűrő

Az alábbi utasításrészlet az alapja, hogy TC_INDEX szűrőt hozzunk létre:

```
... tcindex [ hash SIZE ] [ mask MASK ] [ shift SHIFT ]
            [ pass_on | fall_through ]
            [ classid CLASSID ] [ police POLICE_SPEC ]
```

Egy példa:
Feltesszük, hogy a fogadott csomag EF-fel van jelezve. Az ehhez tartozó érték (RFC2598 szerint) a 101110. Ez azt jelenti, hogy a DS mező értéke 10111000 (0xb8 hexadecimálisan). Ahogy eddig, most is a DS mező értéke átkerül a skb->tc_index változóba. Ezután alkalmazzuk a TC_INDEX szűrőt az aktuális sorbaállítási módszerhez, a következő műveletekkel (MASK és SHIFT a tc_index paraméterei):
érték1 = skb->tc_index & MASK
Kulcs = érték1 >> SHIFT
Legyen: MASK=0xfc, SHIFT=2, ekkor
érték1 = 10111000 & 11111100 = 10111000
kulcs = 10111000 >> 2 = 00101110 (0x2e hexadecimálisan)
Ezt a visszaadott értéket feleltetjük meg a belső sorbaállítási módszer szűrőjének kezelőjéhez. Ha létezik szűrő ezzel az értékkel, akkor az ehhez tartózó irányelv- és mérési feltételek ellenőrzésre kerülnek és egy új osztályazonosító keletkezik az skb->tc_index változóba. Ha nincs szűrő, ami ennek az értéknek megfelelne, akkor az eredmény a fall_through flagtől fog függeni. Ha ez be van állítva, akkor a kulcs értékkel tér vissza mint új osztályazonosító. Ha nincs, akkor hibával tér vissza és a feldolgozás folytatódik a hátralevő szűrőkkel. A fall_throgh flag használatával óvatosan kell bánni.
A hash paraméter a táblázat méretéhez kapcsolódik. A pass_on-t akkor kell használni, ha azt szeretnénk, hogy ha nincs a szűrő eredményével egyező osztályazonosító, akkor próbálja a következő szűrőt. Az alapértelmezett a fall_through.

Ez egy eléggé hatékony módszer, bár itt a bemutatása nem részletes.

2.2.4. A bejövő forgalomi sorbaállítási módszerek

Eddig az összes módszer, amivel foglalkoztunk a kimenő forgalmat állította sorba mielőtt az ténylegesen belépne a hálózatba (egress qdiscs). Emellett minden hálózati eszköznek lehet a bejövő forgalmat sorbaállító módszere is (ingress qdisc), ami a csomagokat nem fogja kiküldeni a hálózat fele. Ehelyett lehetőséget biztosít, hogy az eszközön bejövő forgalmat szűrjük. Ekkor nem vesszük figyelembe, hogy a csomag az adott gépre jön vagy csak átmenő forgalmat képez (forward).
Ahogyan a többi szűrő ez is rendelkezik egy teljesértékű vezérjeles vödör szűrő megvalósítással. Sok funkcióval rendelkezik, amelyekkel a bejövő forgalomra alkalmazhatunk bizonyos szabályozásokat.
Használata egyszerű:

```
# tc qdisc add dev eth0 ingress
```

2.2.5. Random Early Detection (RED)

Ez a rész a nagyobb, gerinchálózatok sorbaállításainak egyik lehetősége, ahol a sávszélesség gyakran több 100 Mbit környékén van. Ez más hozzáállást igényel mint egy otthoni ADSL modem.
A megszokott viselkedés egy routereknél a „tail-drop” (farok levágás). Ez azt jelenti, hogy jönnek a csomagok és ezeket rakja a sorba ill. továbbküldésnél veszi ki onnan, ha a sor tele van, akkor az újonnan érkezett csomagokat eldobja. Ez nem túl barátságos, és az újbóli átvitel szinkronizálásához vezethet, ez pedig hirtelen csomagvesztést, majd újrakküldést okoz, ami csak jobban le fogja terhelni a torlódásban szenvedő routert.
A gerinc routerek gyakran nagy sorral vannak megvalósítva. Míg ez kedvez a jó áteresztőképességnek, addig a válaszidőt tartósan megnöveli, és a TCP kapcsolatokat nem megszokott viselkedésre készteti a torlódás alatt. A „farok levágásos” módszer alkalmazása tovább növeli a kellemetlenségeket az Interneten, és a hálózatot használó „barátságtalan” programok száma is növekszik. Van egy módszer, ami ilyen esetekben jöhet jól, ez a RED, ami a Random Early Detect rövidítése, más néven Random Early Drop, ami a működési elvre is utal. (Ez a módszer a Linux kernelben is megtalálható.) (A RED kitalálói Sally Floyd és Van Jackobson 1990) Ebből is látszik, hogy a RED, az eddigi módszerekkel ellentétben,

megpróbálja a torlódást megelőzni, nem pedig annak kialakulása esetén megpróbálni megszüntetni azt.

Azért ez a módszer sem jelent megoldást minden problémára, még mindig leszek programok, melyek képesek nagyobb szeletet kihasítani a sávszélességből, de a RED-nél ezek kevesebb kárt tesznek majd más kapcsolatok áteresztőképességében illetve várakozási idejükben.

A RED statisztikai valószínűséggel eldobja a csomagokat egy folyamból, mielőtt az elérné a teljesítő képességének határát (azaz eléri az eldobási küszöböt). Ez a torlódásos gerinc kapcsolatot óvatosan lelassítja, és megelőzi a szinkronizálás újraközvetítését. Ez a módszer ráadásul segít a TCP-nek, hogy gyorsabban megtalálja a megfelelő sebességet, úgy hogy megengedi néhány csomag eldobását hamarabb, így a sor hossza alacsonyabb marad, ezáltal a várakozás csökken. A valószínűsége annak, hogy egy csomag eldobásra kerül egy egyedi kapcsolatnál a sávszélesség használatával arányos, a szállított csomagszám helyett. A RED-et úgy tervezték, hogy a TCP-vel szorosan együtt tudjon működni. Mert itt a kapcsolat küldő szereplője nem tájékozódik a torlódásról illetve arról, hogy lassítania kéne közvetlenül, csak onnan, hogy vagy idő túllépést észlel, vagy duplikált ACK-val találkozik. Mint láthattuk a korábbi TCP-vel foglalkozó fejezetben, a TCP pont ezen tulajdonságokat használja fel a torlódás felderítő algoritmusaiban (lassú kezdés). Az eldobás drasztikus megoldásnak tűnhet, de a RED szerint jobb kevesebb csomagot előbb eldobni, mint később sokat.

Az eldobás eldöntéséhez az egyik legfontosabb tényező a router sorának a hossza. De nem annak a pillanatnyi értékét veszi figyelembe, hanem egy átlagértékét:

átlaghossz = (1 – súly) * átlaghossz + súly * hossz

Ahol 0< súly <1 és a hossz a sor hossza, amikor a becslést csináljuk. Az átlagérték számítása szoftveres környezetben minden csomag megérkezésénél történik, míg hardveres megvalósításnál bizonyos időközönként. Az átlagérték a pillanatnyi értékkel szemben, kevésbé érzékeny a tüskékre, mely az Internet egyik velejárója, ezért érdemes azt használni.

A RED használatához további három paramétert kell megadnunk: min, max és burst. A min-nel lehet beállítani azt az értéket, ami előtt az eldobások megkezdődnek. Az érték kisebb, mint a sor mérete, és az érték byte-okban értendő. A max-szal állítjuk be azt a maximum értéket, ami alatt igyekszik majd tartani az algoritmus az értéket. A burst azoknak a csomagoknak a maximális száma, melyek hirtelen képesek átmenni.

A min érték meghatározásához figyelembe kell venni a kívánt legmagasabb várakozási értéket és a sávszélességet, majd ezeket kell összeszorozni. Ha túl kicsire állítjuk a min

értéket, akkor az áteresztőképesség lesz kevés, ha pedig túl nagyra vesszük, akkor a várakozás lesz nagy. A min érték kicsire állításával nem érhetjük el az MTU csökkentésével járó válaszadás javulást lassú kapcsolatoknál. A max értéket legalább a min kétszeresére kell meghatározni, hogy megelőzzük a szinkronizációból adódó problémákat. Lassabb kapcsolatok esetén ez akár négyszerese vagy többszöröse kell, hogy legyen a min-nek. A burst-tel határozzuk meg, hogy a RED miként reagáljon a hirtelen nagy forgalomra. Ez az érték mindenféleképpen legyen nagyobb, mint a min/avpkt.
A fentebbi paramétereken kívül még a következő paraméterek is beállíthatók: limit és avpkt. A limit meghatározza byte-okban, hogy a RED mikor váljon „farok eldobóvá". Ezt tanácsos magasan tartani (kb. A max nyolcszorosán). Az avpktval az átlagos csomagméretet állíthatjuk be. Ez 1500-as MTU mellett 1000 körül lehet megfelelő, gyors kapcsolat esetén.
Létezik a RED-nek több módosított változata. Például a Generic RED, amely néhány belső sorral rendelkezik. A csomagok a tcindex mező alapján sorolódnak be az egyes sorokba. Minden sor a saját paraméterei alapján dönt a csomagok eldobandóságáról. A CISCO-ék a Distributed Weighted RED-et alkalmazzák (Súlyozottan elosztott RED).

2.2.6. DECbit

A DECbit-et használó módszer a RED elődjének is tekinthető, mivel a torlódás felderítő módszere hasonló az előbbiéhez, de időben előbb jött létre. Ez a módszer kevésbé építkezik egy olyan elterjedt protokollra, mint a RED a TCP-re. Itt sokkal szorosabb együttműködés kell mind a külső, mind a fogadó területén, hogy a router üzenetét megértsék, illetve továbbítani tudják azt a megfelelő helyre. E miatt kevésbé tűnik drasztikusnak, mint az eldobásos RED. Ez a módszer is a torlódás megelőzésre épül. (Ez a két módszer ebben is különbözik a többitől, azaz torlódás megelőzés nem ugyanaz, mint a torlódás szabályozás.) A RED-től még az is megkülönbözteti, hogy nem ugyanazzal a módszerrel döntik el, hogy egy csomag mikor kerül eldobásra, illetve mikor mondja azt, hogy torlódást észleltem.
Ennek a módszernek az alapja a DECbit, ami a csomag fejlécében egy másra nem használt bit. Itt most ez kitüntetett szerepet fog játszani. Az a router amelyik azt észleli, hogy hamarosan torlódás következik be, ezt a bitet 1-re állítja. Az észlelés a router saját sorának az állapotával függ össze. Ez a csomag nyilván a küldőtől jött, és nem feléje tart, de őt kellene tájékoztatni, hogy lassítson. Ezt legegyszerűbben úgy teszi meg, hogy a fogadó fele már az

1-es DECbittel rendelkező csomag érkezik meg, ezt a fogadó észleli, és az ő ACK (válasz) csomagjában is beállítja ezt a bitet. Így ez eljutva a küldés eredeti helyére, a küldő is tapasztalhatja a torlódás közeledtét, és e szerint reagálhat.
A módszer egyébként nem egy csomag 1-es bitjére hagyatkozik, hanem 50%-os többség kell, hogy a torlódási ablakot a 0.875-szeresére csökkentse a küldő. (Ez a rész erősen hasonlít a TCP-jére.)

2.2.7. Weighted Round Robin (súlyozott round robin, WRR)

Ez a módszer nincsen benne az alap Linux kernelben, de patchelés után használható. A WRR módszer szétosztja a sávszélességet a saját osztályai között, súlyozott round robin algoritmust használva. Ez a CBQ-hoz hasonlóan osztályokat tartalmaz, amikre tetszőleges számú más módszer illeszthető. Az osztályok a súlyok arányában kapnak sávszélességet, amikor azt az igények megkövetelik. Az egyes osztályok súlyait a tc programmal lehet beállítani. De automatikussá is lehet tenni, fordított arányban az egyes osztályok adatforgalmával.
A WRR rendelkezik egy beépített osztályozóval, ami különböző forrásból érkező csomagokat illetve különböző helyre tartó csomagokat más-más osztályba rakja. Ezt teheti MAC vagy IP alapján.
Ez a módszer olyan helyekre jó, ahol sok egymástól független gép között kell szétosztani az Internetet.

2.2.8. Tanácsok, mikor melyik sort használjuk

Összefoglalva, ezek az egyszerű sorok a csomagok újrarendezésével, lassításával és eldobásával irányítják a forgalmat. Most következzen néhány tanács összefoglalva, mikor melyik sor a legmegfelelőbb.
Ha csak egyszerűen az a célunk, hogy a forgalmat lassítsuk, akkor a vezérjeles vödrös algoritmus a jó választás. Ez a vödör skálázhatóságának köszönhetően, egészen nagy sávszélességig alkalmazható eredményesen.
Ha a kapcsolat igazán terhelt, és biztosak szeretnénk lenni abban is, hogy egyik adatfolyam

se legyen domináns, azaz ne nyomja el a többit, akkor a véletlenszerű egyenlő esélyű sorbaállítás használata a megfelelő.
Nagy gerinc hálózatoknál a Random Early Detection a szóbajövő algoritmus, de ekkor már tényleg tudnia kell az embernek mit is akar.
Ha nem csak a kimenő forgalommal akarunk foglalkozni, akkor az ingress módszert kell alkalmazni. Ekkor nem forgalom alakításról, hanem „politikáról" beszélünk. Ha a forgalom továbbításra kerül (forwarding), akkor a sorokat célszerű a kimenő eszközhöz társítani, így több kimenő eszköz esetén, más-más módszereket alkalmazhatunk, amit éppen a forgalom megkíván. Ha viszont a több kimenő eszközt egyszerre, globálisan szeretnénk kezelni, akkor a bejövő eszközön kell alkalmazni, a megfelelő módszert.
Abban az esetben, ha szabályozni nem, de a forgalmi terheltséget figyelni szeretnénk, akkor használjuk a pfifo sort (ne a pfifo_fast-ot). Ez nem rendelkezik különálló belső kötegekkel, de naplózásra kiválóan alkalmas lehet.

2.3. Osztályos sorbaállítási módszerek

Az osztályos módszerek ott lehetnek hasznosak, ahol több különböző forgalom típus van, melyek más-más bánásmódot igényelnek. Az egyik ilyen módszer a CBQ, azaz az „osztály alapú sorbaállítás" (class based queueing). Széleskörben elterjedt, hogy az emberek a sorbaállítást osztályokkal egyedül a CBQ-val azonosítják, de ez nem így van.
Arról van szó, hogy a CBQ a legrégebbi a társai közül és a legösszetettebb is. De ez az összetettség és a jó dokumentációk hiánya miatt gyakran nem azt csinálja, amit szeretnénk.

2.3.1. Folyamok az osztályos sorbaállítási módszereknél és osztályoknál

Amikor a forgalom belép az osztályos sorba, néhány osztályhoz el kell jutnia, tehát osztályozni kell azt. Hogy el tudjuk dönteni mit kell csinálni egy csomaggal, az un. „szűrőt" (filter) hívjuk segítségül. Fontos lehet tudni, hogy a szűrő a módszeren belülről hívódik meg, nem pedig más kerülő úton.
A szűrők, melyek a módszerhez kapcsolódnak, egy döntést hoznak, ezt felhasználva a qdisc

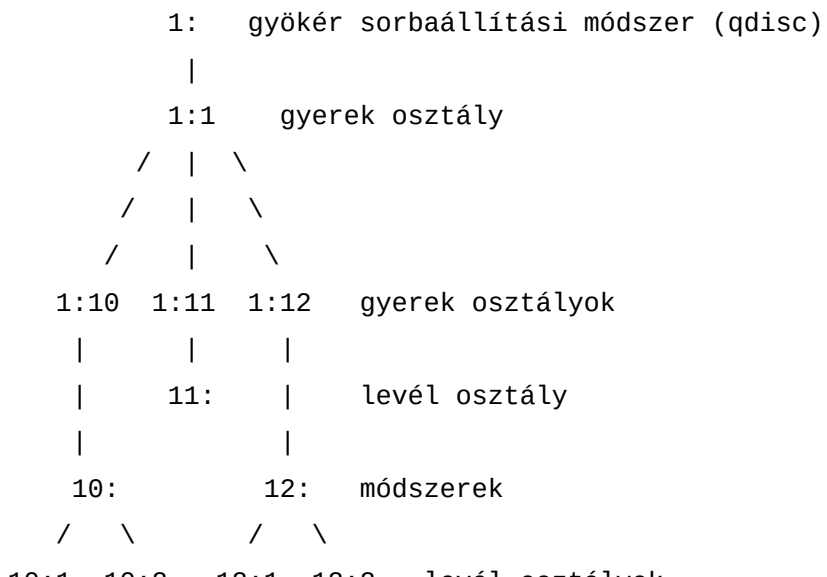
besorolja (enqueueing) a csomagot a megfelelő osztályba. Ez az alosztály még más szűrőket is próbálhat, további tulajdonságok ellenőrzésére. Ha ezt nem teszi, akkor a csomag az osztály saját sorbaállítási módszerének a sorába állítódik be.
Amellett, hogy más módszereket is tartalmaz, a legtöbb osztályos módszer alkalmas formálásra is. Ez hasznos, mert így egyszerre lehet alkalmazni a csomag ütemezőt (pl. SFQ-val) és a sebesség szabályzót. Ez abban az esetekben szükséges, amikor egy nagyobb sebességű eszköz (ethernet) csatlakozik egy lassabbhoz (modem). Ha csak SFQ-t alkalmaznánk, akkor lényegében nem történne semmi, mert a csomagok belépés után várakozás nélkül egyből tovább mennének a routeren, mert a kimenő eszköz gyorsabb, mint az aktuális kapcsolat sebessége. Ezért itt nem keletkezik sor, nincs mit ütemezni.

2.3.2. A qdisc család: gyökér, nyél, testvér és szülő

Minden eszköz interfésznek van kimeneti gyökér sorbaállítási módszere (egress root qdisc). Ilyen például a korábban is emlegetett osztálytalan pfifo_fast módszer, ez az alapértelmezett. Minden qdiscnek és osztálynak van egy nyele, ami a későbbi beállítások során használunk, hogy megcélozzunk egy qdisc-et. Természetesen minden eszköznek a kimenő módszer mellett van egy bejövő sorbaállítási módszere is, ami a bejövő forgalmat felügyeli.
A qdisc-ek nyele két részből áll, egy fő számból és egy al számból (major és minor): <major>:<minor>. Megszokott, hogy a gyökérnek a nyele a '1:', ami ugyanazt jelenti mint '1:0'. Az alszáma egy módszernek mindig 0 (az osztályoknak nem). Az osztályoknak a főszáma mindig ugyanaz mint a szülőjéé. Ez a fő szám, kimenőre illetve a bemenőre egyedinek kell lennie. Az alszámnak a módszeren belül kell egyedinek lennie, tehát fő számonként.

2.3.2.1. Hogyan használjuk a szűrőket a forgalom osztályozására

Egy tipikus osztályos felépítés így nézhet ki:



A csomagok a gyökérben kerülnek be a sorba, és ott is kerülnek ki véglegesen. A gyökérrel csak a kernel tud beszélni.
Egy csomag például a következő féleképpen osztályozódhat: 1: -> 1:1 -> 1:12 -> 12: -> 12:2
A csomag most a 12:2-es osztályhoz tartozó módszer sorában van. Ebben a példában a fa minden csúcsához egy-egy szűrő van rendelve, melyek az elágazásokban döntenek a csomag további útjáról. De például az alábbi is lehetséges: 1: -> 12:2. Ebben az esetben a szűrő a gyökérhez tartozik, ami úgy dönt, hogy a csomagot közvetlenül a 12:2-nek kell küldeni.

2.3.2.2. Hogyan kerülnek a sorból a hardverhez a csomagok

Amikor a kernel úgy dönt, hogy egy csomagot ki lehet küldeni az interfészhez, akkor a gyökér (1:) kap egy „sorból kiáll” üzenetet, ami továbbítódik 1:1 felé, ami szintén továbbítódik a 10:, 11: és 12: felé, ezek a módszerek is kapnak egy „sorból kiáll” üzenetet. A mostani példában a kernelnek be kell járni az egész fát, mert csak a 12:2 tartalmaz csomagot. Az egymásba ágyazott osztályok csak a szülő módszerrel kommunikálnak, az interfésszel soha. Csak a gyökér kapja az üzenetet a kerneltől.
Így az egésznek a következménye az, hogy egy osztály sose kapja gyorsabban az üzeneteket, mint a szülője engedi. És ez az amit mi éppen akarunk, mert így egy belsőbb osztálynak lehet SFQ módszere, ami nem képes forgalomformálásra csak ütemezésre, és közben egy külső módszer végzi a formálást.

2.3.3. A PRIO sorbaállítási módszer

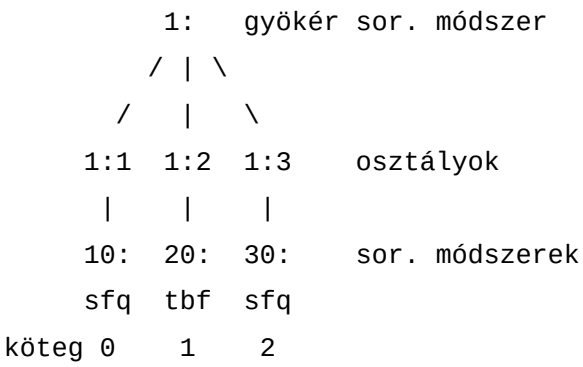
A PRIO nem formál csak az előre beállított szűrők alapján felosztja a forgalmat. Úgy lehetne elképzelni, mint egy „felturbózott" pfifo_fast-ot, minden köteg egy különálló osztály, nem úgy, mint az egyszerű FIFO-nál.
Amikor egy csomag belekerül a PRIO sorába, a szűrő parancsok alapján kiválasztódik neki egy osztály. Alapértelmezésben három osztály jön létre, melyek mindegyike a szegényes FIFO-t tartalmazza, de ezt akármilyen másik módszerre le lehet cserélni.
Amikor egy csomag ki kell, hogy kerüljön a PRIO-ból, először a :1-es osztálynál próbálkozik. A magasabb osztályok csak akkor kerülnek sorra, ha az alacsonyabb nem tudtak csomagot szolgáltatni.
Ez a módszer akkor lehet hasznos, ha priorizálni akarjuk a forgalmat, de nem kizárólag a TOS mező alapján, de kihasználva a tc szűrőjinek előnyét. Más módszereket is lehet társítani a három osztályhoz, és ez az a pont ahol a pfifo_fastot túlnövi, mert ott csak a beépített FIFO állt a rendelkezésünkre.
Mivel ez sem végez formálást, ugyanazok a figyelmeztetések érvényesek itt is, mint az SFQ-nál, tehát csak olyan helyeken használjuk, ahol a fizikai kapcsolat nagyon telített. De fel is oszthatjuk úgy, hogy a belső módszer osztályos legyen, ami elvégzi nekünk a formálást.
A PRIO egy munkamegőrző (work-conserving) ütemező.

2.3.3.1. Paraméterezés és használat

A következő paraméterek használhatóak:
bands: a kötegek száma, ahol minden köteg valójában egy osztály. Ha ezt a paramétert megváltoztatjuk, akkor a következőt is meg kell változtatni.
Priomap: Ha nem szeretnénk a tc szűrőjét használni a forgalom osztályozására, akkor a PRIO a TC_PRIO prioritást nézi a megfelelő besoroláshoz. (Használjuk fel a pfifo_fastnál olvasottakat.)
Abból kifolyólag, hogy a kötegek is osztályok, a következőképpen vannak elnevezve: <főszám>:1 - <főszám>:3 alapértelmezett esetben. Tehát, ha a PRIO módszerünk neve 12:, akkor a tc szűrő a 12:1-esnek adja a legnagyobb prioritást.

2.3.3.2. Példa

Készítsük el az alábbi fának megfelelő beállításokat.



A nagy adatmennyiséggel rendelkező forgalom a 30:-asba megy, míg az interaktív forgalmak a 10:- 20:-asba. Lássuk, konkrétan milyen utasítások szükségesek ezen fa előállításához:

```
# tc qdisc add dev eth0 root handle 1: prio
# tc qdisc add dev eth0 parent 1:1 handle 10: sfq
# tc qdisc add dev eth0 parent 1:2 handle 20: tbf rate 20kbit \
  buffer 1600 limit 3000
# tc qdisc add dev eth0 parent 1:3 handle 30: sfq
```

Nézzük meg mit csináltunk.

```
# tc -s qdisc ls dev eth0
qdisc sfq 30: quantum 1514b
 Sent 0 bytes 0 pkts (dropped 0, overlimits 0)

 qdisc tbf 20: rate 20Kbit burst 1599b lat 667.6ms
 Sent 0 bytes 0 pkts (dropped 0, overlimits 0)

 qdisc sfq 10: quantum 1514b
 Sent 132 bytes 2 pkts (dropped 0, overlimits 0)

 qdisc prio 1: bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
 Sent 174 bytes 3 pkts (dropped 0, overlimits 0)
```

Láthatjuk, hogy a parancs lekérdezése már kis forgalom bonyolítással járt. Ez jelent meg a nullás kötegben. Ezután csináljunk nagy adatmennyiséget produkáló forgalmat, azaz próbáljunk meg egy filet elküldeni, majd újból nézzük meg az eredményt:

```
# scp tc btuska@caesar.elte.hu:./
btuska@caesar.elte.hu's password:
tc                          100% |*****************************|    353 KB
00:00

# tc -s qdisc ls dev eth0
qdisc sfq 30: quantum 1514b
 Sent 384228 bytes 274 pkts (dropped 0, overlimits 0)

 qdisc tbf 20: rate 20Kbit burst 1599b lat 667.6ms
 Sent 2640 bytes 20 pkts (dropped 0, overlimits 0)

 qdisc sfq 10: quantum 1514b
 Sent 2230 bytes 31 pkts (dropped 0, overlimits 0)

 qdisc prio 1: bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
 Sent 389140 bytes 326 pkts (dropped 0, overlimits 0)
```

Mint ahogyan azt vártuk, a forgalom nagy része 30:-asba ment, ami a legalacsonyabb prioritással rendelkezik. Most ellenőrizzük le, hogy az interaktív forgalom, valóban a magasabb prioritású osztályba kerül-e. Csináljunk interaktív forgalmat (pl. ssh, nem scp). Ezután nézzük meg az eredményt.

```
# tc -s qdisc ls dev eth0
qdisc sfq 30: quantum 1514b
 Sent 384228 bytes 274 pkts (dropped 0, overlimits 0)

 qdisc tbf 20: rate 20Kbit burst 1599b lat 667.6ms
 Sent 2640 bytes 20 pkts (dropped 0, overlimits 0)

 qdisc sfq 10: quantum 1514b
 Sent 14926 bytes 193 pkts (dropped 0, overlimits 0)
```

```
qdisc prio 1: bands 3 priomap  1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
Sent 401836 bytes 488 pkts (dropped 0, overlimits 0)
```

Láthatjuk, hogy a dolog működik, mert az összes forgalom a 10:-esbe ment, ami a legnagyobb prioritású. És semmi sem kerül a legalacsonyabb prioritású osztályhoz.

2.3.4. A CBQ sorbaállítási módszer

Már korábban szó esett arról, hogy a CBQ a legösszetettebb, legreklámozottabb, legkevésbé érthető és talán a legravaszabb a fellelhető módszerek közül. Ez nem azért van, mert a készítői gonoszak vagy inkompetensek lennének, távolról sem, inkább azért, mert a CBQ algoritmus nem olyan pontos és nem követi annyira a Linuxos irányvonalat.
Mint ahogyan a többi osztályos módszer is, a CBQ is képes formálásra, de ez az ahol a CBQ nem éppen a legjobban működik. Így kéne működnie: ha egy 10 Mbit/s-os kapcsolatot szeretnénk 1 Mbit/s-ra formálni, akkor a kapcsolat 90%-ban nem kéne, hogy használva legyen. Ha mégis, akkor addig kell visszavenni, amíg 90% alá nem esik. Ezt viszont nagyon nehéz megbecsülni. A CBQ a hardver-rétegtől kért kérések között eltelt ezredmásodpercek számából határozza meg az üres időt. Így közelítőleg meg lehet határozni, hogy egy kapcsolat mennyire telített vagy éppen üres. Ez nem mindig jó, mert nem minden eszköz képes akkora sebességre mint amilyen hálózathoz illeszkedik, például rossz driver miatt.
De sok esetben képes jól működni. Megpróbálom úgy bemutatni a CBQ-t, hogy azt úgy állíthassuk be, hogy a legtöbb esetben képes legyen helytállni.

2.3.4.1. CBQ formálás részletesebben

Mint ahogyan már korábban is említettem, a CBQ úgy működik, hogy megnézi, van-e elég szabad kapacitás és ennek függvényében állítja be a sávszélességet a beállított sebességhez. Magyarul, kiszámolja, hogy az átlagos csomagok között mennyi időnek kell eltelnie. A művelet alatt a tényleges szünetidőt az exponenciális súlyozott mozgó átlaggal (ewma) becsüli meg, ami exponenciálisan veszi figyelembe az új csomagot, szemben a régebbiekkel. A kiszámított szünetidőt kivonjuk a ewma által becsült időből, így megkapjuk az un. „átlagos szünetidőt” (avgidle). Egy teljesen terhelt kapcsolatnak ez az értéke nulla, minden intervallumra jut egy csomag. Egy túlterhelt kapcsolat negatív átlagos szünetidővel

rendelkezik, ekkor a CBQ leállítja a forgalmat egy kis időre. Visszafelé, egy nem terhelt kapcsolat felhalmozhat egy csomó átlag szünetidőt, ami később végtelen sávszélességet eredményez, néhány óra hallgatás után. Ennek megelőzésére szolgál a maxidle paraméter.
Ha túlterhelés van, akkor a CBQ képes magát lefojtani, annyi időre, mint amennyit korábban kiszámolt a csomagok közötti eltelt időre, majd küld egy csomagot és lefojt megint.
Nézzük néhány paraméterét:

avpkt

átlagos csomagméretre becslés byte-okban. A maxidle paraméter kiszámításához szükséges.

bandwidth

az eszköz fizikai sávszélessége, a szünetidő kiszámításához kell.

cell

az idő, ami egy csomag átviteléhez szükséges az eszközön keresztül, lépésekben nőhet a csomag méretétől függően. Általában ez az érték 8. (Kettő hatványnak kell lennie.)

maxburst

Ebből a csomag számból fog a maxidle paraméter meghatározódni. Tehát ha a avgidle a maxidle értéket veszi fel, akkor maxburstnyi csomag képes átmenni, addig amí az avgidle nulla nem lesz. Magasabb érték a nagyobb tüskéknek kedvez. A maxidle értéket közvetlenül nem lehet állítani, csak ezen a paraméteren keresztül.

minburst

már korábban említettem, hogy a CBQ lefojt, ha túlterhelés van. A legjobb megoldás az lenne, ha a kiszámított szünetelési időnként küld egy csomagot. De a Unix alapú rendszereknél nehéz megvalósítani olyan ütemezést ami 10 milimásodpercnél kisebb, ezért jobb hosszabb ideig lefojtani a forgalmat, majd nem egy, hanem minburstnyi csomagot küldeni egyszerre és aztán nem küldeni megint minburst-szörnyi ideig. A várakozási időt offtime-nak nevezik. Magasabb minburst érték pontosabb formáláshoz vezet hosszabb időt nézve, de nagyobb tüskék is lehetnek milimásodperces intervallumokat nézve.

minidle

ha a avgidle nulla alatt van, akkor túlterhelés van és addig kell várni, amíg az avgidle akkora nem lesz, hogy el lehessen küldeni egy csomagot. Hogy elkerüljük, hogy a kapcsolat hosszabb időre leálljon egy hirtelen tüske miatt, az avgidle visszaállítódik a

minidlére, ha már túl alacsony lenne. A minidle-t negatív mikromásodpercekben kell megadni, azaz a 10 azt jelenti, hogy -10 mikromásodperc.

mpu

minimális csomagméret azért szükséges, mert a nulla méretű csomagok, nem nulla méretűek az etherneten, tehát a szállításuk időbe telik. CBQ-nak kell ez az érték, hogy a szünetidőt ki tudja számolni.

rate

az a sebesség, amivel a forgalom elhagyhatja ezt a sort.

A CBQ-nak nagyon sok hasznos kiegészíthetősége van. Például azok az osztályok melyek túlterhelődnek, büntetést kapnak, azaz alacsonyabb prioritásba helyeződnek.

2.3.4.2. A CBQ osztályos tulajdonságai

A szünetidő becslő és formáló tulajdonságai mellett a CBQ hasonlóan viselkedik, mint a PRIO abból a szempontból, hogy az egyes osztályok prioritása különbözik, és az alacsonyabb prioritású osztályok addig mellőzve vannak, amíg van valaki a magasabb prioritású osztályokban.

Ha a hardver-réteg úgy érzi, hogy egy csomagot ki kell küldeni a hálózat felé, akkor egy súlyozott round robin (WRR) algoritmus indul, ez a kisebb prioritás-számú osztályokkal kezd. Ezek csoportosítva lesznek, majd megkérdezik tőlük, hogy van e adatuk. Ha van, akkor egy osztály a sorából kibocsájt valamennyi mennyiségű csomagot, majd a következő osztály próbálkozik.

A súlyozott round robint a következő paraméterekkel lehet szabályozni:

allot

Amikor a CBQ-t megkérik, hogy küldjön egy csomagot ki, akkor próbálkozni fog az összes belső módszernél (az osztályokon belül) egy fordulóban, a priority paraméternek megfelelően. Mikor egy bizonyos osztályra kerül a sor, akkor csak egy korlátozott mennyiségű adatot tud elküldeni. Az allot ez a mennyiséget határozza meg.

prio

A CBQ képes úgy viselkedni, mint a PRIO. A belső osztályok közül a magasabb prioritásúak kapnak addig előnyt, amíg van bennük adat.

weight

Ez segít a súlyozott round robin algoritmusnak. Egy fordulóban az összes osztály esélyt kap a küldésre. Ha van egy osztályunk, ami feltűnően több sávszélességet kap a többinél, akkor meg lesz engedve ennek az osztálynak, hogy több adatot küldhessen egy fordulóban, mint a többiek.

A CBQ összeadja a súlyokat és normalizálja őket, tehát akármekkora számokat használhatunk, csak az arányuk a lényeges. Gyakran a súlyoknak a rate/10 értéket használják, ami jó választásnak tűnik. A normalizált súlyok és az allot paraméter szorzata határozza meg egy osztály egy fordulóban, ha küldhet adatot, akkor mennyit küldhet.

2.3.4.3. A CBQ egyéb paraméterei

Most olyan paraméterekről lesz szó, mely a kapcsolat megosztásához és kölcsönvételéhez szükségesek. A CBQ mindamellett, hogy képes egyszerűen korlátozni az egyes forgalmakat, képes arra is, hogy meg lehessen határozni, hogy egyes osztályok a szabad erőforrásokat kölcsönadják más osztályoknak, azaz ily módon a nem kihasznált sávszélesség is kihasználható egy már terheltebb osztály által.

isolated/sharing

Az az osztály, mely az isolated paraméterrel lett definiálva, nem fog kölcsönadni sávszélességet a testvér osztályoknak. Ez akkor kell, mikor egymással versengő és egymás kölcsönösen nem szerető forgalmak vannak ugyanazon a kapcsolaton, melyek nem akarnak szabad erőforrásokat átadni a másiknak. A sharing paraméter az isolated ellentéte.

bounded/borrow

A bounded pont az isolated másik oldalról nézve, azaz ha a bounded paramétert beállítottuk az osztály létrehozása során, akkor ez az osztály nem fog kölcsönkérni a testvéreitől. A borrow a bounded ellentéte.

Egy tipikus helyzet, ha egy kapcsolaton van két különböző cég, és mindkettőre beállítottuk az isolated és bounded paramétereket, ekkor ők ténylegesen be vannak szorítva a saját sávszélességükbe, mert egyik sem tud a másiktól kölcsönkérni illetve kölcsönadni.

Persze egy osztályon belül lehetnek alosztályok, melyek már más tulajdonságokkal rendelkezhetnek, de csak az adott osztályon belül.

2.3.4.4. Egy példa beállítás

```
          1:          gyökér módszer
          |
         1:1          gyerek osztály
        /   \
       /     \
     1:3     1:4      levél osztályok
      |       |
     30:     40:      módszerek
    (sfq)   (sfq)
```

Egy olyan példát hozunk létre, hogy van két szolgáltatásunk: web, SMTP. A webszerver kapjon 5 Mbit-et, az SMTP kapjon 3 Mbitet, de a kettő együttes forgalma nem haladhatja meg a 6 Mbitet. Az osztályok egymás között tudjanak kölcsönözni és kölcsönkérni is. A kapcsolat fizikai sávszélessége 100 Mbit.

```
# tc qdisc add dev eth0 root handle 1:0 cbq bandwidth 100Mbit         \
  avpkt 1000 cell 8
# tc class add dev eth0 parent 1:0 classid 1:1 cbq bandwidth 100Mbit  \
  rate 6Mbit weight 0.6Mbit prio 8 allot 1514 cell 8 maxburst 20      \
  avpkt 1000 bounded
```

Ez a rész létrehozza a gyökeret és a szokásos 1:1-es osztályt. Ez az osztály kötött (bounded), tehát nem kérhet kölcsön, így a teljes sávszélessége nem haladhatja meg a 6 Mbitet.
A CBQ-nak sok csomópontra van szüksége. A hasonló HTB (később) lényegesen kevesebb beállítást igényel.

```
# tc class add dev eth0 parent 1:1 classid 1:3 cbq bandwidth 100Mbit  \
  rate 5Mbit weight 0.5Mbit prio 5 allot 1514 cell 8 maxburst 20      \
  avpkt 1000
# tc class add dev eth0 parent 1:1 classid 1:4 cbq bandwidth 100Mbit  \
```

```
  rate 3Mbit weight 0.3Mbit prio 5 allot 1514 cell 8 maxburst 20      \
  avpkt 1000
```

Ez a két levél osztály. Láthatjuk, hogy hogyan határozzuk meg a weight (súly) paramétert a rate függvényében. Egyik osztály sem kötött, azaz nincs definiálva a bounded paraméter, de egy olyan osztálynak a gyerekeik, amely kötött. Tehát a két osztály együttes sávszélessége nem haladja meg a szülőét, ami jelen esetben 6 Mbit. Mellesleg, a fő számuknak meg kell, hogy egyezzen a szülő osztály fő számával.

```
# tc qdisc add dev eth0 parent 1:3 handle 30: sfq
# tc qdisc add dev eth0 parent 1:4 handle 40: sfq
```

Mindkét osztály alapesetben a FIFO módszert kapta meg, de ezt mi kicseréltük egy-egy SFQ-ra. Így igazságosabbak lehetünk az adatfolyamokkal.

```
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip \
  sport 80 0xffff flowid 1:3
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip \
  sport 25 0xffff flowid 1:4
```

Ezek a parancsok közvetlenül a gyökér módszerhez csatlakoznak, és a forgalmat a megfelelő osztályhoz küldik el.
Vegyük észre, hogy a „tc class add” utasítást használjuk egy osztály létrehozásához egy módszeren belül, és a „tc qdisc add”-ot, hogy ezekhez az osztályhoz módszereket adjunk.
Talán érdekes lehet mi történik azokkal a csomagokkal, melyek egyik szabályra sem illeszkednek. Ezek az adatok, jelen esetben korlátlanul továbbhaladhatnak az 1:0-as módszer által.
Ha az SMTP és a web együttes sebessége eléri a 6 Mbit/másodperces határt, akkor a sávszélesség a súly paraméter által meghatározott arányban fog szétosztódni, most éppen 5/8-ada a forgalomnak a webszerverhez jut és 3/8-ada a levelező szervernek.

2.3.4.5. Egyéb CBQ paraméterek

Korábban már szó volt róla, hogy az osztályos sorbaállítási módszereknek szükségük van

szűrőkre, hogy meg tudják határozni, egy csomag melyik osztályba kerüljön besorolásra. A szűrő meghívása mellett a CBQ más lehetőségeket is kínál: defmap és split. Ez a rész meglehetősen bonyolult és nem is életbevágóan fontos, de most itt a helye, hogy néhány szót ejtsünk róluk.

Ha leginkább a TOS mező alapján szeretnénk szűrni, akkor van egy különleges lehetőségünk is. Amikor a CBQ kiszámolja, hogy egy csomagot hova kell besorolni, akkor leellenőrzi, hogy az a csomópont „split csomópont" e. Ha igen, akkor az egyik almódszer meg van jelölve, hogy ő szeretné fogadni az összes olyan csomagot, mely egy bizonyos prioritással rendelkezik, ami meghatározható a TOS mezőből, melyet vagy a socket tulajdonságokból határoztak meg, vagy az alkalmazások állítottak be.

A defmap használata egy remek lehetőség arra, hogy gyors szűrőt csináljunk, ami csak bizonyos prioritásokat enged át. Ha a defmap ff, akkor minden illeszkedni fog rá, ha 0, akkor semmi.

Példa:

```
# tc qdisc add dev eth1 root handle 1: cbq bandwidth 10Mbit allot 1514 \
  cell 8 avpkt 1000 mpu 64

# tc class add dev eth1 parent 1:0 classid 1:1 cbq bandwidth 10Mbit    \
  rate 10Mbit allot 1514 cell 8 weight 1Mbit prio 8 maxburst 20        \
  avpkt 1000
```

A defmap a TC_PRIO bitekkel azonos, nézzük ezeket:

```
TC_PRIO..          Szám TOS-nak megfelelő jelentés
-------------------------------------------------
BESTEFFORT         0    Maximális megbízhatóság
FILLER             1    Minimális költség
BULK               2    Maximális áteresztés (0x8)
INTERACTIVE_BULK   4
INTERACTIVE        6    Minimális várakozás (0x10)
CONTROL            7
```

A TC_PRIO.. szám olyan biteknek felel meg, melyek jobbról vannak számozva. A pfifo_fast résznél bővebben le van írva, hogyan kell a TOS biteket átalakítani prioritásokká.
Most nézzük az interaktív és a nagy adatmennyiségű osztályokat:

```
# tc class add dev eth1 parent 1:1 classid 1:2 cbq bandwidth 10Mbit    \
  rate 1Mbit allot 1514 cell 8 weight 100Kbit prio 3 maxburst 20       \
  avpkt 1000 split 1:0 defmap c0

# tc class add dev eth1 parent 1:1 classid 1:3 cbq bandwidth 10Mbit    \
  rate 8Mbit allot 1514 cell 8 weight 800Kbit prio 7 maxburst 20       \
  avpkt 1000 split 1:0 defmap 3f
```

A split módszer az 1:0, itt történik meg a választás. C0 binárisan 11000000, 3F binárisan 0011111111, ez a kettő mindenre illeszkedni fog. Az első osztály a 7-es és 6-os bitek illeszkedését nézi, ami az interaktív és irányító forgalom. A második osztály a maradék.
Az 1:0-as csomópontnál a következő táblázat alapján dönt:

prioritás	küldés helye
0	1:3
1	1:3
2	1:3
3	1:3
4	1:3
5	1:3
6	1:2
7	1:2

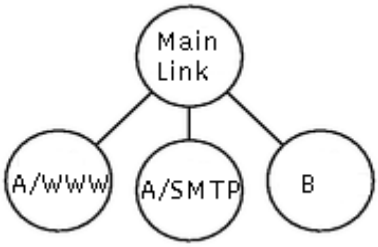
2.3.5. A hierarchikus vödör algoritmus (Hierarchical Token Bucket – HTB)

Martin Devera, a HTB megalkotója, rájött, hogy a CBQ eléggé összetett és úgy tűnik nincs optimalizálva néhány gyakori esetre. Az Ő hierarchikus megközelítési módja nagyon jól alkalmas akkor, amikor egy meghatározott mennyiségű sávszélességünk van és ezt szeretnénk különböző célokra felhasználni, azaz felosztani azt úgy, hogy minden egyes célra garantált sávszélesség jusson olyan lehetőségekkel, mint a sávszélesség kölcsönadás.
A HTB (első ránézésre) úgy működik, mint a CBQ, de vele ellentétben nem használja a szünetidőt a formálás kiszámolásához. Mint ahogyan a neve is mutatja a HTB inkább egy

osztályos vödrös algoritmus. Csak néhány paraméterrel rendelkezik.
Ahogyan egyre összetettebb HTB konfigurációt állítunk össze úgy lesz egyre jobban skálázható. CBQ-val még az egyszerűbb esetek is összetett beállításokat igényelnek néha. A HTB a 2.4.20-as verziójú Linux kernel óta hivatalosan is része a kernelnek. De ahhoz, hogy tudjuk használni a tc programot meg kell patch-elni, vagy megszerezhetjük a binárisokat a HTB web oldaláról (http://luxik.cdi.cz/~devik/qos/htb/). A HTB kernel- és userspace részének a fő verziószáma meg kell, hogy egyezzen, különben a HTB nem fog működni.
A HTB egy érthetőbb, ösztönösebb és gyorsabb alternatíva a CBQ módszerre Linuxban. Mind a CBQ mind a HTB segít a kimenő forgalom szabályozásában egy adott kapcsolaton. Mindkettő lehetőséget biztosít arra, hogy egy fizikai kapcsolaton több különböző lassabb kapcsolatot szimuláljunk és különböző típusú forgalmakat más-más kapcsolaton küldjünk. Mindkét esetben mi határozzuk meg, hogy a fizikai kapcsolatot, hogyan osztjuk fel, és hogy mi alapján döntjük el egy adott csomagról, hogy az melyik szimulált kapcsolaton menjen ki.

2.3.5.1. A kapcsolat megosztás

Most felvázolok egy problémát, amit majd HTB-vel oldunk meg. Van két ügyfél A és B, mindkettő az eth0-n át csatlakoznak az Internetre. B-nek 60 kbps-ot A-nak 40 kbps-ot szeretnénk biztosítani. Ezután „A” sávszélességét két részre osztjuk, a WWW-nek 30 kbps-t akarunk adni és a maradék 10 kbps-t a többi forgalomnak. A nem használt sávszélességet bármelyik osztály használhatja, amelyiknek szüksége van rá (a meghatározott megosztás arányában).
A HTB által biztosított sávszélesség mennyiség minden egyes osztályhoz legalább a minimuma az osztály által igényelt vagy a hozzá társított sávszélességnek. Amikor egy osztály kevesebb sávszélességet igényel, mint amennyi neki van szánva, akkor a fennmaradó (többlet) sávszélesség szétosztódik más osztályok között, amelyek igényt tartanak erre.
Az irodalomban ezt a többlet sávszélesség „borrowing”-nak (kölcsönvételnek) hívják. Továbbá ezt használjuk az irodalomnak megfelelően. Azért megjegyezzük, hogy bizonyos szempontból ez a terminológia nem helyes, mert itt nincs arról szó, hogy amit egyszer „kölcsönadtunk” azt vissza kellene fizetni.
A HTB-ben a különböző forgalmak osztályként jelennek meg. Az egyik legegyszerűbb utat az alábbi ábrán láthatjuk.



Nézzük milyen parancsok szükségesek ehhez:

```
tc qdisc add dev eth0 root handle 1: htb default 12
```

Ez a parancs egy HTB sorbaállítási módszert rendel az eth0 eszközhöz gyökeréhez, amelynek a nyele (handle) megkapja az 1:-es azonosítót. A „default 12” paraméter azt jelenti, hogy azok a csomagok, melyek nincsenek beosztályozva alapból a 1:12-es osztályba kerülnek.

```
tc class add dev eth0 parent 1: classid 1:1 htb rate 100kbps ceil 100kbps
tc class add dev eth0 parent 1:1 classid 1:10 htb rate 30kbps ceil
100kbps
tc class add dev eth0 parent 1:1 classid 1:11 htb rate 10kbps ceil
100kbps
tc class add dev eth0 parent 1:1 classid 1:12 htb rate 60kbps ceil
100kbps
```

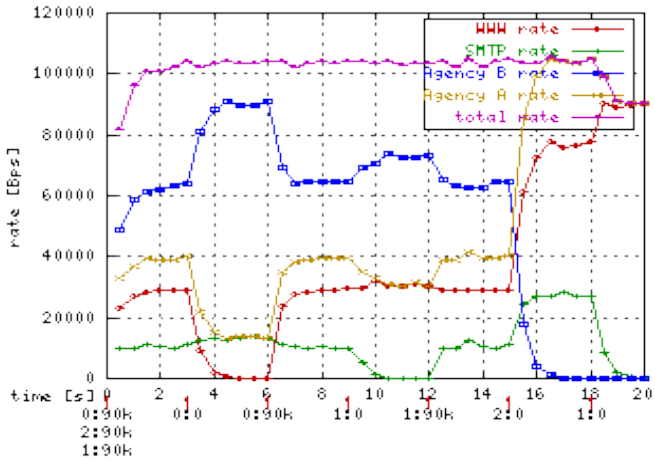
Az első sor létrehoz egy 1:1 azonosítójú gyökér osztályt közvetlenül az 1:-es módszer alá. A gyökér osztály egy olyan osztály, melynek a szülője egy módszer. A gyökér osztály megengedi a gyerekeinek, hogy azok egymásnak kölcsönadjanak sávszélességet, de egy gyökér osztály nem tud kölcsönkérni semelyik másik osztálytól sem. A következő sorokban létrehozzuk a három osztályt közvetlenül a gyökér alá úgy, hogy ha van felesleges sávszélesség, akkor az kölcsön lehessen kérni. A ceil paramétert később magyarázom el.
Fontos megjegyezni, hogy a dev <eszköz> paramétert mindig ki kell írni és nem elég csak a nyél azonosítóra és a szülőre hivatkozni, mert ezek az értékek az eszközhöz képest lokálisak.
Ezek után még azt is el kell dönteni melyik csomag melyik osztályba sorolódjon be. A szűrőkről részletesebben később lesz szó. Az idetartozó parancs így néz ki:

```
tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32 \
   match ip src 1.2.3.4 match ip dport 80 0xffff flowid 1:10
tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32 \
   match ip src 1.2.3.4 flowid 1:11
```

Észrevehetjük, hogy a 1:12-es osztályhoz nem lett szűrő létrehozva, azért mert a gyökér módszernél már megadtuk a default paraméterrel, hogy az egyik szűrőre sem illeszkedő csomagok ide kerüljenek.
Mint korábban említettem, a levél osztályokhoz megadható egy újabb módszer, ha ezt nem tesszük a default pfifo rendelődik hozzájuk.

```
tc qdisc add dev eth0 parent 1:10 handle 20: pfifo limit 5
tc qdisc add dev eth0 parent 1:11 handle 30: pfifo limit 5
tc qdisc add dev eth0 parent 1:12 handle 40: sfq perturb 10
```

Ez az összes parancs, ami szükséges a probléma megoldásához. Most nézzük mi történik ha forgalmat eresztünk a kapcsolatra.
Először mindegyik osztályok 90 kbps sebességű forgalmat próbálunk meg lebonyolítani, majd az egyik osztályon megszűnik a forgalomküldés. Az ábra alján láthatók a bejelentések, azaz melyik osztályhoz tartozó forgalom változott meg éppen: <osztály>:<új sebesség>. Ebből látszik, hogy a nulladik időpillanatban, mindhátom osztály 0 (A WWW része, 1:10), 1 (A SMTP része, 1:11) és 2 (B, 1:12) 90 kbps, a 0 osztály 3-ik időpilanatban 0, majd a 6-ikban megint 90 kbps.



Tehát kezdetben mindegyik osztályra 90 kbps sebességű forgalmat küldünk, mivel ez az összes osztálynál nagyobb, mint az osztályhoz rendelt maximális sebesség, ezért mindegyik beáll az előre meghatározottra. A 3-ik időpillanatban, amikor már nem küldünk tovább csomagot a 0-as osztályra, a sebesség szétosztás újra megtörténik a maradék két osztály között, de csak az újonnan felszabadult erőforrások lesznek szétosztva, mégpedig a parancsokban meghatározott sebességek arányában. Tehát egy rész jut az 1-es osztálynak, és hat rész a 2-esnek. Azaz például az 1-es osztályt nézve ez azt jelenti, hogy az

<új sebesség> = <régi sebesség> + <0-as osztály felszabadult része>/7.

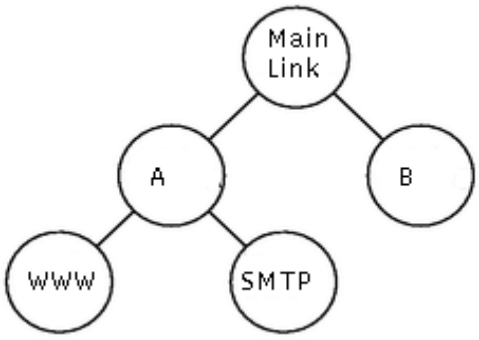
Így ez az 1-es osztálynak jelentős növekedés (40%). Hasonló ehhez a 9-es időpontnál történő eset, amikor az 1-es osztály szűnik meg, és az ő sávszélessége osztódik szét. A 15-ös időpillanatban egyszerű látni, hogy a 2-es osztály megszűnésével a sávszélességéből három rész a 0-s osztálynak jut míg egy rész az 1-esnek. A 18-as időben az 1-es osztály megkapja mind a 90 kbps igényelt sebességet.

Ez egy jó alkalom arra, hogy említést tegyünk a quantum fogalmáról. Valójában amikor több osztály is kölcsön akar kérni kapnak néhány byte-ot mielőtt kiszolgálnák a többi versenyző osztályt. Ez a byte-mennyiség a quantum. Láthatjuk, ha néhányan versenyeznek a szülő osztály sávszélességéért, akkor ezt a quantumok arányában kapják meg. Fontos, hogy ez a szám lehető legkisebb legyen, de még nagyobb mint az MTU értéke. Általános esetben nincs szükség a quantum paraméter megadására, mert a HTB egy kiszámolt értéket fog választani, mikor létrehozzuk az osztály, és akkor is amikor módosítjuk azt. A számítás: a sebesség osztva az r2q globális paraméterrel. Ennek az alapértéke 10. Ez 1500-as MTU mellett 15 kBbs (120 kbit) sebességnél optimális.

Ez a megoldás jó lehet akkor, amikor A és B nem különböző ügyfelek. Mert például, ha A fizeti az egész 40 kbps-ot, akkor a nem-használt WWW sávját szeretné a másik saját szolgáltatásának adni, nem pedig B-nek.

2.3.5.2. A hierarchia megosztás

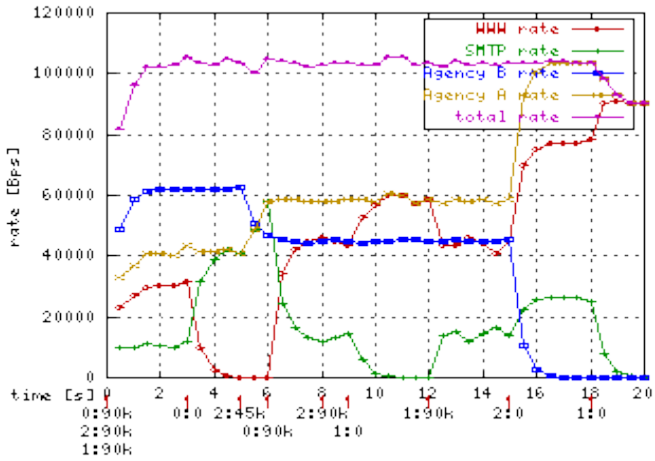
Az előző fejezetben felvetett probléma megoldása látható az alábbi ábrán.



Most az A ügyfélnek is megvan teljesen a saját osztálya. Itt is érvényes a fentiekben említett szabály: a HTB által biztosított sávszélesség mennyiség minden egyes osztályhoz legalább a minimuma az osztály által igényelt vagy a hozzá társított sávszélességnek. Ez jelen esetben csak azokra az osztályokra érvényes melyek nem szülői más osztályoknak, azaz levél osztályok. Azokra az osztályokra melyek szülői másoknak (belső osztályok) a következő szabály érvényes: a biztosított szolgáltatás legalább a minimuma a hozzátársítottnak vagy a gyerekei által követeltnek. Ebben az esetben mi az A osztályhoz 40 kbps-ot rendeltünk hozzá, ami azt jelenti, hogy ha A WWW forgalma nem éri el a hozzárendelt értéket, akkor fennmaradó általa nem használt sávszélesség az A többi szolgáltatása között fog feloszlani, egészen addig amíg az összes el nem éri a 40 kbps-ot.
Legtöbbször a szülő sávszélességét a gyerekeihez társítottak összegeként szokás meghatározni.
Hogy az ábrához megfelelő beállításokhoz jussunk a következő parancsokat kell kiadni:

```
tc class add dev eth0 parent 1: classid 1:1 htb rate 100kbps ceil 100kbps
tc class add dev eth0 parent 1:1 classid 1:2 htb rate 40kbps ceil 100kbps
tc class add dev eth0 parent 1:2 classid 1:10 htb rate 30kbps ceil
100kbps
tc class add dev eth0 parent 1:2 classid 1:11 htb rate 10kbps ceil
100kbps
tc class add dev eth0 parent 1:1 classid 1:12 htb rate 60kbps ceil
100kbps
```

Nézzük, hogy milyen grafikon is tartozik ehhez a megoldáshoz, ami jól szemlélteti az előnyét:



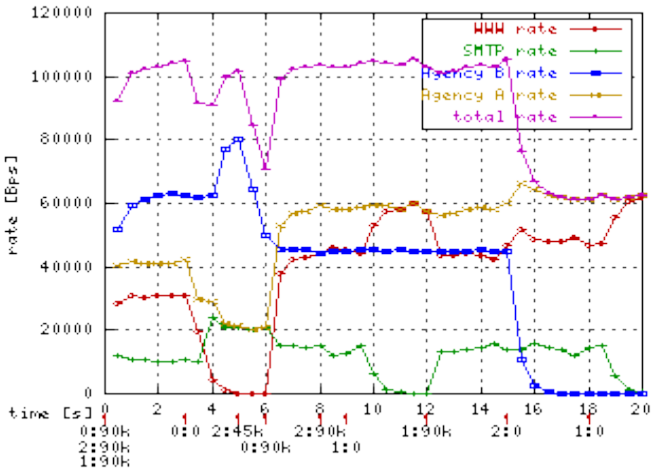
Láthatjuk, hogy amikor A WWW forgalma megáll a 3-as időpillanatban, akkor a hozzárendelt sávszélesség A maradék szolgáltatásai között fog eloszlani, azaz jelen esetben A egész sávszélességét az SMTP forgalom teheti ki. Azt is láthatjuk, hogy amikor A gyerekei nem használják ki A teljes 40 kbps-ját, akkor a felesleget B fel tudja használni.

2.3.5.3. A sávszélesség csúcsa (rate ceiling)

Ebben a fejezetben lesz szó a ceil paraméter használatáról. A ceil paraméterben határozhatjuk meg, hogy egy osztály maximálisan mennyi sávszélességet használhat. Ez fogja behatárolni, hogy egy osztály mennyit kérhet kölcsön. Alapértelmezett értéke megegyezik a rate paraméterben megadott értékkel. (A példákban ezért is kellett használni ezt a paramétert, hogy működjön a kölcsönvétel.)

Most változtassuk meg az előző példában használt 100 kbps-os ceil értéket az 1:2-es osztály számára 60 kbps-ra és a 1:11-es osztály számára 20 kbps-ra.

Az új grafikonon az előzőhöz képest több változást is láthatunk. Például a 3-as időpillanatban, amikor a WWW forgalom megáll, akkor A összesen csak 20 kbps-ot használ a 1:11 révén aminek az új ceil értéke 20 kbps. A maradék 20 kbps-ot természetesen így B kaphatja meg. A másik különbség a 15-os időpillanatban van, amikor B megállítja az adatforgalmát. A régi ceil értékkel A-hoz kerülhetne az összes sávszélesség, de most csak 60 kbps-ig mehet fel, a maradék 40 kbps kihasználatlan marad.



Ez hasznos lehet Internet szolgáltatóknál, mert ők akkor is korlátozni szeretnék az egyes ügyfeleiket, amikor mások nem használják a szolgáltatást.

Fontos megjegyezni, hogy a gyökérosztály nem kérhet kölcsön senkitől, így nincs értelme ceil paramétert beállítani ehhez az osztályhoz. Az is kiderült, hogy a ceil értéke legalább a rate értékének kell lennie. Sőt, az osztály ceil értéke jó, hogyha legalább akkora, mint bármelyik gyerekéé.

2.3.5.4. A burst paraméter

Az egyes hálózati eszközök egy időben csak egy csomagot tudnak elküldeni, és ezt is csak egy hardver-függő mértékben. A kapcsolat megosztó szoftver ezt használja fel, hogy megbecsülje a többszörös kapcsolat különböző sebességeit. Ebből kifolyólag a ceil és rate értékek nem tekinthetőek olyan értékeknek, melyek azonnal reagálnának a hálózati forgalom hirtelen változásaira, de átlagot tudnak mondani több időn át, hogy mennyi csomagot lehet küldeni. Ami valójában történik és a probléma az az, hogy az egyik osztálytól jövő forgalom néhány csomag erejéig maximális sebességű, és ezután más osztályok egy kis időre felfüggesztődnek. A burst és cburst paraméterek felügyelik azt, hogy mennyi adatot lehet küldeni maximális hardver sebességen, anélkül, hogy ez a fennakadás megtörténne más osztályoknál.

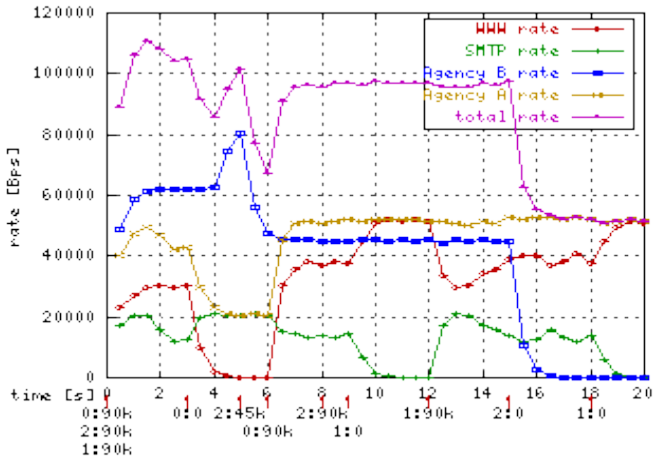
Ha a cburst érték túl kicsi (legjobb az egy csomag méretnyi), akkor ez úgy alakítja a tüskéket, hogy azok nem érik el a ceil értéket, hasonlóan, mint a vezérjeles vödrös algoritmus esetén a peakrate érték.

Ha a burst paraméter értékét úgy állítjuk be egy szülő osztálynak, hogy az kisebb, mint valamelyik gyerekéé akkor várhatjuk, hogy a szülő néha beragad, mert a gyerek több csomagot küld, mint amennyit a szülő képes lekezelni. A HTB képes visszaemlékezni ezekre a beragadt csomagokra, azaz a negatív csúcsokra legfeljebb egy percig.

Kérdezhetnénk mi az értelme ennek a paraméternek? Ez egy olcsó megoldás arra, amikor csökkenteni akarjuk a válaszidőt egy túlterhelt és torlódásokkal rendelkező kapcsolat esetén. A www forgalom ilyen csúcsos természetű, tehát ha jön egy kérés az oldal fele, akkor arra egyből jön a válasz (ez a csúcs), majd olvassuk azt. Amíg nincs kérés addig a burst „újratöltődik".

Láthattuk, hogy a burst és cburst értékeket legalább akkorának kell választani, mint a bármelyik gyerek osztálynál.

Nézzük a grafikont, amikor is megváltozott a WWW és SMTP forgalom burst értéke 20 kb-re, míg a cburst ugyanaz maradt (2 kb alapértelmezésben).



A 13-as időpillanatban a zöld domb az új burst értéknek köszönhető az SMTP osztálynál. Mivel ez a forgalom a 9-es időtől nem volt kihasználva, így a burst feltöltődött 20 kb-re. A domb csak 20kbps-ig mehet fel, mert a ceil érték behatárolja (és a cburst csomagméret-közeli).

Ki lehet találni, hogy a 7-es időpillanatban miért nincs piros és sárga domb. Azért nincs, mert ebben a pillanatban a sárga már a ceil határon van, így nincs hely a burstnek.

Észrevehetünk egy nem kívánt hatást, mégpedig a 4-es időpillanatban a lilának vagy egy bemélyedése. Ez azért van, mert az 1:1-es gyökér osztálynak nem adtunk burst értéket. Ez egy „visszaemlékezett" domb még az 1-es időből, és amikor a 4-esben a kék kölcsön akarta

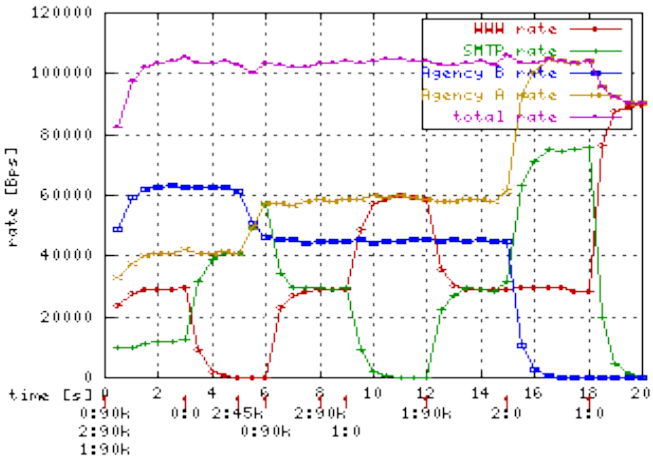
kérni a sárga felszabadult sávszélességét, akkor a kérés még nem lett átadva. De végül saját maga kompenzálta azt.

Az időosztásból adódhatnak hibák, amikor magas sebességekkel dolgozunk alacsony időzítés mellett. Ekkor az összes osztály számára alacsony burst és cburst értéket kell meghatározni. Például egy i386 alapú rendszer esetég az időzítő 10 milimásodperces, míg Alpha rendszeren 1. A minimális burstöt a max_rate és a az időzítő felbontásának a szorzata határozza meg. Tehát 10 Mbit és egy egyszerű i386 esetén 12 kb-es burst érték szükséges.

Ha túl alacsony burstöt határozunk meg, akkor alacsonyabb sávszélességet tapasztalhatunk, mint amit eredetileg beállítottunk. A legutóbbi tc program már kiszámítja a legkisebb burst-t, ha nem adtunk meg semmit.

2.3.5.5. A sávszélesség megosztás priorizálása

A forgalom priorizálásának két oldala van. Az első arra hat, hogy hogyan osztódjon szét a felesleges sávszélesség a testvér osztályok között. Eddig csak annyit láttunk, hogy a rate-k arányában osztódott szét a felesleg. Most a „hierarchia megosztás” fejezetben használt beállításokból indulunk ki. Megváltoztatjuk az összes osztály prioritását 1-re, de az SMTP-s (zöld) osztályét 0-ra, azaz magasabb prioritásra.



Láthatjuk, hogy a jobban priorizált osztály megkapja az egész sávszélességet, ami feleslegessé vált. A szabály az, hogy a magasabb prioritású osztálynak lesz felajánlva először a felesleges sávszélesség. De a garantált rate és ceil szabályok ugyanúgy érvényesek.

Például a 3-as időpillanatban, amikor A WWW forgalma megszűnik, a belőle felszabadult erőforrásokat teljes egészében A SMTP forgalma kapja meg. De amikor 6-nál visszajön a WWW forgalom, akkor A összes sávszélességén kell kettejüknek osztozkodni.
A másik oldal a csomagok késleltetése. Eléggé nehéz megbecsülni egy olyan ethernet kapcsolaton, ami kellően gyors. De van egy egyszerű megoldás. Hozzáadunk egy egyszerű HTB-t egyetlen osztállyal, melynek a rate-je alacsonyabb, mint a kapcsolat sebessége. Ezután a dolgok hasonlóképpen mennek, mint a régebbi beállításoknál. Tehát hozzáadunk egy módszert, egy főosztállyal, majd ehhez a gyerekeket. Ekkor egy lassabb kapcsolatot szimulálunk, nagyobb várakozásokkal.
Lássuk egy példán:

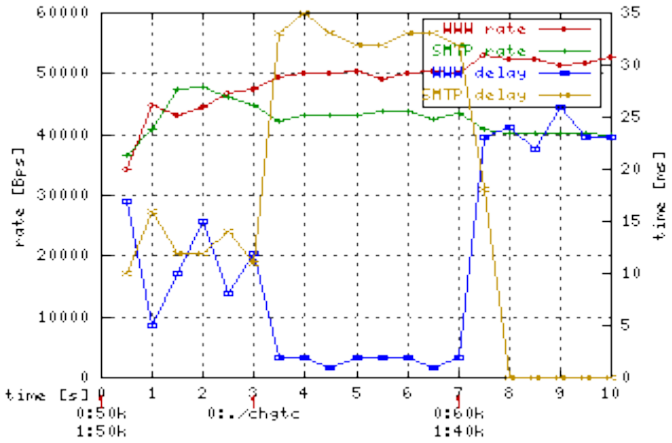
```
# qdisc for delay simulation
tc qdisc add dev eth0 root handle 100: htb
tc class add dev eth0 parent 100: classid 100:1 htb rate 90kbps

# real measured qdisc
tc qdisc add dev eth0 parent 100:1 handle 1: htb
tc class add dev eth0 parent 1: classid 1:2 htb rate 100kbps

tc class add dev eth0 parent 1:2 classid 1:10 htb rate 50kbps \
   ceil 100kbps prio 1
tc class add dev eth0 parent 1:2 classid 1:11 htb rate 50kbps \
   ceil 100kbps prio 1
tc qdisc add dev eth0 parent 1:10 handle 20: pfifo limit 2
tc qdisc add dev eth0 parent 1:11 handle 21: pfifo limit 2
```

Megjegyzendő, hogy egy HTB módszer egy másik HTB módszer alá helyezése nem ugyanaz, mint egymás alatt levő HTB osztályok ugyanazon a módszeren belül. Mert egy HTB osztály amikor tud küldeni, akkor az olyan hamar el lesz küldve, ahogyan a hardver tudja. Tehát ez nem lesz az ősök által korlátozva, csak a felszerelés által. Viszont a HTB a HTB-ben esetnél a külső módszer egy új hardver eszközt szimulál a belső HTB-nek.
Először mindkét osztályon 50 kbps forgalmat eresztünk át. A 3-as időpillanatban megváltoztatjuk a 1:10-es olsztály prioritását:

```
tc class change dev eth0 parent 1:2 classid 1:10 htb \
 rate 50kbps ceil 100kbps burst 2k prio 0
```



A hatást nézzük meg az új grafikonon. Itt már szerepelnek a várakozási értékek is.

Mint láthatjuk a WWW várakozási értéke nulla közeli értékre esett vissza, míg az SMTP várakozása nagyon megnőtt. Ez általános, ha az egyik osztály várakozását csökkentjük priorizálással, akkor más osztályok várakozása romlik. Később a 7-es időben a megnöveljük a WWW forgalmat 60 kbps-ra és az SMTP lecsökkentjük 40 kbps-ra. Ekkor egy újabb érdekes jelenséget tapasztalhatunk, mégpedig azt, hogy a túl terhelt osztállyal (WWW) szemben a HTB a nem túlterhelt osztálynak (SMTP) kedvez. Most konkrétan a nem túlterhelt SMTP várakozása csökken, míg a túlterhelt WWW várakozása megnőtt.
Mely osztályokon ajánlatos használni a priorizálást? Erre ott lehet igazán szükség, ahol alacsony várakozás szükséges. Például videó és audió streamek esetén, ahol a megfelelő rate kell annak megelőzése érdekében, hogy ez a forgalom megölje a többit. Vagy az interaktív forgalmaknál (telnet, ssh), ahol a forgalom ugrás szerű egy kis időre, ami nincs negatív kihatással a többi forgalomra. Egy népszerű „trükk”, hogy nagyobb prioritást adnak az ICMP csomagoknak, így telített kapcsolat esetén is alacsony lehet a ping idő, de technikai szempontból nem ez a cél.

2.3.5.6. A statisztika elemzése

A tc program lehetőséget ad arra, hogy statisztikai adatokat tudjunk meg az alkalmazott sorbaállítási módszerekről. Most értelmezzük ezeket a statisztikai adatokat. A jelenlegi adatok a „kapcsolat megosztás” fejezet beállításai alapján jelennek meg.

```
# tc -s -d qdisc show dev eth0
 qdisc pfifo 22: limit 5p
 Sent 0 bytes 0 pkts (dropped 0, overlimits 0)

 qdisc pfifo 21: limit 5p
 Sent 2891500 bytes 5783 pkts (dropped 820, overlimits 0)

 qdisc pfifo 20: limit 5p
 Sent 1760000 bytes 3520 pkts (dropped 3320, overlimits 0)

 qdisc htb 1: r2q 10 default 1 direct_packets_stat 0
 Sent 4651500 bytes 9303 pkts (dropped 4140, overlimits 34251)
```

Az első három sorbaállítási módszer a HTB gyerekei, ezek a PFIFO sorok magukat magyarázzák. Az overlimits paraméter azt írja le, hogy a módszer hányszor várakoztatott meg csomagokat. A direct_packets_stat a soron közvetlenül áthaladó csomagok számát adja meg.
Most nézzük az egyes osztályokhoz tartozó statisztikát:

```
tc -s -d class show dev eth0
class htb 1:1 root prio 0 rate 800Kbit ceil 800Kbit burst 2Kb/8 mpu 0b
    cburst 2Kb/8 mpu 0b quantum 10240 level 3
 Sent 5914000 bytes 11828 pkts (dropped 0, overlimits 0)
 rate 70196bps 141pps
 lended: 6872 borrowed: 0 giants: 0

class htb 1:2 parent 1:1 prio 0 rate 320Kbit ceil 4000Kbit burst 2Kb/8
mpu 0b
    cburst 2Kb/8 mpu 0b quantum 4096 level 2
 Sent 5914000 bytes 11828 pkts (dropped 0, overlimits 0)
 rate 70196bps 141pps
 lended: 1017 borrowed: 6872 giants: 0

class htb 1:10 parent 1:2 leaf 20: prio 1 rate 224Kbit ceil 800Kbit burst
2Kb/8 mpu 0b
    cburst 2Kb/8 mpu 0b quantum 2867 level 0
```

```
Sent 2269000 bytes 4538 pkts (dropped 4400, overlimits 36358)
rate 14635bps 29pps
lended: 2939 borrowed: 1599 giants: 0
```

Az utolsó két osztály statisztikája nincs itt (1:11 és 1:12). Egy osztályra vonatkozó statisztikában láthatjuk, hogy az adott osztályt milyen paraméterekkel hoztuk létre. Az overlimits paraméter az osztályoknál mást jelent, mint a soroknál. Itt azt adja meg, hogy hányszor történt olyan eset, amikor az osztályt felszólították csomag küldésre, de az a rate és ceil megszorítások miatt nem tudott küldeni.

A rate és pps (packets per secundum) paraméterek az elmúlt 10 másodpercbeli átlagsebességet adják meg byte-ban illetve csomagszámban. Ezt az értéket a HTB is felhasználja egyéb számításokhoz.

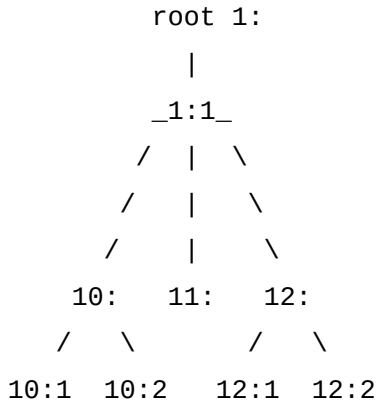
A lended azt adja meg, hogy egy osztály mennyi csomagot adott kölcsön más osztálynak az a ő sávszélességéből. A borrowed azon csomagok száma melyek kölcsönkért sávszélességből mentek át az osztályon. A lended kiszámítása mindig osztály szinten lokálisan történik, míg a borrowed paraméter tranzitív. Tehát ha a 1:10-es osztály kölcsönkér, akkor a 1:2 is kölcsönkér az 1:1-es osztálytól, így egy menetben mind a 1:10 és 1:2 osztályok borrowed paramétere nő.

A giants érték azoknak a csomagoknak a száma melyeknek a mérete meghaladta a tc által beállított mtu értéket.

3. Csomagok osztályozása szűrőkkel

3.1. Általában a szűrőkről

Talán már az eddigiekből is kiderülhetett, hogy a szűrők fontos szerepet játszanak az osztályos sorbaállítási módszereknél. A sok-sok csomagnál el kell dönteni, hogy melyik osztály szabályai legyenek érvényesek rájuk, azaz a csomag ezen módszerben való tartózkodása során milyen osztály-láncot járjon be. Látható, hogy a szűrőket csak osztályos sorokhoz tudunk hozzárendelni, a szűrőknek is ugyanúgy van szülője, mint az osztályoknak és ezeknek a módszereinek is. Egy szűrő szülője csak egy osztályos sorbaállítási módszer (handle-je) lehet. A további magyarázathoz tekintsük a következő ábrát:



Amikor egy csomag besorolásra kerül, azaz bekerül az osztályos módszerbe, akkor minden elágazásnál a szűrő-láncon elindul, hogy a további útjáról információt szerezzen. Egy tipikus beállításnál a 1:1-nél egy szűrőnek kell lennie, ami a csomagot elirányítja mondjuk 12:-be, ahol szintén vagy egy szűrő, és ez elirányítja a csomagot 12:2-be. De előfordulhat az is, hogy 12:-höz nincsen hozzárendelve szűrő, hanem az összes szűrő 1:1-hez van rendelve, és a csomagok innen leszek elirányítva a levelekbe közvetlenül. De pontosabb szűrést tesz lehetővé, ha a specifikus teszteket alacsonyabb szintre helyezzük.
Egy csomagot nem tudunk „felfele” szűrni, csak lefele lehetséges a besorolásuk. Mikor ki akarnak kerülni a sorból, akkor kerülnek egyre feljebb, ahol végül az interfész található.

3.1.1. Néhány egyszerű szűrő példa

Mint ahogyan azt már az osztályozást tárgyaló fejezetben is láttuk, a csomagok illeszkedését szinte bármi alapján lehet vizsgálni, kellően bonyolult kifejezésekkel. Indulásképpen nézzük nyilvánvaló dolgokat, hogyan kell megvalósítani.
Tehát legyen egy PRIO módszerünk, amit 10:-nek nevezünk el, ez három osztályt tartalmaz. Az szeretnénk, hogy a 22-es portra és a 22-es portról érkező csomagok (ssh) a legnagyobb prioritással rendelkező kötegbe kerüljenek. Ehhez a következő szűrők kellenek:

```
# tc filter add dev eth0 protocol ip parent 10: prio 1 u32 match \
  ip dport 22 0xffff flowid 10:1
# tc filter add dev eth0 protocol ip parent 10: prio 1 u32 match \
  ip sport 22 0xffff flowid 10:1
# tc filter add dev eth0 protocol ip parent 10: prio 2 flowid 10:2
```

Elemezzük ezt a néhány sort. Az eht0 eszköz 10:-es csomópontjához egy olyan szűrőt rendeltünk, ami 1-es prioritású u32-es szűrő és a 22-es portra menő csomagok illeszkednek rá. A következő egy ugyanilyen szűrő csak a 22-es célporttal rendelkező csomagok illeszkednek rá. Ezeket a csomagokat a 10:1-es osztály fele irányítja a szűrő a flowid paraméter értéke alapján. Az utolsó sor azt határozza meg, hogyha eddig nem történt illeszkedés, akkor a csomagok alacsonyabb prioritással a 10:2-es osztályba menjenek.
Minden eszközhöz (most eth0) saját szűrőket rendelhetünk, mert minden eszköznek az azonosítók tekintetében (handle) saját névtere van.
Nézzünk egy hasonló példát, ahol a csomagok illeszkedését IP cím alapján végezzük:

```
# tc filter add dev eth0 parent 10:0 protocol ip prio 1 u32 \
  match ip dst 4.3.2.1/32 flowid 10:1
# tc filter add dev eth0 parent 10:0 protocol ip prio 1 u32 \
  match ip src 1.2.3.4/32 flowid 10:1
# tc filter add dev eth0 protocol ip parent 10: prio 2      \
  flowid 10:2
```

Itt azok a forgalmak kerülnek nagyobb prioritású osztályba, melyek a 4.3.2.1-es IP-re mennek, illetve azok melyek a 1.2.3.4-ról jönnek, a maradék mind alacsonyabb prioritású

lesz. Ezeket a port/IP szűrő feltételeket egy szűrőben is össze lehet fűzni. Például 1.2.3.4-es IP-nek a 80-as portjáról jövő forgalmak illeszkedése:

```
# tc filter add dev eth0 parent 10:0 protocol ip prio 1 u32 \
  match ip src 4.3.2.1/32 match ip sport 80 0xffff flowid 10:1
```

3.1.2. Általános szűrő parancsok

A legtöbb szűrő parancs ugyanúgy kezdődik:

```
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32
```

Ezek az u32-nek nevezett illeszkedéseket vizsgálják, amely a csomag megadott részeinek a vizsgálatát takarja. Ezek a részek a következők lehetnek:
cél/forrás IP cím: például forrásra „match ip src 12.3.0/24”, a cél ugyanez, csak src helyett dst van. Ezzel nem csak egy OP címet, hanem egy egész tartományt teszünk illeszkedővé. Ha a mask /32, azaz egy egyszerű hosztról van szó, akkor ezt nem kötelező kiírni.
Cél/forrás port (minden IP protokollra): forrás esetén: „match ip sport 80 0xffff”, cél esetén: „match ip dport 80 0xffff”.
IP protokoll alapján (tcp, udp, icmp, gre): icmp protokollra: „match ip protocol 1 0xff”. Néhány protokoll száma: icmp -1, tcp – 6, udp – 17, gre – 47.
Az u32 illeszkedésen kívül természetesen lehet használni más illeszkedéseket is:
fw jelöléses illeszkedés: routolás során meg tudunk jelölni csomagokat (például iptables-szel), mely jelek a csomag a routeren való tartózkodása során van életben. Így tudunk kapcsolatot teremteni az egyes interfészek között. Ez a következőképpen történik:

```
# tc filter add dev eth1 protocol ip parent 1:0 prio 1 handle 6  \
  fw flowid 1:1
```

Látjuk, hogy ez a szűrő az egyes hálózati eszközhöz tartozik, és a 6-os jelű csomagokat az 1:1-es osztályba továbbítja. Ez a jelet például így is megkaphatta:

```
# iptables -A PREROUTING -t mangle -i eth0 -j MARK --set-mark 6
```

A csomag a 6-os jelet a nullás eszközön kapta meg, és az egyes eszközön használtuk fel. Így a tc parancs ezen részének bővebb ismerte nélkül is, nagyszerűen lehet szűrni a csomagokat. (Személy szerint én jobban preferálom ezt a megoldást, mert így elegendő egyszer jól megcsinálni tc-vel az osztály hierarchiát és később iptables-szel eldönteni, melyik csomag melyik osztályba is kerüljön. Általában amúgy is az iptables táblázatai gyakrabban módosulnak és kényelmesebb eszközöket biztosítanak ezekhez a gyakori módosításokhoz.)
A TOS mező alapján való szűrés is lehetséges. Például, ha a minimális várakozású csomagokat akarjuk szűrni:

```
# tc filter add dev ppp0 parent 1:0 protocol ip prio 10 u32 \
      match ip tos 0x10 0xff \
     flowid 1:4
```

De például a nagy mennyiségű (bulk) forgalom esetén „0x08 0xff”-et kell használni.

3.2. Fejlettebb szűrők a csomagok (újra)osztályozására

Ebben a fejezetben néhány szűrőt veszünk részletesebben szemügyre. Egy lista a teljesség igénye nélkül:

fw

Egy tűzfal általi megjelölés alapján dönt.

U32

A csomag fejlécbeli mező alapján dönt (például: forrás IP).

route

Mint a neve is mutatja, csomag útvonala alapján dönt.

rsvp, rsvp6

A csomagok irányítása az RSVP alapján történik. Ezt olyan hálózatokban érdemes használni, ami a mi irányításunk alatt van, mert az Internet nem tolerálja az RSVP.

tcindex

A DSMARK sorbaállítási módszernél alkalmazott szűrő.

Látszik, hogy számtalan lehetőség van általában a csomagok osztályozására, renszertől függő lehet, hogy melyik használata az előnyösebb. A legtöbb szűrőnek vannak olyan paraméterei melyek közösek:

protocol

Mely protokollokat kívánjuk elfogadni a szűrővel. Ez legtöbbször az IP szokott lenni. Ezt a paramétert kötelezően meg kell adni.

parent

Azt az azonosítót adja meg, amihez a szűrőt hozzárendeltük. Például: HTB-nél ez a gyökér módszer (1:). Ezt is kötelező megadni a definició során.

prio

Az osztályozó prioritása. Az alacsonyabb értékű kerül előbb ellenőrzésre.

handle:

A jelentése egyes szűrőknél eltérő lehet.

A továbbiakhoz leszögezünk pár dolgot, ami a fejezet végéig élni fog: az formálandó forgalom HostA fele megy ki, a gyökérosztály az 1:-en van, kifele a kiválasztott forgalom az 1:1-es osztályon keresztül halad.

3.2.1. Az u32 osztályozó

A jelenleg megvalósított szűrők közül az u32 a legfejlettebb. Az u32 hashelési táblázatokra épül, ez teszi akkor is erőteljessé, amikor sok-sok szűrőszabály van. A legegyszerűbb formájában ez csupán rekordok listája, melyek két részből állnak: a kiválasztó (selector) és a hatás (action). A kiválasztók mindig az éppen feldolgozás alatt lévő IP csomaggal végzik az összehasonlítást, egészen az első illeszkedésig. Ezután a hozzátársított hatás lép érvénybe, hajtódik végre. Az egyik legegyszerűbb típusú hatás az az, hogy a csomagot átirányítjuk egy másik osztályba.

Most is a tc programot fogjuk a szűrők beállításához használni. A parancsok három részből fognak állni: a szűrő specifikáció, a kiválasztó és a hatás. A specifikációban a már korábban tárgyalt általános paraméterek és a tc-hez szükséges részek találhatók:

```
tc filter add dev IF [ protocol PROTO ]
                     [ (preference|priority) PRIO ]
                     [ parent CBQ ]
```

A továbbiakban, mint általában is, a protocol az „ip” lesz. A preference paraméterrel a szűrő prioritását állíthatjuk be, de használhatjuk erre a priority értéket is. Ez fontos, ha sok szűrőnk (szabályok listája) van különböző prioritásokkal. Ekkor a listában a szabályok a hozzáadás sorrendjében kerülnek elfogadásra, de a listák feldolgozási sorrendje prioritás alapján történik. A parent paraméterrel a szűrő szülőjét adhatjuk meg, ez az adott módszer azonosítója lesz.

3.2.1.1. Az u32 kiválasztója (selector)

Az u32 szelektora azokat az információkat tartalmazza, melyek alapján el tudja dönteni, hogy a feldolgozás alatt levő csomag illeszkedik-e erre a mintára. Azaz pontosabban megfogalmazva, definiálja azokat a biteket melyeknek illeszkednie kell a csomag fejlécében, ezenkívüli részeknek nem kell. De ezt az egyszerű műveletet nagyon hatásosan végzi. Nézzük a következő példát az életből:

```
# tc filter add dev eth0 protocol ip parent 1:0 pref 10 u32 \
  match u32 00100000 00ff0000 at 0 flowid 1:10
```

Az első sor ennek a fejezetnek a szempontjából nem lényeges, csak azokat a paramétereket tartalmazza, melyek a szűrő hash táblájának kellenek. A második sor tartalmazza a kiválasztással kapcsolatos információkat. A szelekor azokra az IP fejlécekre fog illeszkedni, melyek a második byte-ja 0x10 (0010). A 00ff szám lesz az illeszkedés maszkja, azaz megadja mely biteknek kell meg egyezniük. Ez jelen esetben az ff miatt akkor történik meg, ha a byte pontosan 0x10. Az a paraméter az illeszkedés kezdetét jelenti, ebben az esetben ez a csomag eleje. Most ez azt jelenti, hogy azon csomagok fognak illeszkedni, melyeknek a ToS mezőben be van állítva az alacsony várakoztatási bit (ls. ToS mezőt magyarázó fejezet). Most nézzünk egy másik példát:

```
# tc filter add dev eth0 protocol ip parent 1:0 pref 10 u32 \
  match u32 00000016 0000ffff at nexthdr+0 flowid 1:10
```

Ami újdonság lehet itt az az at paraméternél szereplő nexthdr. Ez a következő fejlécet adja

meg, ami ebbe az IP csomagba be van ágyazva, azaz egy magasabb szintű protokoll fejlécének a kezdete. Így ebben a példában az illeszkedés a következő fejléc kezdetétől lesz vizsgálva. Ennek a szűrőnek a második 32 bites szóra lesz hatása. A TCP és UDP protokolloknál ez a rész tartalmazza a cél portot, és ha a 0x0016 hexaszámot átalakítjuk decimálissá, akkor 22-őt kapunk, ami az SSH portja.
Végül nézzünk egy olyan illeszkedést, melynél a célportot használjuk fel: match 9db50000 ffff0000 at 16. Itt a 17-ik byte-tól kezdődően kettő byte illeszkedését vizsgáljuk. Most ez azon csomagokat választja ki melyek a 157.181.0.0/16-os ip tartományba tartanak (ELTENET).

3.2.1.2. Általános szelektorok

Mint láthattuk az általános szelektorok olyan mintát, maszkot és minta-eltolást definiálnak, melyek a csomag tartalmára kell illeszkedniük. Az illeszkedést természetesen az IP fejléc bármelyik bitjére vizsgálhatjuk (sőt magassabb szintű fejléceken is). Ezeket a szűrőket sokkal nehezebb írni és olvasni, mint a specifikus szelektorral rendelkező társaikat. A következő szintaxissal rendelkeznek:

```
match [ u32 | u16 | u8 ] PATTERN MASK [ at OFFSET | nexthdr+OFFSET]
```

Az u32, u16, u8 valamelyike határozza meg, hogy éppen mekkora hosszon vizsgáljuk az illeszkedést. A pattern lesz a minta, és a mask pedig a maszk, ezeknek a hossza meg kell, hogy feleljen a ux paraméterben beállítottnak. Az offset paraméter azt az eltolást jelenti, ahonnan a csomagnak illeszkednie kell az adott mintára. Ez byte-ban értendő. A nexthdr változóban ugorhatunk egy magasabb szintű protokoll fejlécének a kezdetére. Néhány példa:
A következő utasítás olyan csomagokat irányít át melyek élettartama (TTL) 64. A TTL mező az IP fejléc nyolcadik byte-jától kezdődik és egy byte hosszú.

```
# tc filter add dev eth0 parent 1:0 prio 10 u32 \
     match u8 64 0xff at 8 flowid 1:4
```

A következő példa eléggé érdekes, olyan csomagokat szűrünk melyek ACK bitje igaz és a méretük kisebb, mint 64 byte:

```
# tc filter add dev eth0 parent 1:0 protocol ip prio 10 u32 \
     match ip protocol 6 0xff \
     match u8 0x10 0xff at nexthdr+13 \
     match u16 0x0000 0xffc0 at 2 \
     flowid 1:3
```

Ebben a példában előfordul az az eset, amikor több szelektorunk van. Ebben az esetben a végeredmény a szelektorok eredménye lesz összeéselve (logikai és). Az első szelektorban, ahol a protokollt határozzuk meg, azaz most a 6-osat (TCP), csinálhatunk egy IP fejléc tizedik byte-jára illeszkedő szabályt, ugyanis itt van a protokoll meghatározása: match u8 0x06 0xff at 9. Viszont az ACK bit meghatározására nincs specifikus szelekor, ezért használjuk az általánosat. Az ACK bit a TCP fejlécben (nem az IP-jében) a 14-ik byte első felében van (0x10). Egy másik megoldásban akár mi is meghatározhatjuk az IP fejléc hosszát egy általános kiválasztással.

3.2.1.3. A specifikus szelektorok

A specifikus szelektorokat érdemes minden olyan helyzetben használni, amikor van, mert így a tc parancs sokkal jobban olvasható és könnyebben módosítható lesz. De mint láttuk egyes dolgok csak általános kiválasztással valósíthatóak meg, de azokkal minden, még a specifikusak is.

```
ip tos      -  1 byte
ip sport    -  2 byte
ip dport    -  2 byte
ip src      -  4 byte (nem szükséges maszk, és tartomány is megadható)
ip dst      -  4 byte (nem szükséges maszk, és tartomány is megadható)
ip protocol -  1 byte
```

3.2.2. A route osztályozó

Ez a szűrő a routing tábla eredményén alapul. Azaz amikor egy csomag osztályokon keresztül halad és elér egy route szűrőt, akkor a szűrő felbontja a csomagokat a routing tábla információi alapján. Egy route szűrő definiciója így kezdődik:

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 route
```

Itt egy route osztályozót adunk hozzá az 1:0-es csomóponthoz 100-as prioritással. Amikor egy csomag eléri ezt a csomópontot, akkor „összebeszél” a routing táblával, ha a csomag illeszkedik, akkor besorolódik a megfelelő osztályba 100-as prioritással. Ehhez a megfelelő routing bejegyzések szükségesek. Definiálunk egy „birodalmat” (realm) mely a cél vagy forrás akármelyikén alapul:

```
# ip route add hoszt/hálózat via átjáró dev eszköz realm realm_száma
```

Amikor egy route szűrőt csinálunk, akkor ezt a realm_számot tudjuk felhasználni, hogy azonosítsuk a hálózatot vagy hosztot az illeszkedés szempontjából. Itt megadhatjuk, hogy az nézzük ahonnan jött a csomag, vagy azt hogy hova tart. Íme egy példa:

```
# ip route add 157.181.0.0/16 dev eth0 realm 2
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 \
  route from 2 classid 1:2
```

Ez az ELTENET-ről jött csomagokkal való illeszkedést nézi és ez alapján dönt.

3.2.3. Vezérelves szűrők

Ezzel a szűrővel még bonyolultabb beállításokra nyílik lehetőség. Olyan szűrőt lehet létrehozni, ahol az illeszkedés alapját egy beállított sávszélesség érték is jelentheti. És mindezt nem egy egyszerű formában, hanem több lehetőséget is biztosítva. Meghatározhatjuk, hogy a sávszélesség túllépése esetén az egész forgalmat szüneteltetjük, vagy csak azt amennyivel túllépte a határt.
Azokkal a csomagokkal, melyek a túllépett sávszélességbe tartoznak több dolgot is tehetünk: eldobás, újraosztályozás vagy másik illeszkedő szűrő keresése.

3.2.3.1. A vezérlés útjai

Alapvetően két lehetőség van a vezérelvek megvalósítására. Ha a kernel az estimator opcióval fordítottuk (CONFIG_NET_ESTIMATOR=y), akkor a kernel meg tudja becsülni, hogy egyes szűrőkön keresztül mennyi forgalom megy át. Ez a becslés a CPU-t sem terheli le, mert másodpercenként 25-ször számolja meg mennyi adat ment át, majd ebből számolja ki a bitrate-et.
A másik mód a vezérlésre a már jól ismert TBF (vezérjeles vödrös algoritmus), most éppen a szűrőn belül. A TBF-re csak azok a forgalmak illeszkednek, melyek a beállított sávszélességen belül találhatóak, ha több lenne, akkor a felesleg egy „túlterhelési hatás" eljárás alá kerül, amit mi választhatunk meg hogy mi legyen.
A kernel becslése elégé egyszerűen állítható be, mert csak egy paramétert kell definiálni, az avrate-et, azaz az átlagos sebességet. Amíg a forgalom a meghatározott érték alatt van addig a szűrő a csomagokat a classid-ban meghatározott osztály fele küldi, de ha a sebesség meghaladja az értéket, akkor egy művelet hajtódik végre, ami alapértelmezésben az újraosztályozás (reclassify). A kernel exponenciális súlyozott mozgó átlaggal (ewma) számol, ami kevésbé érzékeny a rövid tüskékre.
A TBF-nek már több paramétere van: burst, buffer, maxburst, mtu, minburst, mpu és rate. Ezek teljesen úgy viselkednek, mint a vezérjeles vödrös algoritmus fejezetben tárgyaltak. De fontos, hogy az mtu érteke ne legyen nagyon alacsony, mert akkor egy csomagot sem fog átengedni a TBF.

3.2.3.2. Túlterhelési műveletek

Ha a szűrő úgy látja, hogy a forgalom túlterhelt, akkor egy műveletet hajt végre. Ez a művelet az alábbi négy közül kerülhet ki:

continue

Éppen az aktuális szűrőre nem illeszkedik a csomag, de a szűrő tovább engedi őt, hogy más szűrők vizsgálhassák.

drop

Ez egy eléggé durva megoldás, ami a forgalmat a meghatározott sebesség korláton belül tartja. Ez leggyakrabban a bejövő forgalom politikájánál szokták alkalmazni.

pass/ok

Ez akkor lehet hasznos, ha van egy bonyolult szűrőnk, de azt éppen nem akarjuk alkalmazni. Ezzel a módszerrel hatástalanná tehetjük a szűrőnket, miközben megmarad.

reclassify

ez az alapértelmezett művelet.

3.2.3.3. Példák

Egy példa, ha a bejövő icmp forgalmat akarjuk 2 kbit alá szorítani úgy, hogy a többletet eldobjuk:

```
# tc filter add dev eth0 parent ffff: protocol ip prio 20 \
     u32 match ip protocol 1 0xff \
     police rate 2kbit buffer 10k drop \
     flowid :1
```

Egy meghatározott méret feletti csomagok eldobása:

```
# tc filter add dev eth0 parent ffff: protocol ip prio 20 u32 match tos 0
0 \
     police mtu 84 drop flowid :1
```

3.2.4. Hashelési szűrők a nagyon gyors szűréshez

Eddig nem tértünk ki, hogy mennyi szabályt érdemes alkalmazni, vagy mennyi szabályig hatásosak ezek a szűrők. Nos bár nehéz meghatározni a számát, de pár száznál nem érdemes többet használni. Ha már többet akarunk, akár több ezeret, akkor mindenféleképpen az ebben a fejezetben tárgyalt hashelési szűrőket érdemes alkalmazni. Előfordulhat, hogy olyan környezetben kell a forgalmat szabályozni, ahol több ezer gép illetve kliens található. Ezeknek a QoS specifikációja eltérő lehet, ekkor az eddigi ismereteinkkel csak olyan szabályrendszert tudnánk létrehozni, melyek sok időt igényelnének az illeszkedés

eldöntéséhez.
Alapértelmezett helyzetben a szűrők egy nagy, csökkenő prioritású láncban helyezkednek el. Ha ezer szabályunk van, akkor akár ezer ellenőrzésre is szükség lehet, ahhoz, hogy eldöntsük mi is legyen egyetlen csomaggal. Gyorsíthatjuk az illeszkedés ellenőrzését úgy, hogy 256 láncot hozunk létre, négy szabállyal mindegyikben, és a csomagokat feltudjuk osztani a 256 lánc között úgy, hogy a csomag sorsát el tudják dönteni a benne levő szabályok.
A hasheléssel erre lehetőség nyílik. Képzeljük el, hogy van 1016 modemes ügyfelünk a hálózatunkban, ezek az alábbi IP tartományból kapnak IP címet: 1.2.0.1-1.2.3.254. Mindegyik ügyfél a három osztály valamelyikébe fog sorolódni. Az lapeset, hogy brute-force módon megcsináljuk az összes ügyfélhez a szűrőt valahogy így:

```
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.0.1 classid 1:1
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.0.2 classid 1:1
...
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.3.253 classid 1:3
# tc filter add dev eth1 parent 1:0 protocol ip prio 100 match ip src \
  1.2.3.254 classid 1:2
```

Láthatjuk, hogy ez nem lesz egy hatásos megoldás. Ahhoz, hogy felturbózzuk ezt egy kicsit, úgy alakítunk ki négyes csoportokat, hogy az IP cím utolsó számjegye legyen a hash-kulcs. Ezután majd legfeljebb négy ellenőrzésre lesz szükség, átlagosan kettőre.
A szükséges beállítás talán bonyolult, de időben mindenféleképpen megéri. Elsőként csináljuk meg a gyökér szűrőt, majd hozzuk létre a 254 táblát:

```
# tc filter add dev eth1 parent 1:0 prio 5 protocol ip u32
# tc filter add dev eth1 parent 1:0 prio 5 handle 2: protocol ip u32
divisor 256
```

Ezután létre hozzuk a bejegyzéseket, most példaképpen az 123-hoz tartozóakat:

```
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 2:7b: \
```

```
        match ip src 1.2.0.123 flowid 1:1
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 2:7b: \
        match ip src 1.2.1.123 flowid 1:2
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 2:7b: \
        match ip src 1.2.2.123 flowid 1:3
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 2:7b: \
        match ip src 1.2.3.123 flowid 1:2
```

Ezek a bejegyzések az 1.2.0.123, 1.2.1.123, 1.2.2.123 és 1.2.3.123 IP címekhez készültek, ezekről az címekről jött csomagok sorrendben az 1:1, 1:2, 1:3 és 1:2 osztályokba fognak kerülni. A hashelési láncokat magunknak kell definiálni (2:7b:)
Ezután létre kell hozni a hashelési szűrőt, ami a forgalmat a megfelelő táblába irányítja:

```
# tc filter add dev eth1 protocol ip parent 1:0 prio 5 u32 ht 800:: \
        match ip src 1.2.0.0/16 \
        hashkey mask 0x000000ff at 12 \
        link 2:
```

Az alapértelmezett hash-táblát 800::-nak hívják, és az összes szűrés innen indul ki. Ezután a forrás címe kerül kiválasztásra, ami az ip fejléc 12-, 13-, 14- és 15-ik byte-ján helyezkedik el, itt a maszk jelzi, hogy csak az utolsó rész az érdekes. Ezután a 2:-as hash táblára kerül a vezérlés, amit korábban definiáltunk. Nyilván az a jó hash tábla, ami teljesen egyenletes és a láncai minél rövidebbek.

3.3. Összefoglalás

Láthatjuk, hogy ezen szoftveres megoldások milyen sokrétűen fedik le a problémát, pedig a megvalósításukhoz nem szükséges semmilyen különleges fizikai eszköz, beállításukat még egy egyszerű alap hálózati ismeretekkel rendelkező ember is el tudja végezni. A hatásfok javításához persze szükség van ezen a területen szerzett tapasztalatokra, mert nem mindegy, hol milyen algoritmust használunk.
A tapasztalat azt mutatja, hogy ezek a módszerek tényleg sokat segíthetnek olyan helyeken, ahol sok gépnek az Internet összeköttetését kell figyelni, ráadásul ezek egy a belső hálózatnál jóval lassabb kimenő vonalon csatlakoznak a külső hálózatra.

4. Alternatív megoldások

Eddig láthattunk a szállítási rétegben és a hálózati rétegben megvalósított megoldásokkal. Ebből a hálózati rétegben létező megoldások a szigorúbbak, azaz egy felhasználó kevésbé tud ezen szigorítások ellen tenni. Míg a TCP helyett választhat másik megoldást is, addig a hálózati réteget nem tudja megkerülni, mert ebben a rétegben sokkal szabványosabb az együtt működés. Ezentúl torlódás megelőző lehetőségek rejtőznek az adatkapcsolati rétegben is, azaz van olyan cég mely olyan eszközöket gyárt melyek erre lehetőséget biztosítanak. Ekkor persze erősen szükség lehet az eszközök közötti kommunikációra, szükség van arra, hogy "azonos nyelven beszéljenek", ugyanazt a protokollt ismerjék.
Nézzük meg, hogy az egyik legelterjedtebb hálózati eszközöket gyártó Cisco milyen megoldásokat kínál (fellelhető dokumentációk alapján).

4.1. Nagy gyártók megoldásai

Nyilván egy eléggé nagy és befolyásos gyártóra van szükség ahhoz, hogy a többiekkel összefogva, vagy netalántán rákényszerítve arra, hogy egy egységesebb szabvány jöjjön létre, mely egy olyan protokollt írhat le mely kellően nagy szabadságot biztosít a beállíthatóságra, sok-sok szempont szerint. Ezek a gyártók mindig is törekedtek ezeknek a feltételeknek a minél előnyösebb megvalósítására a termékeikben. Sok megoldás született és halt meg, mert általában a különböző gyártók eszközei nem voltak kompatibilisek egymással, persze azonos gyártó eszközeivel, jól lehetett konfigurálni, de azon kívül nem volt a rendszer összehangolva. De gyártótól függetlenül is, sok lehetőség kínálkozik az egységesítésre, persze ezek némelyik már jóval túlmutat az adatkapcsolati réteg képességein.

4.1.1. A QoS ellenőrzés

A hálózat- és szolgáltatás minőségének a fogalmának a legáltalánosabb definiciója az ISO-tól származik: " a minőség valamely termék, rendszer vagy folyamat valamennyi saját

jellemzőjének az együttes képessége, hogy kielégítse a vevők és más érdekelt felek követelményeit". A hálózatnak a minőségét annak leggyengébb pontja határozza meg. A szolgáltatás minősége (QoS) definiálható a szolgáltató és a felhasználó szemszögéből is. A felhasználó szemszögéből a QoS a számára szükséges kritériumokat jelenti, melyek a szolgáltatás hiánymentes igénybevételéhez szükségesek. A szolgáltató szemszögéből a QoS azokat a képességeket jelenti, melyek hozzájárulnak a felhasználó követelményeit tükröző szolgáltatásokhoz. A QoS-t nem szabad összetéveszteni a hálózat minőségével, mert az csak egy része a szolgáltatás minőségének. A QoS az az amit a felhasználó majd ténylegesen tapasztal.

A problémát már rég óta az okozza, hogy az IP fölött számtalan szolgáltatás lehetséges egyszerre (VoIP, videó streamek, www, e-mail), melyek között a versengést meg kell tudni akadályozni. Ezek a szolgáltatások nem azonos tulajdonságokkal rendelkeznek, nem azonos mértékben és módon szeretnék használni az adott hálózatot. Egyes alkalmazások kevésbé érzékenyek az újraküldésre, ezek egy pont-pont stratégiát (pl. TCP) használhatnak az adatok érkezésének korrigálására. De a legtöbb interaktív, valós idejű alkalmazás ezt rosszul tűri. Ezek számára szükséges, hogy az átvitel minél gyorsabb és minél egyenletesebb legyen. Ez egy új szolgáltatás modellt igényel, ahol az egyes alkalmazások le tudják ellenőrizni, hogy a hálózat biztosítani tudja az igényeket.

Erre a QoS egy megoldást kínál úgy, hogy az irányítást a mi kezünkbe adja. A QoS szabálya az, hogy olyan utakat találjon melyek során sorba tudja állítani a szolgáltatásoknak szükséges sávszélességeket eleget téve a hálózati forgalom összesített folyamának egy adott pillanatban. Ez a képesség a priorizálásra és a forgalom szintjeinek a szabályozására egy kitüntetett tényező, és egy nagy különbség az "új világ" hálózatában. A QoS technika egy olyan hasznot jelent, mely során a rendelkezésre álló kapacitásokat a legnagyobb mértékben tudjuk használni az értékes szolgáltatások számára.

A QoS olyan hálózatokon használható, ahol biztosítva van a szolgáltatás-események feletti irányítás, azaz a hálózat jól meg van tervezve, a forgalom típus azonosítható, a belépés ellenőrizhető és szabályozható.

A QoS az alkalmazásokat két nagy csoportra osztja: a valós idejűekre és a nem-valós idejűekre. Ezutóbbiakat néha „hagyományos adat" alkalmazásoknak is nevezik, mert már régóta és többségben találhatók meg a hálózaton. Ilyen alkalmazások a Telnet, FTP, e-mail, WWW stb. Ezek az alkalmazások nem igényelnek a különösebb garanciát az adatok időbeliségére, azaz ha egy adat később érkezik, akkor nem történik semmi baj. Rugalmasan

reagálnak a megnövekedett várakozásokra. Ezeknek is jobb az alacsony várakozási idő, de nem lesznek használhatatlanok magas várakozási idő esetén sem. Míg egy valós idejű alkalmazás (audió stream) esetében, ahol a küldés és a fogadás azonos ütemben történik, elképzelhetetlen, hogy egy csomag eldobás majd újraküldés esetén a kapcsolat akadozzon, majd késsen. Erre egy megoldás a fogadó oldali puffer (playback point), de ez egy bizonyos eltolást, azaz késleltetést eredményez, és hosszabb idejű megnövekedett várakozás esetén a puffer kiürül, és akkor ugyanott vagyunk, ahonnan elindultunk. Ráadásul a késleltetés miatt a társalgások körülményessé válhatnak. Nyilvánvaló egy új megoldás (QoS) szükségessége.

4.1.1.1. A gyorstárazás szerepe

A gyorstár (cache) használata egy költséghatékony és széleskörben elterjedt módszer arra, hogy a gyakran megtekintett oldalakat a felhasználóhoz közel tárolva megelőzzük a külső hálózat terheltségét a kétszeres tartalom letöltéstől. Az mindegy, hogy ez a tartalom egy web oldal vagy egy videó stream, a lényeg ugyanaz. Mindkettő hatékony út a sávszélesség optimalizálás felé, mivel az adatok elérését a külső és belső hálózat határára helyezi. A Cisco erre célra fejlesztette ki a Web Gyorsítótár Kommunikációs Protokollt (WCCP), hogy az eszközök melyek ezt támogatják kommunikálni tudjanak a routerekkel. Ez a protokoll sok termékükben megtalálható, ezzel lehetővé téve a transzparens, skálázható és biztonságos bemutatását és alkalmazását a gyorstárazó technikának.

4.1.1.2. Szelektíven korlátozott hozzáférési arány

A foglalt hozzáférési arány (CAR) egy határ központú QoS mechanizmus, melyeket a Cisco IOS alapú hálózati eszközök támogatnak. A CAR ellenőrzött hozzáférési aránya lehetővé teszi, hogy meghatározzuk a felhasználók hozzáférési sebességét egy adott csomagból az IP cím, alkalmazás, precedencia, port vagy a MAC (Media Access Control – eszköz fizikai azonosító) címe alapján. Ez egy eléggé kézenfekvő megoldás, például ha egy szolgáltatás nagy nemkívánatos (esetlegesen durva) hálózati forgalmat bonyolít, akkor irányíthatjuk a CAR-t, hogy szabályozza a sebességét, azaz “elkedvteleníti” azt. Ezzel egy időben egy másik szolgáltatást, ami fontosabb, azt előlépteti és teljes sebességet ad neki (“bátorítja”),

ezzel növelve a hatékonyságot. Sőt olyan beállítási lehetőségekkel is rendelkezik, hogy a nemkívánatos forgalmak gyorsítótár-megkerülési kedvét is elveszi. Így megint csak nő a sebesség, mert azoknak kedvez, akik gyorsítótárat használnak, és azoknak nem kedvez, akik nem használják a tárat.

4.1.1.3. A hálózati folyamkapcsolás

A hálózati folyamkapcsolással nagy hatásfokot lehet elérni a hálózati rétegbeli szolgáltatásoknál és a szemcsézett adatok gyűjtésénél a hálózati határokon. Ennek felhasználásával a Cisco IOS-t támogató hosztok az irányításunk alá kerülnek. A hálózati folyamokon keresztül lehetőség nyílik folyamokon való mennyiségek összegyűjtésére, számolására, hálózat tervezésre és monitorozására. Egyes folyamok a következő tulajdonságokkal rendelkeznek: forrás és cél IP cím, a folyamok indulási és befejezési időbélyegzője, csomag- és byte-számláló, a következő hop router címe, bemenő és kimenő fizikai port interfész, cél és forrás TCP/UDP port, IP protokoll típusa, TOS mező, TCP flagek, cél és forrás autonóm rendszerek száma, valamint a cél és forrás alhálózati maszkok. Ezzel a magasan szemcsézett folyamokkal lehetőség nyílik a különböző szolgáltatások költségeinek mérésére napszak, szolgáltatásosztály, alkalmazás használtság és hálózati forgalom szerint.
A folyam mentén levő routerek mindegyike rendelkezik valamekkora pufferrel. Ezek a torlódás megelőzésében játszák a szokásos szerepüket. Egy kapcsolat csak akkor jön létre, amikor az kész útvonalon rendelkezésre áll a megfelelő mennyiségű puffer minden routernél. Ennek az egyik fő hibája, hogy erőforrás pazarláshoz vezethet, mert olyan erőforrások is lefoglalódhatnak, amiket esetleg még nem is használ ki az, aki lefoglalta.
A folyamok fontos absztrakciója az erőforrás lefoglalásoknak. A folyam fogalma azért helytállóbb, mint a csatorna elnevezés, mert itt lényeges szempont az, hogy a csatorna zárt és a közbenső része teljesen láthatatlan a külvilág fele, míg a folyamoknál nem csak a végpontok látszódnak, hanem a közbenső routerek is. Bár alapvetően mindkét megközelítés pont-pont alapú.

4.1.1.4. Az IP precedencia alapú forgalom osztályozás

Az IP precedencia lehetővé teszi a határokon a hálózati forgalom particionálását különböző szolgáltatás osztályokba. Valójában hat elkülönített osztállyal rendelkezik, plusz az ezekhez társított kiterjesztett hozzáférési listákkal (ACL), melyek lehetővé teszik a hálózati politikák alkalmazását, hogy a torlódást kezelhessük az egyes osztályoknál.
Amúgy az IP precedencia nem más, mint az IP fejlécbeli TOS mező 3 precedencia bitje. A precedencia társításokra meglehetősen rugalmas eszközök állnak rendelkezésre: ügyfél társítás (router, alkalmazás) és hálózati társítás (IP, MAC, port). Működhet kétféle módban is. Az első a passzív mód, ekkor az ügyfél állítja be a kívánt precedenciát. A másik mód az aktív, ahol különféle definiált politikák alapján állítjuk be, vagy írjuk felül a precedenciát. Az IP precedenciát be lehet úgy állítani, hogy a szomszédos, de heterogén hálózati technikák is fogadni tudják azt (Frame relay-ek, ATM-ek).

4.1.1.5. Sávszélesség foglalás (Integrated Services)

Itt a legfontosabb protokoll az RSVP (Resource Reservation Protocol) és Internet Engineering Task Force (IETF, RFC 2205), melyekkel a sávszélesség lefoglalása válik lehetővé helyi hálózatokban. Egy eszköz, amely a gerincen videót küld az RSVP-t használhatja, hogy jelezze a sávszélesség és QoS igényét a hálózaton. Ekkor az egész úton lefoglalódik ez neki, vagy egy jel küldődik vissza, hogy nem megfelelő kapacitás van valahol az út során. Az RSVP multicast környezetben igen előnyösen használható és skálázható.
Az integrált szolgáltatás olyan szolgáltatás osztályokat biztosít, melyek az egyes alkalmazások szükségleteit írja le. Ezenfelül meghatározza, hogy az RSVP-nek miként kell együttműködnie ezekkel az osztályokkal, hogyan foglaljon számukra erőforrást. Az egyik ilyen osztály azokhoz az alkalmazásokhoz lett kialakítva, melyek nem tűrik jól a késleltetett csomagokat. Ekkor a hálózat garantálni tud egy maximális várakozási értéket, amiből az alkalmazás beállíthatja azt az értéket, hogy mennyi információt kell ideiglenesen eltárolnia pufferelésre (playback point). Ez a garantált szolgáltatás. Ezenfelül még több osztály van a szolgáltatás osztályok csoportjában.
Most nézzük a menetét, hogy hogyan is zajlik ez az erőforrás lefoglalás. Először az

alkalmazás elmond néhány információt magáról: a csomagok célját, a szolgáltatás típusát, milyen adatokkal dolgozunk, és ennek néhány tulajdonságát, esetlegesen a szükséges erőforrásokat. Ez az információ a flowspec (azoknak a csomagoknak a csoportja, amit ez az alkalmazás kér lesz a folyam). Ennek két része van: a folyam karakterét leíró rész a Tspec, a folyam szükségleteit leíró rész az Rspec (ez a kisebb). Ennek megadása fontos, mert az engedélyezés során a protokoll a sok folyam ezen információja alapján dönti el, hogy melyeknek tudja biztosítani a zökkenőmentes továbbítást.
Ebből látszik, hogy a következő lépés az erőforrás foglalás menetében az az, hogy a hálózatnak el kell döntenie, mely szolgáltatásokat bír majd el, és melyeket kell elutasítania. A harmadik lépés az, hogy a hálózati eszközök megkapják a szükséges információkat ahhoz, hogy biztosítani tudják a szolgáltatásokhoz szükséges (a flowspec-ben leírt) erőforrásokat. Ez a rész maga az erőforrás foglalás. Végül a routereknek és a switcheknek a folyam fennálása során biztosítani kell az ígéreteket a megfelelő sorkezeléssel és ütemezéssel. Ez a menet utolsó lépése, a csomag ütemezés.
Az ígéretek betartását a következő módszer nagyban segíti.

4.1.1.6. A vezérjeles vödrös felmérés

Tudhatjuk már korábbról, hogy a hirtelen tüskékkel rendelkező forgalom az egész forgalom minőségére kihatással van, mert megnöveli a lappangási időt, „idegessé” teszi az egészet. Ez a vibrálás néhány alkalmazásnak tényleges problémát okoz, gondoljunk egy nagysebességű videó konferenciára, ahol a folyamatos közvetítés nagyon fontos lehet. A Cisco egy népszerű megoldást kínál a hirtelen tüskék megfékezésére nagy forgalomnál, ez a vezérjeles vödör alkalmazása. Ezt a technikát egyszerűen egy vödörrel és a vezérjelekkel lehet leírni. Egy csomag akkor haladhat át ezen a vödrön, ha elhasznált egy jelet, persze a jelek elfogyhatnak, és az utántöltést mi határozhatjuk meg, mind mennyiségileg és sebességileg. Mikor egy tüske érkezik, akkor az egyes csomagok addig mehetnek át, amíg azt a vezérjeles vödör jelekkel bírja. Amikor az érkező csomagok kimerítik a jelkészletet, akkor elérték a burst rate-t, azaz már nem lesznek elfogadottak a politika szerint. Ezeknek a kimaradt csomagoknak a kezelésére több lehetőség is van: várakoztatás egy sorban, alacsonyabb szintű sávszélességen való feldolgozás, eldobás, precedencia (újra)beállítás. (A módszer bővebben az osztálytalan sorbaállítási módszereknél tárgyaljuk.)

4.1.1.7. A kedvezményes sorok, súlyozottan egyenlő esélyű sorbaállítás

Egy másik gerinc alapú irányítási lehetőség (Cisco QoS) az a kombinálása a kedvezményes soroknak és a súlyozottan egyenlő esélyű soroknak (WFQ). A kedvezményes sor biztosítja a legfontosabb forgalmak gyors kezelését abban a pontban, ahol ez használva van. Ez arra van tervezve, hogy a fontos forgalom szigorú prioritást kapjon. Ez a sor a prioritás meghatározási forrását rugalmasan kezeli (protokoll, bejövő interfész, csomagméret, forrás és cél cím). A kedvezményes sor minden csomagot négy alsorba sorol, gazdaságos, standard, közepes és prémium. Azon csomagok melyek nem lettek besorolva a prioritás lista alapján, azok a közepes sorba kerülnek. A sor algoritmusa során teljesen előre helyezi a magasabb prioritás osztályban levő csomagokat az alacsonyabbak elé. Ez a megközelítés egyszerű és intuitív, de a várakozásokat áthelyezi a magasabb prioritási osztályokból az alacsonyabbakba, ezzel az alacsonyabb osztályokban növelve az idegességet (azaz uralkodóvá teszi a magasabb forgalmat). Ez a magasabb prioritás osztályok maximális sebességének a korlátozásával elkerülhetjük.
A másik kezünkben viszont ott a WFQ, ami olyan eredményes kezelést biztosít a magas prioritás osztályoknak, melyek alacsony várakozási időt igényelnek, hogy közben a maradék sávszélességet egyenlően felosztja az alacsonyabb prioritásúak között. Azaz a forgalmat fölosztja, alacsony és magas prioritású folyamokra (például IP precedencia alapján), a magasak azonnali kezelést kapnak, az alacsonyak pedig egyenlően osztozkodhatnak a maradékon.

4.1.1.8. A Random Early Detection (RED)

A RED egy olyan eszközt ad, mellyel rugalmasan állíthatjuk be a forgalom kezelő politikát úgy, hogy az áteresztő képességet maximalizáljuk a torlódási feltételeket betartva. A RED segít a torlódások értelmes elkerülésében olyan algoritmusok megvalósításával melyek a hosztoknak védelmet ad és magában foglalja a következő képességeket:
Megkülönbözteti az elfogadható és túlzó forgalom tüskéket. Együttműködik a forgalom forrásával, hogy elkerüljék a TCP lassú kezdés ingadozását, ami periodikus hullámokat hozhat létre a hálózati torlódásokban. A sávszélességet a felhasználás arányában osztja fel.

Beállítja a minimum és maximum szintet a sor hosszának küszöbértékeire (csomageldobási valószínűség).
Ezekért a RED együttműködve a TCP-vel előrelát és kezeli a torlódást a nagy forgalom alatt, hogy azt maximalizálja.

5. Összegzés és tapasztalatok

Most, hogy sikerült a torlódás és a hozzá kapcsolódó témáknak nagy részét áttekinteni, láthatjuk, hogy ennek az egyszerű problémának a megoldása nem is olyan egyértelmű, mint amilyennek látszik. A tárgyalt módszerek egy része nem a kis hálózatokban felmerült esetekben alkalmazható, azaz a nagy hálózatoknál is igen fontos, hogy a forgalom minél zökkenőmentesebb legyen, sőt igazán itt a legfontosabb. Az ember elsőre azt gondolná, hogy itt akkora sebesség határok vannak, melyeket nehéz elérni. Sokszor a torlódások előtt már a forgalom egyenetlensége jelentkezik, amiket a tárgyal módszerekkel hatékonyan ki lehet védeni.

5.1. Személyes tapasztalatok

Ezt a témát azért választottam, mert fontosnak éreztem, mégpedig azért, mert ahol én lakom ez nagy problémát jelent. Ez egy kollégium, ahol több, mint 100 gép csatlakozik az Internetre, és a kimenő/bejövő összes sávszélesség igen csekély (szám szerint 1,8 Mbit, ami gépenként alig több, mint 1 kbyte/másodperces sebességet jelent, ezt egy egyszerű modem is képes lehet felülmúlni). A gépek igen magas száma miatt a torlódás fokozottan jelentkezett. Ezt az egyszerű felhasználó onnan láthatta, hogy a weblapok nagyon lassan jelentek meg, és a válaszidők is siralmasak voltak. A csomagok 20-30%-a odaveszett, de némely esetekben nem volt ritka az 50-60%-os veszteség sem. A válaszidőket, például a ping programmal nézve, lemérve az érték 2 és 3 másodperc közé esett. Ilyen válaszidők mellett, számolva a csomagveszteségekkel is, semmilyen interaktív program futtatására nem volt lehetőség. Emellett a sávszélesség kihasználtságára vonatkozó adatok megmutatták, hogy a hasznos forgalom kb. a tényleges forgalom felére, de legtöbbször az alá esett vissza.
Másfél éve megadatott, hogy ebben a kollégiumban rendszergazda legyek. Így lehetőségem lett a rendszerbe mélyebben belelátni, felderíteni annak gyengeségeit. A probléma adott volt, de a megoldás eleinte nem látszódott. Ekkor kezdtem a témában jobban elmélyedni, kerestem a lehetséges megoldásokat, azt is figyelembe véve, hogy a kollégium semmilyen pénzügyi támogatást nem volt hajlandó biztosítani. Így csak a meglévő eszközökre

hagyatkozhattam, ami a routert megszemélyesítő PC-ben ki is merült. A legnagyobb segítség a Linux maga, hiszen a kernel meglehetősen sok módszert támogat. A sok egymástól jól elkülöníthető gép figyelembe vételével egyértelmű volt, hogy valamelyik osztályos sorbaállítási módszer lesz a kiindulási alap. Ebben az időszakban kezdett a kernel részévé válni a HTB, mely már korábbról megmutatta, hogy a CBQ méltó vetélytársa lehet. A választásom erre esett, a sok dokumentáció olvasgatás után (persze nem ez volt az első választás, de az első sikeresnek mondható).

A forgalom osztályozásánál figyelembe vettem a csomagok funkcióját, azaz, hogy ezek a csomagok kérések-e vagy már a kapott válaszok, melyek már adatot tartalmaznak. A figyelés részletesen csak a kimenő forgalomra vonatkozott, mint azt már korábban is említettem. Már a kimenő forgalomnál szabályozni szerettem volna a bejövő forgalmat. A szabályrendszerem nem kedvezett a nagy adatmennyiségű forgalmaknak inkább a kisebb és rövidebbeket szerette, azaz azokat melyek nem jelentettek folyamatos terhelést a rendszer számára. A kifejezetten filecserélőkhöz tartozó „folyamokat” erőteljesebb korlátozás alá vetettem. A rendszer külön problémája az, hogy csak half-duplex módban képes működni, tehát ha a forgalom erőteljes kifelé, akkor az akadályozza a bejövő forgalmat. Ennek megfelelően, hogy az átlag felhasználónak jobb legyen, a kimenő forgalom nagy adat mennyiségű részét külön szigorral figyeltem és korlátoztam (a legtöbb felhasználó inkább letölt). Eközben figyelembe kellett venni a „hasznos” kimenő forgalom ne szenvedjen csorbát, például a kérések, vagy a levélküldés.

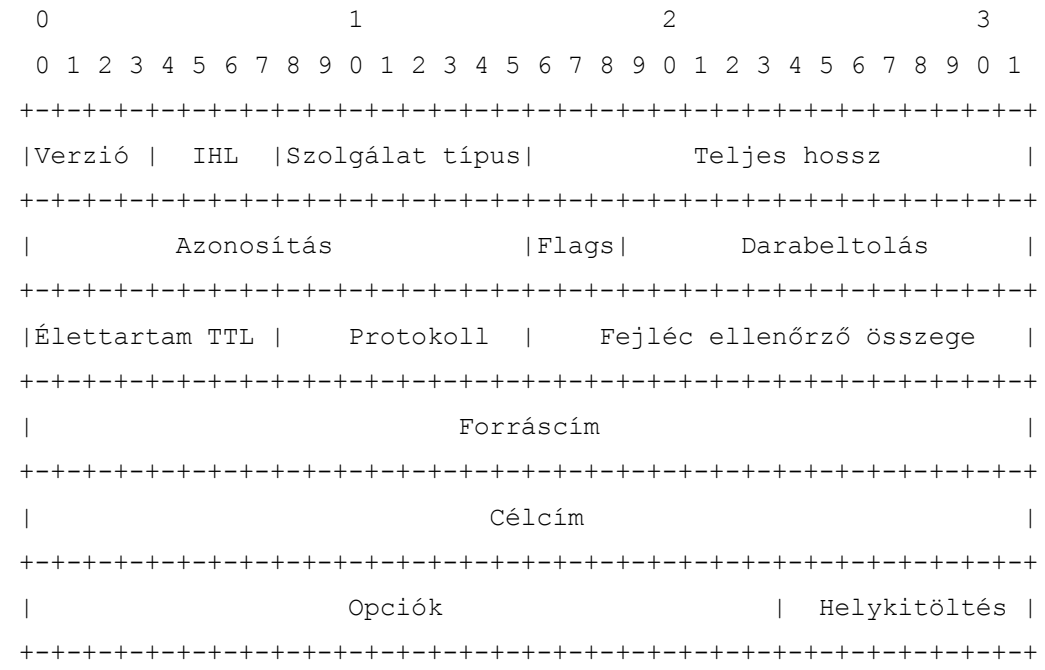
Az intézkedéseim hatására az átlagos válaszidő a tizedére (!), a csomagveszteség a felére csökkent, a hálózat kihasználtsága pedig másfélszeresére nőtt (70-80%). Az vitathatatlan, hogy a módszer sikert hozott, de közben az is látszik, hogy nem „mindenható”, mert bár kisebb mértékben, de csomagok még mindig elvesznek. Ez nagyban köszönhető a hálózat sebességének és a gépek számának aránytalanságával. Én egyértelműen látom, hogy az Internet és az általános számítógép-hálózatok a mai napokban nem nélkülözheti a torlódás védelmi módszereket.

Jelmagyarázat

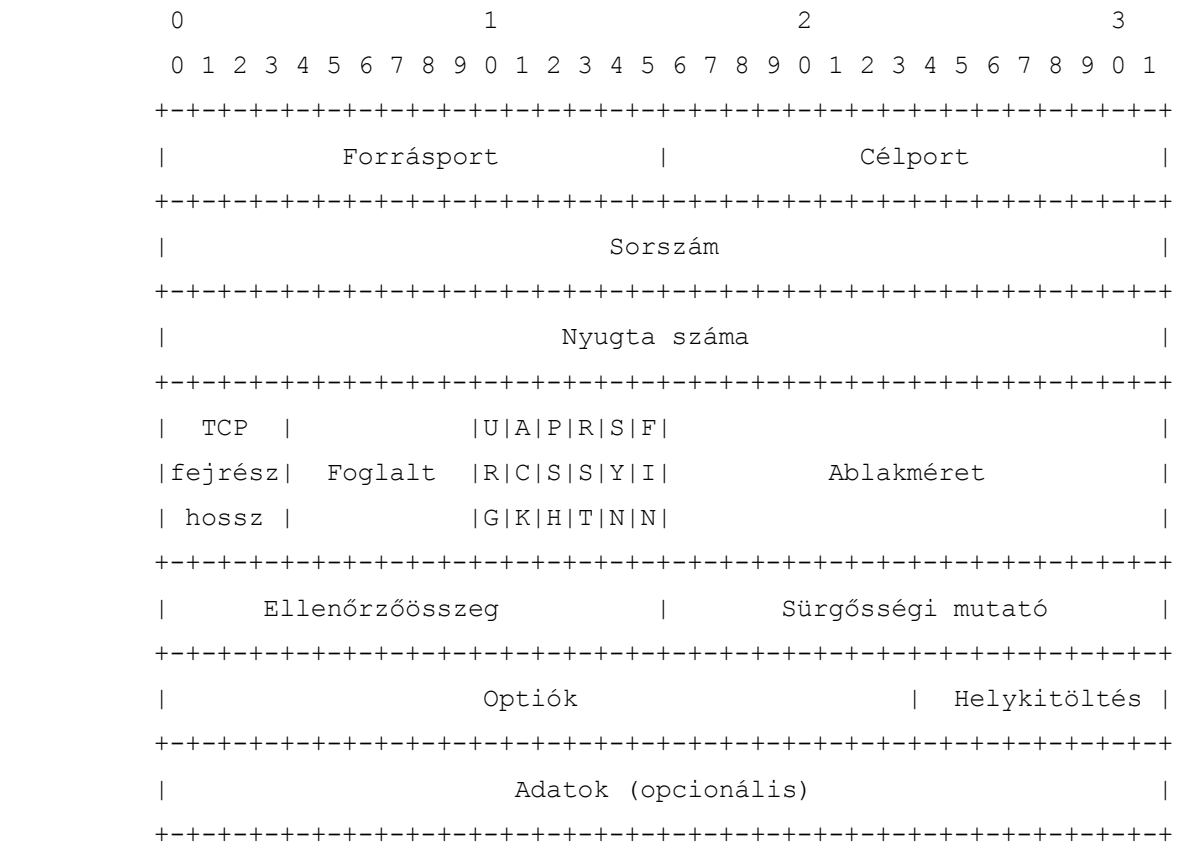
- **Interaktivitás, interaktív protokoll**: olyan protokoll, ami a hálózat egyik oldalán történő változásokra a másik oldalon azonnali reakciót vált ki, például az ssh (nem scp).
- **„Megkülönböztetett Szolgáltatások" (Differentiated Services)** RFC2475
- s**orbaállítási módszer (queueing discipline, qdisc)**: egy algoritmus ami az eszköz sorát kezeli. Ez lehet kimenő (egress) és bejövő (ingress)
- **gyökér qdisc**: az a módszer ami közvetlenül az eszközhöz van rendelve
- **osztálytalan sorbaállítási módszer (classless qdisc)**: olyan módszer, aminek nincsen külön beállítható alrészei, alosztályai
- **osztályos módszerek (classful qdisc)**: az osztályos qdisc összetett osztályokat tartalmaz, néhány ezek közül további sorbaállítási módszereket alkalmaz, melyek szintén lehetnek osztályosak. Ha szigorúan vesszük a definiciót, akkor a pfifo_fast is osztályos, mert három kötegből áll, amik valójában osztályok. De a felhasználó szempontjából ezek nem szabályozhatóak, azaz osztálytalan, mert az osztályai nem módosíthatóak a tc által.
- **Osztályok**: Egy osztályos sorbaállítási módszer rendelkezhet, néhány osztállyal, melyek a módszernek belső osztályai. De egy osztálynak is lehetnek további alosztályai, vagy sorbaállítási módszerei (qdisc). Ekkor a szülő az osztály. Levél osztály az, aminek már nincs gyerek osztálya. Az ilyen osztályhoz csak egy qdisc tartozik, ez felel az ebből az osztályból jövő adatok küldéséért. Mikor létrehozunk egy osztályt, akkor egy fifo qdisc is hozzárendelődik. Ha ehhez az osztályhoz új gyerek osztályt hozunk létre, akkor ez a qdisc megsemmisül. A levél osztályok fifo qdisc-jét később ki lehet cserélni egy megfelelőbb módszerre, ami természetesen lehet akár osztályos is.
- **Osztályozó**: Minden osztályos módszernek szüksége van erre, ami meghatározza, hogy az egyes osztályokból kikerülő csomagok hova kerüljenek.
- **Szűrő (filter)**: Az osztályozás során szükség van szűrőkre, hogy meg tudjuk határozni, mikor melyik osztályt kell alkalmazni. Ezért egy szűrő több feltételből áll, és a szűrő akkor fogad el egy csomagot, ha az összes feltétel teljesül.
- **Ütemezés**: Néha a módszer úgy dönthet (az osztályozó segítségével), hogy egyes csomagok korábban hagyják el az osztályt, mint a többi várakozó csomag. Ez a folyamat az ütemezés. Újrarendezésnek is szokták hívni, de ez zavaró lehet.
- **Alakítás, formálás (shaping)**: azt a folyamatot, amikor a csomagokat kiküldésük előtt várakoztatjuk, hogy a forgalom a beállított maximális áteresztőképességnek megfeleljen.

Formálásról a kimenő forgalom esetén beszélhetünk.(Amikor csomagokat dobunk el, hogy a forgalmat lassítsuk, gyakran ezt is formálásnak nevezzük.)

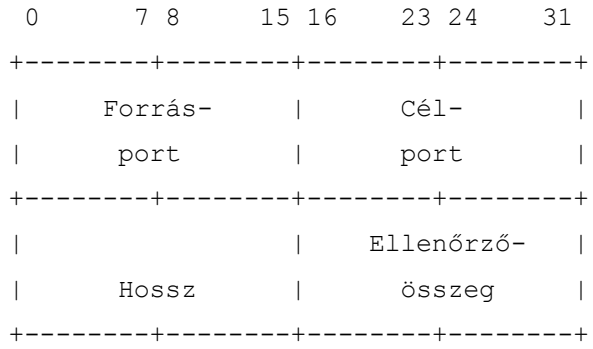
- **Politika (policing)**: A csomagok várakoztatása és eldobása, annak érdekében, hogy a forgalmat a sávszélesség alatt tartsuk. Linuxban a politika csak csomag eldobásra képes, nem tud várakoztatni, mert nincs a bejövő csomagoknak sora.
- **A munkamegőrző módszer(Work-Conserving és non-Work-Conserving):** a munkamegőrző módszer mindig követel csomagot, ha van, azaz sohasem várakozik, ha a hálózati eszköz tud adni. Míg a nem munkamegőrző módszernél (például a vezérjeles vödrös algoritmus) szükséges lehet, hogy csomagokat tartson meg annak érdekében, hogy a sávszélesség korlátokat be tudja tartani. Azaz annak ellenére, hogy már tudnának neki adni csomagot, a nem munkamegőrző módszer nem fog kérni egyet sem bizonyos esetekben.
- **IP fejléc:**



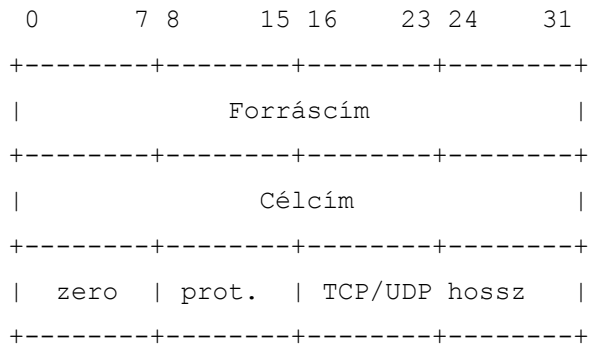
- **TCP fejléc:**



- **UDP fejléc:**



- **TCP/UDP pszeudó fejléc:**



Források

- lartc.org (2004.02.28)
- Számítógép-hálózatok 5.3. fejezet (Andrew S. Tanenbaum) – Panem Könyvkiadó Kft. 1999
- http://wipl-wrr.sourceforge.net/ (2004.02.09.)
- http://www.szabilinux.hu/konya/indul.htm (2004.02.27)
- http://www.szabilinux.hu/forditasok/hunglossary.html (2004.02.27)
- http://www.isi.edu/div7/rsvp/overview.html (2004.03.23)
- http://www.faqs.org/rfcs (2004.05.15)
- http://irh.inf.unideb.hu/user/jsztrik/education/05/tartalom.html (2004.05.16)
- Congestion avoidance and control (V. Jackobson) – Proceedings of the SIGCOMM'88 Symposium 1988
- Supporting real-time applications in an integrates services packet network (D. Clarc, S. Shenker, L. Zhang) – Proceedings of the SIGCOMM'92 Symposium 1992
- Random early detection gateways for congestion avoidance (S. Floyd és V. Jackobson) – 1993
- Congestion Control in Computer Networks: Issues and Trends (R. Jain) – IEEE Network Magazine, vol. 4 (24-30), 1993 május-június
- A Taxonomy for Congestion Control Algoriths in Packes Switching Networks (C-Q. Yang, A.V.S. Reddy) – IEEE Network Magazine, vol. 9 (34-45), 1995 július-augusztus