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

Egyedi hálózati ütemezési algoritmus implementálása és tesztelése Linuxban

TDK DOLGOZAT

Készítette
Pelyva Miklós

Konzulensek
Megyesi Péter
Dr. Molnár Sándor

Ipari konzulensek
Dr. Rácz Sándor
Dr. Szabó Géza

2014. október 22.

Tartalomjegyzék

1. fejezet

Bevezető és motiváció

1.1. Áttekintés

Bátran kijelenthető, hogy az internet korában élő embernek az egyik legfontosabb szükséglete az információszerzés lett. A Cisco felmérése szerint [14] 2013-ban 4,1 milliárd ember használt nagyjából 7 milliárd mobilkészüléket (mint például okostelefont, tabletet, laptopot) és ez a szám csak nőni fog: 2018-ra 4,9 milliárd földlakónak lehet összesen több mint 10 milliárd mobilkészüléke.

Az okoskészülékek pedig egyre nagyobb számban jelennek meg a piacon, melyek talán a klasszikus értelemben vett telefonálásnál is fontosabb szolgáltatást képesek nyújtani: az internet elérést.

Az okostelefonok és okoskészülékek tehát uralják a távközlési piacot, a hálózatok ütemesen fejlődnek, folyamatosan állnak át a 3G és 4G technológiára világszerte. Az Ericsson 2014-es jelentése szerint [6] 2019-re a világ lakosságának több mint 65%-a LTE, azaz 4G hálózatokon fog az internethez kapcsolódni. A svéd telekommunikációs óriás az adatforgalom arányainak alakulására is kitér, melynek legnagyobb részét manapság már a videómegosztás teszi ki: 2013-ban a mobil adatforgalom közel 40%-át videók nézésére használták fel. Ez a szám 2019-re több mint 50% lehet.

Egyre nagyobb szükség van tehát a gyors és megbízható hálózatokra, mert az igények is egyre csak nőnek. Az embereket egyre jobban zavarják a nem elég jó kép- és hangminőségű videók. Egy amerikai, médiaadatfolyam-szolgáltatásokat kínáló cég felmérése szerint a felhasználók 90%-ának van szüksége videó-hozzáférésre nap mint nap és 68%-uk tartja fontosnak azt, hogy eközben nagy felbontású (HD – High Definition) minőségű képsorokat láthassanak [8].

1.2. Motiváció

A fentiekből kikövetkeztethető tehát, hogy a jelenben és a közeli jövőben nagyon nagy figyelem fordul a videó adatfolyamok hatékony, gyors és jó minőségű küldése irányába. Sok hálózat azonban nem elég fejlett és nem áll készen arra, hogy infrastrukturális fejlesztésekkel jelentős mértékben növelje adatforgalmát. Ezekben az esetekben a szolgáltatóknak egyéb megoldásokhoz kell folyamodniuk.

Fontos terület a videó kódolók fejlesztése, illetve az adatfolyamok esetében az IP csomagok fejléceinek tömörítésével is kísérleteznek, de nagyon jelentős szerepük van a hálózati ütemező algoritmusoknak is. Ezek az ütemezési diszciplínák az út-

vonalválasztók kimenő interfészein minden egyes IP csomagot látnak és kezelnek, adatpufferekbe fűzik be, illetve veszik ki őket.

A fentieken kívül az ütemezők felelnek az adatátvitel szolgáltatásminőségéért is (QoS – Quality of Service), melyre akkor is nagy szükség van, ha becslések szerint a mobilinternet adatátviteli sebessége a duplájára is nőtt 2013 és 2014 között [14]. Az ütemező algoritmusok megfelelő konfigurálásával és működésével tehát jelentősen meg lehet növelni a videó adatcsomagok sebességét és megbízhatóságát a műsorszóró szerverek és a kliensek között.

A projekt fontos részét képezi az IP fejléc DSCP (Differentiated Services Code Point) mezőjének kihasználása, aminek egyedi változtatása alapján a különböző igényű alkalmazások megkülönböztetett kiszolgálása érdekében minden IP csomaghoz egyedi prioritást rendelhetünk hozzá. Ezzel az IP csomagokat a számukra megfelelő ütemezési sorba sorolhatjuk be, hogy ezek után a kifűzésnél minél hamarabb átjussanak a hálózaton, az okoseszköz felhasználójához.

Az Ericsson Magyarország Kft. Traffic Lab laboratóriumában folyó kutatásokhoz kapcsolódóan a TDK munkámban egy új és egyedi ütemezőnek a megvalósítását, implementálását és tesztelését végeztem el. A dolgozatomban ennek a kutatási és fejlesztési munkáit mutatom be részletesen dokumentálva az eredményeket.

Egy olyan ütemező algoritmust mutatok be, amely pontosan a videó adatfolyamokra specializált diszciplínát valósít meg. Az új koncepció szerint minden felhasználó minden alkalmazásához külön ütemezési sort rendelünk, ezáltal egyenként és egyedileg jól menedzselhető, könnyebben szabályozható forgalmat küldünk át a hálózatunkon.

A rövid bevezető után, a második fejezetben összefoglalom kutatómunkámat az ütemezési algoritmusokról és ismertetem a Linux kernelben használt legfontosabb ütemezési diszciplínákat, melyek a Linux hálózati vermének (network stack) legjelentősebb szolgáltatásminőségi ellenőrei. Ezen kívül szót ejtek a `tc` (traffic control) program használatáról is.

A harmadik fejezet az implementációt írja le. Elsőként ismertetem a külső konzulenseim által tervezett ütemező specifikációját. A fejezetben bemutatom azokat a szempontokat is, melyek alapján eldőlt, hogy mely már létező ütemező algoritmusból indul ki a munkám, illetve, hogy mely egyéb diszciplínák működése hatott az egyedi ütemező algoritmus elkészültére. Ezután a forráskód fontosabb részeit is kiemelem és tárgyalom.

A negyedik részben a tesztelést írom le. Bemutatom a fejlesztés során használt tesztkörnyezetet, amely virtuális gépen, egy Ubuntu disztribúción zajlott le. A tesztkörnyezet a Mininet [9] nevű hálózatemulátorban hoztam létre. Elsőként az ütemező algoritmus validációját mutatom be, majd összehasonlító méréseket végzek a Linuxban használt egyéb sorba rendezési diszciplínákkal.

Az ötödik, egyben utolsó fejezetben összefoglalom munkámat és a továbblépési lehetőségeket vizsgálom, amelyek alapján célkitűzést fogalmazok meg a jövőben elvégzendő munkákról is.

2. fejezet

Technológiai háttér és kapcsolódó irodalom

Ebben a fejezetben a munkám elméleti hátterét ismertetem. Elsőként részletesebben tárgyalom a szolgáltatásminőségi (QoS) mechanizmusokat, a fejezet második nagyobb részében pedig az ütemezőkről, illetve az ütemezési diszciplínákról írok, melyeket a `tc` (traffic control) programcsomaggal rendelhetjük Linux kernelben az eszközünk interfészeihez.

2.1. Az ütemezés

A számítógépeinken, okoseszközeinken és munkaállomásainkon sokféle szolgáltatást használunk, melyek egyéni igényt követelnek meg, mégis legtöbbször azonos linken kell őket kiszolgálni. Ez a konvergens hálózatok elterjedésének következménye, ugyanis minden adatcsomag IP felett halad át.

Erre a problémára adható logikus válasz a különböző átviteli szolgáltatások szétválasztása, differenciálása (Differentiated Services), melyet a hálózati csomópontoknak kell megoldaniuk. Azaz a hálózat csomópontjai, azokon belül is a hálózati ütemezők felelősek az adatforgalom szolgáltatásminőségéért.

2.2. QoS mechanizmusok

Egy hálózati megoldás QoS-ének biztosítása kell hogy legyen a második legfontosabb dolog azután, hogy a szolgáltatás egyáltalán létrejött és funkcionál [4]. A QoS nagy jelentőségéből adódik, hogy annak mértéke a hálózatban nem más, mint a felhasználók által érzékelt teljes teljesítmény.

2.2.1. Az ütemezők legjelentősebb QoS paraméterei

A hálózati szolgáltatás QoS-ének mértékét vizsgálva többféle szempontot kell figyelembe vennünk. Ilyenek például a sávszélesség, a továbbítási késleltetés, a jitter, azaz a késleltetési ingadozás, illetve a veszteségek.

Sávszélesség

A sávszélesség a legnagyobb adatátviteli sebesség, ami két végpont között fenntartható [10]. Az alkalmazások sávszélességi követelményeinek nehéz megfelelni ütemezők használata nélkül. Abban az esetben, ha igazságos módon osztjuk meg a teljes sávszélességet, az egységnyi érték csak attól fog függni, hogy hány darab folyam igényel a sávszélességből. Ez nagyon nehézzé teszi a szolgáltatásminőség betartását, ugyanis nem lesz biztosítható elég sávszélesség nagyobb igényű folyamoknak. Ennek elkerülése érdekében mindenképpen ütemezőt kell alkalmazni.

Átviteli késleltetés

A késleltetés a csomag forrástól való elküldése és a célba való megérkezése között eltelt idő. A csomagoknak időben kell megérkezniük a késésre érzékeny alkalmazásoknál, mint például az IP telefóniánál, vagy a műsorszórt multimédia-szolgáltatásoknál. A megnövekedett késleltetés csökkenti a szolgáltatás minőségét. Akkor fordul például elő, amikor a hálózati csomópontokon megtelnek a pufferek a ki- vagy bemenő oldalakon [10].

Késleltetési ingadozás – a jitter

A jitter a csomagok közötti érkezési variáció, a két végpont közötti átviteli késleltetés rövid idejű (< 100 ms) ingadozása [10]. A fogadó csomópontokon pufferelés használatával kontrollálni lehet a jittert. A hálózatban a küldőtől elvárható a megbízhatóság, hogy szabályos időközönként küldje el a csomagokat. A csomagok azonban késlekedhetnek a hálózaton való áthaladásuk során, vagyis nem érkeznek meg azonos, szabályos időközönként a fogadó állomásra. A jitter tulajdonképpen a különbség aközött, hogy a csomagot mikorra várjuk, és mikor érkezik meg.

Csomagvesztési arány

A csomagvesztési arány az elveszett és az átvitt csomagok aránya, mely leírja egy kommunikációs kapcsolat megbízhatóságát. A csomagok különböző okok miatt veszhetnek el a hálózaton belül. Ilyen például az, mikor torlódás van a hálózat egyik csomópontjában és egy puffer telítődik, ami miatt késlekedni fog a csomagtovábbítás. A telített pufferbe érkező csomagokat ebben az esetben el kell dobnia a hálózatnak. Egyéb okok lehetnek még a fizikai környezetben is, ilyen például, ha megszakad egy összeköttetés a hálózatban, vagy ha egy csomagot túl későn továbbít egy hálózat.

A csomag újra elküldésével a csomagvesztés kiküszöbölhető, TCP kapcsolatoknál kötelező, de a valós időben zajló szolgáltatásokat továbbító közegeknél (UDP kapcsolat esetén) csak megnöveli a késleltetést [10].

2.2.2. QoS biztosításának módszerei IP alapú hálózatokban

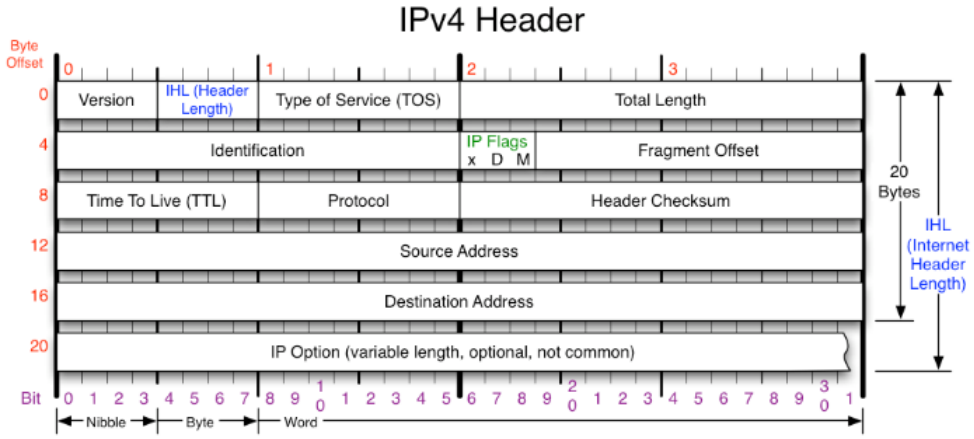
IntServ (Integrated Services)

Az IntServ keretrendszer folyamonkénti (*per flow*) szolgáltatást nyújt. Feladata az, hogy az adatfolyamokat elválassza az egyéb adatforgalmaktól. Három fő feladatot lát el: a befogadás-engedélyezés alapján az útválasztó csak akkor fogad be egy igényt, ha tudja neki garantálni a megfelelő QoS-t. A folyam útvonalának kijelölését a forrástól

a célig erőforrás-lefoglalásnak nevezzük, a forgalomszabályozás pedig arra szolgál, hogy a lefoglalt minőségi paramétereket kikényszerítse [3].

DiffServ (Differentiated Services)

A DiffServ a megkülönböztetett szolgáltatásokat foglalja magában, melyről már a 2.1. részben is szó esett. Nem egyedi folyamokhoz, hanem inkább kisszámú forgalomosztályhoz rendeli erőforrásait [2]. Feladata az IntServ hiányosságainak kiküszöbölése. A csomagokat egyenként minőségi osztályokba sorolja, illetve az éppen feldolgozásra váró csomagokat a minőségi osztályuk alapján kezeli [10]. Ehhez az IPv4 csomagok fejlécében található ToS (Type of Service) mezőt használja. A 2.1. ábrán látható az IPv4 fejléc és azon belül a ToS mező elhelyezkedése. Ez lesz az a 8 bites mező, ami alapján a TDK dolgozatban dokumentált egyedi ütemezési algoritmus a csomagokat ütemezési sorokba fogja besorolni.



2.1. ábra. Az IPv4 fejléc

2.3. Hálózati ütemezők

2.3.1. Bevezető

A hálózati ütemezők csomagkapcsolt hálózatok csomópontjaiban találhatóak meg, feladatuk az, hogy menedzseljék a hálózati forgalom kéréseit, a hálózatban átmenő csomagok sorozatait az interfészek küldő- (*egress*) és fogadó (*ingress*) oldalán egyaránt. Ezek az ütemezők tulajdonképpen adatpufferek, melyekbe a hálózati csomagok befűződnek (*enqueue*) és ideiglenesen tárolódnak, majd kifűződnek (*dequeue*) valamilyen algoritmus alapján.

A pufferterület akár különböző sorokra is bontható, melyek közül mindegyikben egy adott folyam csomagjai tárolódnak. A felkonfigurált besorolási szabályok határozzák meg, hogy mely csomag mely sorban kaphat helyet.

Az ütemezőkre azért van szükség, mert a hálózati erőforrások végesek, azaz sokszor fordul elő, hogy olyan igénybevételnek vannak kitéve, amelyet nem képesek kiszolgálni, így várakozni kell rájuk. De ezzel párhuzamosan az is előfordulhat, hogy nincsenek eléggé leterhelve a hálózati erőforrások. Ezt a két szélsőséget, vagyis a for-

galom statisztikus ingadozását az ütemezők használatával lehet kiküszöbölni, melyek hosszú idejű átlagban képesek lesznek a kérések kiszolgálására [15].

A kérdés csak az, hogy milyen stratégiákat és szempontokat vegyünk figyelembe a sorok felépítésénél, hogy a kiszolgálási elvárások a lehető legoptimálisabb módon teljesülhessenek.

Olyan feladatütemezési módszert kell alkalmazni, amely biztosítja a megfelelő kiszolgálási minőséget, illetve igazságosan osztja el az erőforrásokat a kérések között. Az előbbit a puffer túlcsordulása esetén csomagdobással, az utóbbit a kiszolgálási sorrend meghatározásával érhetjük el.

Az alkalmazásainknak legalább kétféle módját különböztetjük meg. Az egyik típus a késleltetést jól tűrő alkalmazás, mely el kell, hogy viselje a folyamatosan változó áteresztőképességet. Ilyenkor az erőforrások igazságosan vannak megosztva. Ezt nevezzük *best effort*, azaz legjobb szándékú igénynek, melynél nincs garancia arra, hogy a csomagok elérik a céljukat a hálózatban. A másik típus az intoleráns alkalmazások típusa, melyek garantált sávszélességet, illetve szolgáltatás-minőséget várnak. Ebben az esetben a vesztési arány és a késleltetés korlátozva van, az átviteli sebesség pedig garantált. Ez a típus a garantált szolgáltatású igény [4].

Összefoglalva tehát a *best effort* szolgáltatások védelmet és igazságosságot igényelnek, míg a garantált kiszolgálási igényű alkalmazások inkább teljesítménykorlátok biztosítását, illetve beengedés-szabályozást követelnek meg.

2.3.2. Hálózati ütemezési diszciplínák

A számítástechnika hőskora óta többféle hálózati ütemezési diszciplínát fejlesztettek ki. Az ütemezési algoritmusokat ezekben a diszciplínákban használják, melyek segítségével a hálózati interfészeken a csomagok továbbítása, sorokba-, illetve újrarendezése, késleltetése vagy eldobása is megtörténhet. A sorba rendezési diszciplínák mindegyikét azért fejlesztették ki, hogy kompenzálják a hálózatban fellépő, a 2.2 részben már említett különböző állapotokat, azaz, hogy megfelelő QoS-t biztosítsanak a csomagkapcsolt hálózatokban.

Minden hálózati interfésznek minimum két sora, más néven puffere van, ahol a csomagok egy bizonyos ideig helyet foglalnak, mielőtt továbbítódnának. Az egyik sor a bejövő csomagokért felel, ez a belépési sor (*ingress queue*). A kimenő csomagok pedig a kilépési sorban (*egress queue*) kapnak helyet. A sorba rendezési diszciplínák használatával a kilépési sor különböző beállításait tudjuk megváltoztatni, míg a belépési sorok fölött korlátozott a fennhatóságunk [11].

Kilépési sor (*egress queue*)

A sor kapacitását képesek vagyunk változtatni az `ifconfig` paranccsal, ekkor a `txqueuelen` mezőt kell átírnunk. A sorkapacitást nem bájtokban vagy bitekben mérik, hanem abban, hogy a sor hány darab csomagot tud egyszerre tárolni. Ha a sor tele van és további csomagok érkeznének bele, akkor túlcsordulás történik, azaz a bejövő csomagok eldobódnak és sosem érik el a céljukat.

A hálózati interfészek soros eszközök, azaz a csomagok egymás után hagyják el a sorokat és egymás után továbbítódnak a célállomásuk felé. Az ütemező fő feladata tehát az, hogy eldöntse, melyik csomag hagyja el következőleg a sort [11]. Ezt pedig egy bizonyos algoritmus és annak konfigurációja szerint határozza meg.

Belépési sor (*ingress queue*)

A belépési sorban nincs lehetőségünk a csomagok sorrendjének manipulációjára. Azon kívül, hogy olyan sorrendben továbbítjuk őket, mint ahogyan megérkeztek, az egyetlen ettől eltérő lehetőségünk a csomagok eldobása. A TCP kapcsolat esetében ebből is profitálhatunk azonban, ha eldobjuk a TCP „ACK" csomagokat: így a TCP protokoll észleli a torlódást és lecsökkenti az átvitel arányát a forrásnál [11].

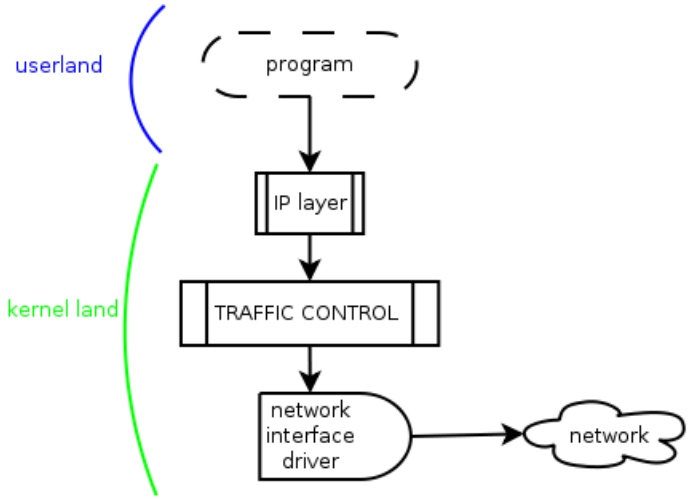
2.4. Forgalomszabályozás

Az IntServ keretrendszer forgalomszabályozási mechanizmusát a Linux kernelben a `tc` (traffic control) nevű programmal lehet nagyon hatékonyan gyakorlatba ültetni, melynek forráskódjai a `net/sched` könyvtárban találhatóak meg a kernel forrásfájljai között.

Az internet világában minden csomagokból épül fel, egy rendszer menedzselése tehát nem más, mint a csomagok menedzselése. A kernel térben elhelyezkedő, adatkapcsolati- és a szállítási réteg közötti `tc` alrendszerrel (2.2. ábra) a hálózati forgalmunk kimenő csomagjait tudjuk menedzselni, illetve manipulálni.

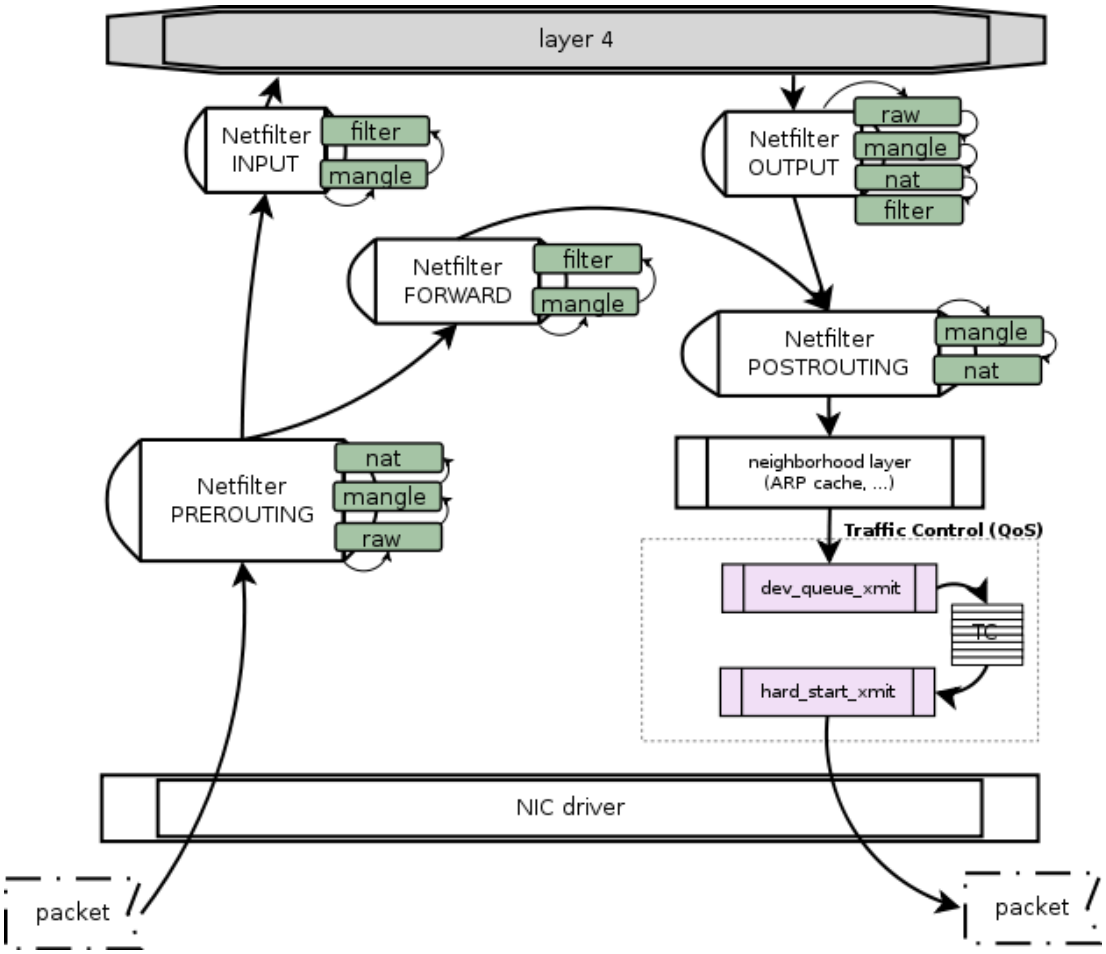
2.4.1. A `tc` – traffic control

A tc modul tehát állandóan használatban van, akkor is aktív a kernelben, ha éppen nem tudunk róla, hogy alkalmaznánk. Alapértelmezett beállításban ugyanis az ütemező egy *fifo* (First In, First Out) típusú sort menedzsel [1].



2.2. ábra. A traffic control elhelyezkedése a kernelben [1]

A 2.3. ábra egy csomag útját mutatja be a kernelben, attól kezdve, hogy belép (*ingress*) egészen addig, hogy elhagyja (*egress*) azt. Ha csak a kimenő útra koncentrálunk, láthatjuk, hogy a csomag a szállítási rétegből érkezik (transport layer), ami után a hálózati rétegbe jut (bár ez nincs ábrázolva a képen). Az `iptables` programcsomag által használt `netfilter` funkciók (`OUTPUT` és `POSTROUTING`) integrálva vannak a hálózati rétegbe és az IP manipulációs függvények között foglalnak helyet (fejléc készítés, fragmentálás). A `POSTROUTING` funkció `nat` kimeneténél a csomag a kimenő sorhoz (*egress queue*) továbbítódik, ahol a `tc` működésbe lép [1].



2.3. ábra. A csomag útja a kernelben [1]

A legtöbb hálózati eszköz sorokban tárolja a kimenő csomagjait, hogy azokat ott ütemezze. A kernel ehhez az ütemezéshez szolgáltat algoritmusokat, azaz ütemezési diszciplínákat, melyek manipulálni tudják a sorokat. A `tc` feladata az, hogy ezeket a diszciplínákat a kimenő sorhoz rendelve kiválogassa, hogy melyik csomag küldhető ki a hálózati interfészre.

A `tc` az `sk_buff` struktúrákkal dolgozik, ami a csomagokat reprezentálja a kernelben. Az `sk_buff` egy megosztott struktúra, azaz mindenhonnan elérhető, elkerülve így azt, hogy szükségtelen adatmásolatok, duplikációk jöjjenek létre. Az `sk_buff` tehát minden fontos információt tartalmaz az adott csomagról, ami alapján az manipulálható [13]. Az ütemezési algoritmus implementációja szempontjából kulcsfontosságú a használata.

A 2.3. ábrán látható `dev_queue_xmit` függvény az, amely megállapítja egy csomagról, hogy kiküldhető-e a hálózatra, azaz kompatibilis-e a fragmentálása a hálózati kártyával, illetve hogy az ellenőrző összeg megegyezik-e [1]. Ezután, ha már van egy ütemezési sor az eszközhöz rendelve, a csomag `sk_buff` struktúrája hozzáadódik ehhez a sorhoz (egy `_enqueue_` függvény segítségével), majd a `qdisc_run` függvény meghívásával előidézzük egy csomag kiküldését belőle. Az imént befűzött csomag valószínűleg nem hagyja el azonnal a puffert, mivel azonban a sorhoz már hozzáadódott, később biztosan ki fog belőle fűződni [13].

Minden hálózati eszközhöz tartozik egy gyökér típusú sorba rendezési diszciplína.

Ennek a *qdisc*-nek (queueing discipline) lehetnek levelei, melyeket osztályoknak nevezünk és mindegyik osztály saját sorba rendezési diszciplínát használhat, melyek együttesen egy fát alkotnak.

2.4.2. Hálózati ütemező algoritmusok a Linux kernelben

A Linux kernelben az alapértelmezett sorba rendezési diszciplína a `pfifo_fast` névre hallgat, amely a priorizált *fifo* csomagütemező. Ezt az ütemezési algoritmust használva az első csomag, ami a pufferbe érkezik, lesz az első, amely a várakozási ideje leteltével később elhagyja azt.

A kernelben ezen kívül nagyon sokféle ütemezési algoritmus van implementálva, melyeket közül néhányat ebben a részben mutatok be. Azokat az algoritmusokat válaskernelztottam ki, amelyek megértése és használata nélkülözhetetlen volt a feladatom elkészítése során.

Mindegyik algoritmus egy-egy *qdisc*-ként van jelen a kernelben és modulként bármikor betölthető. A sorba rendezési diszciplínákat két csoportba tudjuk besorolni: az osztály nélküli- (*classless*), illetve osztályba tartozó (*classful*) diszciplínákba [7].

Osztály nélküli diszciplínák

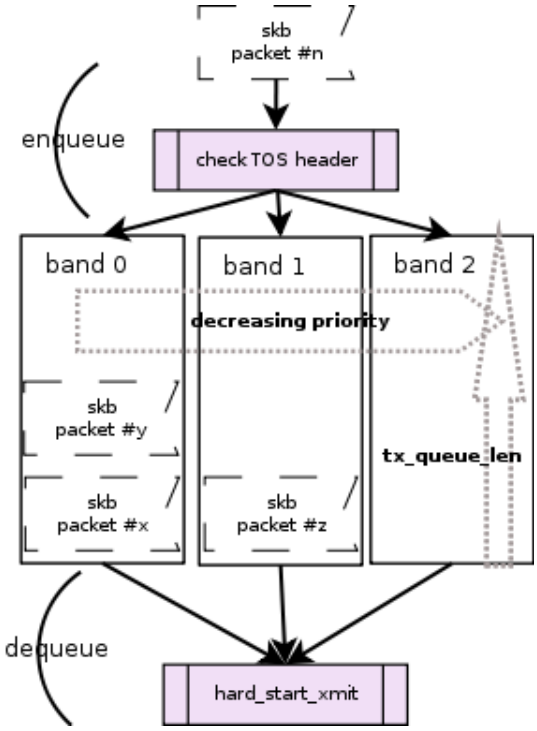
Az osztály nélküli diszciplínák után nem lehet más ütemezőt kapcsolni, tehát nem lehet a kiindulópontja egy másik csomópontnak. Az viszont nincs minden esetben megtiltva, hogy egy osztályba tartozó ütemezési fa leveleire osztály nélküli *qdisc* kerülhessen. Az osztály nélküli diszciplínák legtöbb esetben csak újraütemezésre, késleltetésre, vagy eldobásra ítélhetik a csomagokat [7].

PFIFO_FAST (`sch_fifo`). A `pfifo_fast` tehát az alapértelmezett ütemezési algoritmus a Linux kernelben, akkor van használatban, amikor semmilyen másik diszciplínát nem definiáltunk explicit módon. Tehát kijelenthetjük azt, hogy az esetek túlnyomó többségében ezt használja a Linux rendszer.

Fejlettebb, mint egy egyszerű *fifo* ütemező, mert három sáv van benne megvalósítva, melyek párhuzamosan dolgoznak (2.4. ábra). Ezek a sávok rendre a 0, 1, illetve 2 számozást kapták és szekvenciálisan ürülnek ki: amíg 0 nem üres, addig az 1-es sáv nem lesz feldolgozva és így tovább, azaz a 2-es számú sáv lesz az utolsó, ahonnan kikerülhetnek a csomagok [7]. Tulajdonképpen tehát háromféle prioritást állítható be ezzel a módszerrel. A DiffServ keretrendszer működése is importálva van ebbe az ütemezési algoritmusba, ugyanis a kernel a Type of Service (ToS) mezőt használja arra, hogy megállapítsa, hogy a három sáv közül melyikbe kerüljön a befűzendő csomag [1].

SFQ (`sch_sfq`) – Stochastic Fairness Queueing. Az SFQ algoritmus privilégiumok adása nélkül osztja meg a sávszélességet. A megosztás fair, mert sztochasztikus, de véletlenszerű is lehet. A csomagok IP fejlécéből létrehozható egy-egy *hash* függvénnyel egy szám, amely alapján 1024 vödörbe (*buckets*) osztható minden csomag, amit később *round robin* módon ürít ki az ütemező.

Az az előnye, hogy egyik csomag sem élvez elsőbbséget a másik előtt. A sávszélesség megosztása majdnem mindig fair marad, kivéve néhány esetet, mint például azt, ha két különböző csomagnak valahogyan azonos *hash* számot sikerül kiosztani.



2.4. ábra. A PFIFO_FAST ütemező működése [1]

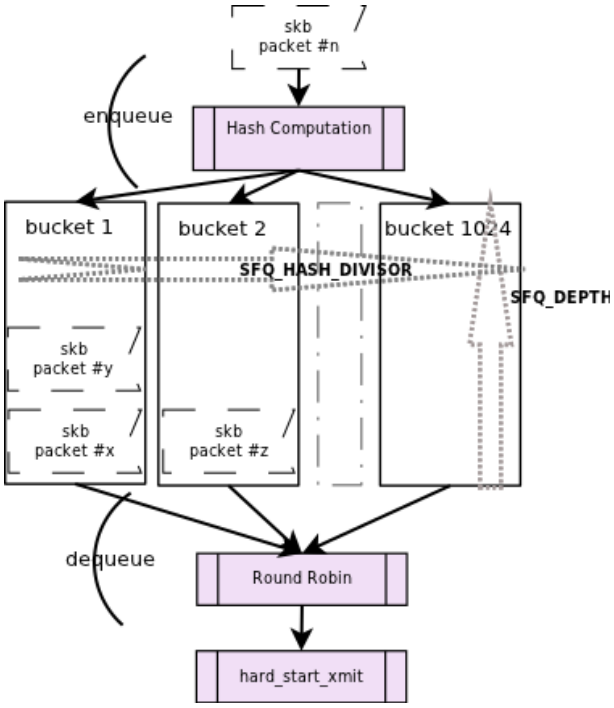
Így ugyan abba a vödörbe kerül a két csomag, vagyis az egyik sokkal hamarabb fog feltöltődni, mint a többi, megtörve ezzel az algoritmus igazságosságát. Ezt úgy kerüli el az SFQ, hogy alapértelmezett módban 10 másodpercenként megváltoztatja a *hash* algoritmust.

A 2.4. ábrán látható, hogy hogyan dolgozza fel a csomagokat az SFQ ütemező. A `tc` parancs meghívásakor meg kell adnunk, hogy az SFQ vödrei mekkorák legyenek (*limit* kapcsoló, maximum 128 lehet), hogy milyen gyakran frissüljön a *hash* algoritmus (*perturb* kapcsoló másodpercben), illetve, hogy mi a maximum mennyisége azoknak a bájtoknak, amiket a *round robin* kifűző folyamat kifűzhet egy vödörből (*quantum* kapcsoló). Ez a szám általában nem lehet kisebb, mint az MTU (Maximum Transmission Unit), mert akkor fent állhatna olyan eset is, hogy olyan méretű csomag van a pufferben, amit az SFQ algoritmus egyszerűen nem lenne képes kifűzni [1]. Emiatt tehát előbb-utóbb beragadna az ütemező és egyetlen csomag sem jutna át rajta.

TBF (`sch_tbf`) – Tocken Bucket Filter. Amikor a sávszélességet szeretnénk kontrollálni az a legfőbb problémánk, hogy hogyan találjunk effektív könyvelési módszert a méretek számontartására. A memóriában számolást ugyanis valós időben megoldani nehéz [1].

A csomagok számolása helyett a TBF algoritmus bizonyos időközönként küld egy *token*t egy verembe (pufferbe). Ez a módszer teljesen független attól, hogy éppen van-e csomagtovábbítás vagy nincs. Amikor egy csomag beérkezik az ütemezőbe, akkor a méretétől függően egy bizonyos mennyiségű tokent fog felhasználni, hogy kifűzhető állapotba kerüljön. Ha nincs elegendő *token* számára, akkor várnia kell.

Az a frekvencia határozza meg az átviteli sebességet, az átviteli rátát, amivel



2.5. ábra. Az SFQ ütemező működése [1]

a *token*ek hozzáadódnak a veremhez. A TBF másik jellemzője, hogy megengedi a börsztölést. Azaz, ha egy verem teljesen megtelt, mert nincsen éppen csomagtovábbítás, akkor a következő beérkező csomagoknak nem kell várniuk, azonnal továbbíthatók mindenféle limitáció nélkül, amíg a verem ki nem ürül. A TBF-ben a *burst* paraméter határozza meg a vermek méretét.

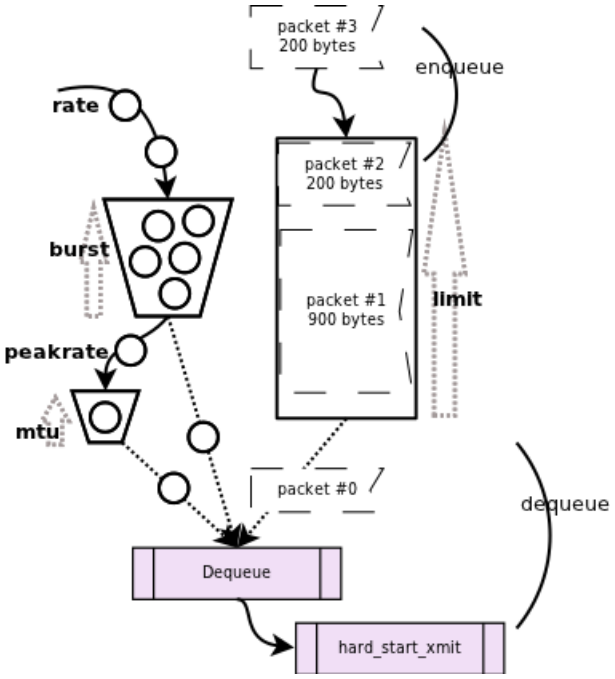
Nagy méretű *burst* értékkel, illetve mindenféle limit nélkül azonban a csomagok átrobognak az ütemezőn. Ezért a TBF-ben implementálva van egy második verem is, ami kisebb, általában akkora mint az MTU mérete. Nem tud tehát sok *token*t tárolni, de sokkal hamarabb újra tud töltődni, mint a nagy verem. Ez a második ráta a csúcsráta (*peakrate*) és a *burst* maximális sebességét fogja meghatározni.

Tehát ha minden jól működik, akkor a csomagok be- és kifűzése a csúcsráta sebességével történik. Ha vannak *token*ek, amikor egy csomag beérkezik a TBF-be, akkor ez a csomag a csúcsráta sebességével továbbítódik, egészen addig, amíg az első verem üres. Ezt láthatjuk a 2.6. ábrán is.

Ha egy interfészt le szeretnénk lassítani, akkor arra a célra a TBF az egyik legprecízebb *qdisc* implementáció és a processzort sem terheli le túlságosan [7].

Osztályba tartozó diszciplínák

Az osztályba tartozó *qdisc*-ek akkor nagyon hasznosak, ha különböző típusú forgalmaink vannak, melyek különböző bánásmódot igényelnek. Fákba rendezhetjük őket, így képesek leszünk a forgalmat a QoS paraméterek alapján alakítani, manipulálni. A következő részben a teljesség igénye nélkül csak két algoritmust ismertetek, melyeknek szerepük lesz az egyedi ütemezési algoritmus implementációjában.



2.6. ábra. A *tbf* ütemező működése

PRIO (`sch_prio`). A PRIO algoritmus nem alakítja a forgalmat, csak felosztja azt különböző, előre konfigurált szűrők alapján. A PRIO ütemezési diszciplínát úgy is felfoghatjuk, mint egy továbbfejlesztett PFIFO_FAST algoritmust, ugyanis minden sávja egy elszigetelt osztály, ahelyett, hogy egy egyszerű *fifo* sor lenne.

Ha egy csomag a PRIO ütemezési diszciplínával fűződik be, akkor kiválasztódik egy osztály az általunk megadott szűrők által. Alapbeállításban három osztály jön létre, melyek felhasználhatóak bármely *qdisc* csatolására. Ha egy csomagot ki szeretnénk fűzni a PRIO-ból, akkor kezdetben az első osztályból próbál kifűzni a diszciplína. Magasabb osztályokat csak akkor használ, ha az alacsonyabb sávok nem adnak fel csomagot [7].

A PRIO *qdisc*-et akkor érdemes használni, ha úgy akarunk csomagokat priorizálni, hogy nem tartjuk elegendőnek azt, hogy csak a ToS bitek használjuk fel hozzá. Ebben az esetben a `tc` szűrők további lehetőségeket teremtenek számunkra. A PRIO ütemező nagy a munkaigényű, így csak akkor ajánlatos alkalmazni, ha a fizikai hálózatunkban viszonylag sok forgalmi igény érkezik be. De akkor is ajánlott a használata, ha egy másik diszciplínával (pl. HTB) már előtte lekorlátoztuk a forgalmat.

A *bands* kapcsolóval megváltoztathatjuk az előre definiált három sáv mennyiségét, de ha változtatunk ezen, akkor a *priomap* kapcsolót is alkalmaznunk kell, amely a ToS mező alapján állapítja meg a csomagprioritások alakulását.

HTB (`sch_htb`) – Hierarchical Token Bucket. A Hierarchical Token Bucket a TBF továbbfejlesztett változata, amely az osztályok fogalmát is bevezeti. Minden osztály egy TBF-szerű *qdisc*nek tekinthető, melyek együtt fává állnak össze. A fának van gyökere és levelei, melyek kilépési pontként csatolhatók a fához. A HTB bevezeti az osztályok közötti priorizálást és a lehetőséget arra, hogy az osztályok sávszélességet kölcsönözzenek egymástól.

A forgalomszabályozó paraméterei a TBF-ből ismert *burst* és *rate*, illetve az SFQ-

ban tárgyalt *quantum*. Az új paraméterek a *ceil* és *cburst*, melyek közül az előbbi azt szabja meg, hogy mennyi plusz sávszélességet kölcsönözhet az adott ág a többitől, az utóbbi pedig a *ceil* értékéhez tartozó *burst* [1].

Modernebb, újabb kernelek esetén a CBQ (Class Based Queueing) helyett ezt a diszciplínát érdemes használni, ha több különböző folyamat miatt részletekre szeretnénk felosztani a maximális sávszélességet, mindegyik ágon garantált értékkel [7].

Az osztály alapú ütemezés (CBQ) tárgyalásától a dolgozatomban eltekintek, mert ez a legkomplexebb és legnehezebben alkalmazható diszciplína. Ezen kívül az algoritmusának realizálása a Linux környezettől idegen [7].

3. fejezet

Az új ütemezési algoritmus implementálása

Az Ericsson Traffic Lab laboratóriumában folyó kutatásokhoz kapcsolódóan célul tűztem ki egy új ütemezési algoritmus implementálását és tesztelését, melynek megvalósítása során egy olyan diszciplínát realizáltam, amely képes a különböző felhasználókhoz tartozó csomagokat különálló várakozási sorokban kezelni és soronként egyedileg menedzselni.

Ez biztosítja azt a differenciálást, ami alapján a különböző igényeket elváró alkalmazások megkülönböztetett kiszolgálásban részesülhetnek. A megkülönböztetést az IP csomag fejléc 8 bites Type of Service (ToS) mezőjének küldő oldalon való beállítása fogja jelenteni.

3.1. A koncepció

A struktúrát előzetes feltevés szerint 32 eszköz használja, interneten lévő videó-adat letöltésére és megnézésére. A hálózati ütemezési algoritmus tehát az internet és az eszközök közötti *switch* kliensek felé menő irányában lévő interfészén léphet működésbe. Az architektúra felépítése miatt csak belső, privát hálózaton működtethető a diszciplína.

Az eszközök mindegyikéhez egyenként négy darab, egymástól független várakozási sor tartozik. Így kapunk végül 128 sort, melyekben kernel szinten teljesen független, egymást nem átfedő ütemezési mechanizmusok mennek végbe. Ezek a sorok lesznek az egyenként és egyedileg jól menedzselhető, könnyebben szabályozható ütemezők.

Az IP fejléc 8 bites Type of Service (ToS) mezőjét felhasználva minden IP csomag befűzése során bekategorizálható, hogy a 128 sor közül melyikbe, illetve a soron belül hova kerüljön: annak végére, elejére, vagy közepére. Ezen kívül minden befűzött csomag kap egy súlyozást is, amelynek nagysága a kifűzés sorrendjét hivatott eldönteni. Előzetes megállapodás szerint a ToS mező 8 bitjét a következő módon osztottuk fel:

- 0-1 bit: `subQueueID` (2 bit)
- 2-3 bit: `delayClassID` (2 bit)
- 4-7 bit: `queueWeightID` (4 bit)

A sorokba történő besorolás egyedül a célállomás IP címétől függ. A célcím utolsó bájtját öt bitre maszkolva egy 0 és 31 között számot kapunk, melyek a 32 sort

azonosítják. Ha megvan a várakozási sor száma, akkor a ToS mező `subQueueID`-ja dönti el, hogy az adott eszköznek fenntartott négy lehetséges alsor közül melyik pufferébe kerüljenek a csomagok. A 0 és 3 közötti négy lehetőség azonosítja az alsort.

A Type of Service mező következő két bitje határozzák meg a `delayClassID`-t, ami a soron belüli pozíció elfoglalását véglegesíti. A 0-ás és 2-es értékek esetén a már meglévő sor végére, 1-es és 3-as érték esetén pedig az elejére kerül be a csomag. Ezeket a csomagokat a várakozási sorok pufferében a kernel láncolt listában tárolja, tehát a befűzés a listákba való befűzést jelenti.

A befűzött sorok minden esetben egy súlyozási értéket kapnak, szintén a ToS mezőjük alapján. Ezt az értéket a nyolc bites mező utolsó négy bitje határozza meg. A súlyozás nagysága pedig a kifűzés sorrendjét állapítja meg.

Így valósul meg tehát a befűzési mechanizmus az új algoritmusban. A kifűzés egyszerűbben alakul, ennek során ugyanis mindig a várakozási sor végéről fűzzük ki a csomagot. Ennek a sornak a kijelölése pedig a következőképpen zajlik: mindig annak a várakozási sornak a végéből fűzünk ki, amelyik sorban a legnagyobb súlyú csomag található.

A következő kifűzési ciklusban a maximális súly vizsgálatát az előzőleg kifűzött sor utáni sorból kezdjük, nem kiéheztetve így egy olyan várakozási sort, amiben sok nagy súlyú csomag sorakozott fel, azaz ekkor egy, a *round robin* működéshez hasonló algoritmust követünk.

A súlyozás megállapítása során egész számokat kellett alkalmaznom, ugyanis a kernel nem kezeli a `float` típusú változókat, illetve az egyszerűbb átláthatóság kedvéért csak pozitív számokkal akartam dolgozni. Így a legkézenfekvőbb megoldás kettő hatványainak alkalmazása volt, úgy, hogy a normális, semleges súly a 128 lett, azaz a 16 lehetőség közül az egyik középső, a 28. A 128 alatti számok tehát kisebb, míg értelem szerűen az annál nagyobb számok nagyobb súlyokat jelentenek. Így született meg a 3.1. táblázatban látható súlyozás a 4 bites `queueWeightID` felhasználásával.

ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Súly	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	Inf.

3.1. táblázat: A *queueWeightID* és a súlyok összerendelése

3.2. Implementálás

3.2.1. Az előkészületek

Az algoritmus leprogramozásához először ki kellett választanom, hogy melyik kernel verzióban valósítsam meg az ütemezőt. A projekt kezdetekor a 3.12-es Kernel volt a legfrissebb stabil verziójú, hosszú távon támogatott kernel, így amellett döntöttem. A forráskód letöltése után a kernelt lefordítottam, a forrásfájlokat pedig Eclipse-ben indexeltem. A modulok fordítását terminálon keresztül hajtottam végre a projekt során.

A Linux kernel `C` nyelven íródott a világ talán legjobb programozóinak összefogásával. Viszont a kernel kódjában a hibakeresés nehezen megvalósítható folyamat, legegyszerűbb és legkézenfekvőbb módszer rá a kód teletűzdelése `printk` függvényekkel, amely a `printf` kiíratófüggvény megfelelője a kernel térben.

Ezzel a módszerrel tudtam megérteni a kernelben már leprogramozott és meg-

bízhatóan alkalmazható hálózati ütemezők működését. Munkám során mindig az eredeti kódból indultam ki, és ügyeltem arra, hogy működő kódból működő kódot készítsek, azaz kis lépésekben haladjak. Ellenkező esetben ugyanis megnő a kernelpánikok száma és nehezebben detektálhatóvá válik az elrontott kódrészlet is. A kernel pánikokat elkerülni bár nehezen lehet, de a lefagyásokat és a rendszer újraindulását lerövidíthetjük, ha virtuális gépen fejlesztünk.

Az implementáció során a Mininet [9] nevű hálózat-emulátor keretrendszerben teszteltem rendszeresen az ütemező működését. Ezzel a módszerrel virtuális hosztokat és egy *switch*-et tudtam létrehozni, melynek interfészére a `tc` (trafic controll) programmal voltam képes az általam készülő ütemezőt felcsatolni és megállapítani ezzel, hogy a modul egyáltalán működőképes-e az előző változtatás óta. A hibakeresés hiánya miatt a tesztelés ezen módja jelentős mértékben növelte az implementációra szánt időt. (A Mininet keretrendszerre és a tesztelés módjára a következő fejezetben fogok részletesebben kitérni.)

3.2.2. Kész ütemezési algoritmusok vizsgálata

A második fejezetben már említett ütemezési algoritmusokat megvizsgáltam, hogy melyik az, amely működésben a legközelebb áll a felvázolt koncepcióhoz. Így esett a választás a PFIFO_FAST és a PRIO nevű ütemezőkre.

Az `sch_fifo.c` kódját tanulmányozva megértettem a *socket buffer*ek (`sk_buff`) használatát, illetve azok láncolt listába fűzésének mechanizmusát. Az `sch_prio.c` ütemező kódjában pedig láthattam, hogy hogyan lehet a kernelben több ütemezési sort is lefoglalni, megcímezni és használni.

A két forráskód alapján több sarokpontot véltem felfedezni. Ilyen például a *qdisc* inicializálás folyamata, a sorok helyfoglalásának módja, a *socket buffer*ek struktúrája és belőlük az IPv4 csomagok kivétele, illetve az `sk_buff` láncolt listák felépítése. Ezeken kívül a befűzési és kifűzési függvényekről is szót ejtek, valamint egy olyan globálisan definiált struktúráról is, amibe az algoritmus helyes működéséhez szükséges adatokat lehet elmenteni.

3.2.3. Az új ütemezési algoritmus implementálása

Az implementáció során az `sch_prio.c` kódjából indultam ki és formáltam át a specifikációknak megfelelően. Az így létrehozott saját ütemezési algoritmusra a továbbiakban SCHEDULER ütemező néven fogok hivatkozni.

A modul inicializálása

A Linux kernelben a hálózati ütemezők modulként vannak megírva. A modulok beillesztése során (`insmod sch_scheduler.ko`) a `module_init()` függvény hívódik meg, kivételük esetében (`rmmod sch_scheduler`) pedig a `module_exit()`. A modul inicializálása során meghívódik a `scheduler_module_init(void)` függvény, amely regisztrálja a *qdisc*-et a `register_qdisc(&scheduler_qdisc_ops)` hívással. A `register_qdisc()` egy `Qdisc_ops` típusú struktúrát tölt fel, melynek adattagjait a következő szintaktikával éri el az `sch_scheduler.c` esetén:

```
static struct Qdisc_ops scheduler_qdisc_ops __read_mostly = {
        .next = NULL,
```

```
        .cl_ops = &scheduler_class_ops,
        .id = "prio",
        .priv_size = sizeof(struct scheduler_sched_data),
        .enqueue = scheduler_enqueue,
        .dequeue = scheduler_dequeue,
        .peek = scheduler_peek,
        .drop = scheduler_drop,
        .init = scheduler_init,
        .reset = scheduler_reset,
        .destroy = scheduler_destroy,
        .change = scheduler_tune,
        .dump = scheduler_dump,
        .owner = THIS_MODULE,
};
```

A fentiekben látható, hogy az ütemező minden feladatához külön függvény hívódik meg. A projekt szempontjából legjelentősebb az `.init`, a `.priv_size`, az `.enqueue` és a `.dequeue` adattagok tárgyalása.

A `prio_sched_data` deklarálása

Az `sch_prio.c` kódban globálisan definiálva megtalálhatjuk a `prio_sched_data` struktúrát. A SCHEDULER *qdisc* esetében néhány adattagot meg kellett változtatni az eredeti `sch_prio.c-hez` képest, mely után a következő eredmény született:

```
struct scheduler_sched_data {

        struct Qdisc *queues[128];

        int maxWeight[128];
        int sizeBytes[128];
        int sizePackets[128];

        int sumSizeBytes;
        int sumSizePackets;
        int lastServedQueueID;

};
```

Az `scheduler_sched_data` struktúrában megtalálható a 128 egyedi ütemezési sor deklarációja, ezen kívül olyan `int` tömbök, melyek minden sorhoz eltárolják az adott sorban lévő csomagok közül a legnagyobb súlyút, illetve az egész sor méretét bájtokban, és a csomagok számát az adott sorban. Egyéb változók még a 128 sorban lévő összes csomag mérete bájtban, az összes csomag mennyisége és végül egy szám, ami mindig elmenti a legutóbbi kifűzés sorának számát. Erre az *round robin*-szerű működés miatt van szükség, melyről a kifűzés (*dequeue*) tárgyalása során írok részletesebben.

A fenti adatstruktúrában létrehozott `queues[128]` tömbnek helyet is kell foglalni, hogy megfelelően működjön az ütemezőnk. Ahogy az elméleti részben említettem,

a PRIO *qdisc* használata során sávok (*bands*) jönnek létre a kernelben, melyek függetlenül működő, előre definiált osztályokként használhatók. A SCHEDULER *qdisc* esetén is pontosan erre van szükség, csak nem három, hanem 128 darab sávra, melyek méretét az `scheduler_qdisc_ops .priv_size` adattagja tárolja a kódban.

A SCHEDULER *qdisc* inicializálása és az ütemezési sorok helyfoglalása

A *qdisc* inicializálása az `.init` adattag betöltődésével megy végbe, amely meghívja az `scheduler_init()` függvényt. Ezen belül fog lefutni a struktúra számára a helyfoglalása.

A `scheduler_sched_data` struktúra méretfoglalását az `.init` adattag meghívása után a `qdisc_priv(sch)` függvény teszi meg. A létrehozott példányt (`q`) azonban fel is kell tölteni a kezdeti értékekkel, mely során az `int`-ek definiálása egyértelmű, a 128 soré viszont kevésbe: ekkor üres sorokkal kell egyenlővé tennünk a tömbünk minden sorát a következőképpen: (`q->queues[i] = &noop_qdisc;`).

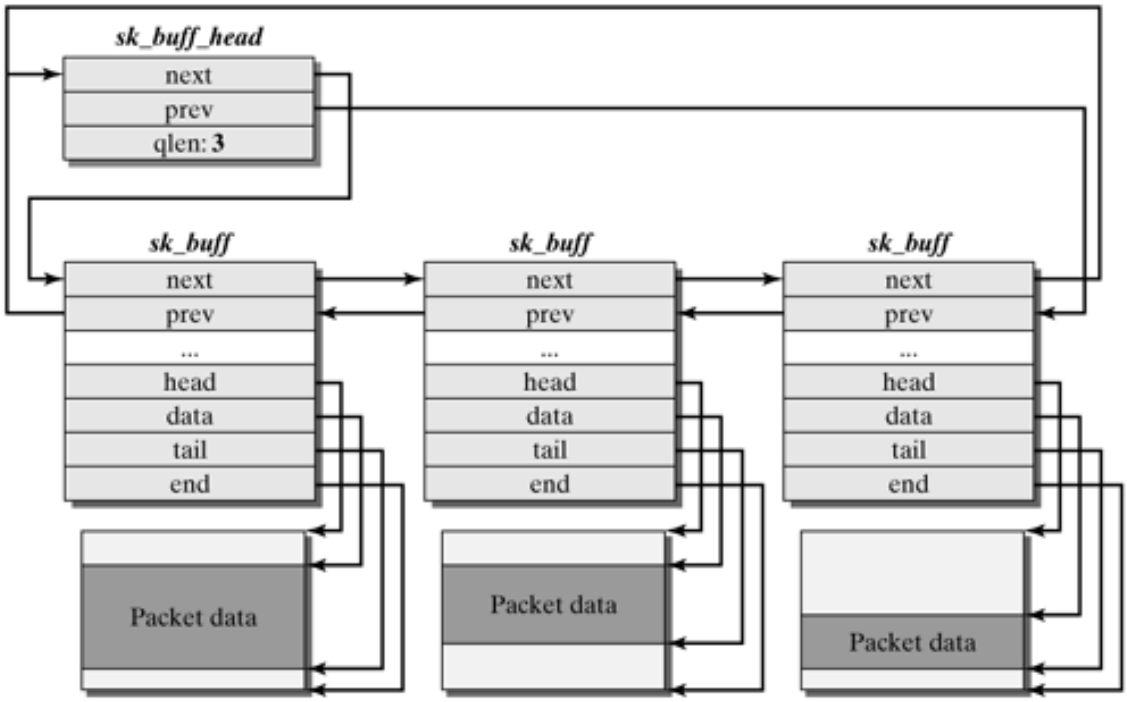
```
static int scheduler_init(struct Qdisc *sch, struct nlattr *opt)
{
        struct scheduler_sched_data *q = qdisc_priv(sch);
        int i;
        for (i = 0; i < 128; i++) {
                q->maxWeight[i] = -1;
                q->shareByte[i] = 0;
                q->sizeBytes[i] = 0;
                q->sizePackets[i] = 0;
        }

        q->lastServedQueueID = -1;
        q->sumSizeBytes = 0;
        q->sumSizePackets = 0;

        for (i = 0; i < 128; i++) {
                q->queues[i] = &noop_qdisc;
        }

        if (opt == NULL) {
                return -EINVAL;
        } else {
                int err;
                if ((err = scheduler_tune(sch, opt)) != 0) {
                        return err;
                }
        }
        return 0;
}
```

A `Qdisc` struktúrák, vagyis az ütemezési sorok helyes foglalását az `scheduler_tune()` függvény hívása teljesíti be, amely a `prio_tune()` [5] mintájára készült.

Socket buffer struktúrák és a láncolt listák felépítése

A *socket buffer* struktúrák (`struct sk_buff`) azok a tárolók, amelyekben a Linux kernel a hálózati csomagokat kezeli. Ha egy csomag beérkezik a hálózati kártyához, a hálózati eszköz vezérlője lefoglal neki egy `skb`-t a `netdev_alloc_skb()` függvény meghívásával [13].

Az `sk_buff` struktúrák láncolt listaként kapcsolódnak egymásba az ütemezési sorban, azaz egy `Qdisc` struktúra példányban. Minden példányban található egy `sk_buff_head` struktúra, mely az első és utolsó eleme is egyben az `sk_buff`-ok láncolatának. A *socket buffer* struktúrák a struct `sk_buff *next` és a `struct sk_buff *prev` mutatókkal kapcsolódnak össze. Ez felépítés a 3.1. ábrán tekinthető meg.



3.1. ábra. Az `skb` struktúrák láncolata [13]

Az `sk_buff` struktúrából érhető el többek között az IP csomag fejléce is. A SCHEDULER *qdisc* implementációja során az IP fejléchez, azon belül is a célcímhez, illetve a ToS mezőhöz való hozzáférés kulcskérdés, ezt a következő kódrészlettel valósíthatjuk meg:

```
struct iphdr* iph;
int destIP;
int ToS;
iph = ip_hdr(skb);
destIP = iph->daddr;
ToS = iph->tos;
```

Enqueue (befűzés)

A fenti kódrészletek bemutatása után rá lehet térni az algoritmus lényegi kérdéseire: mi alapján derül ki, hogy melyik ütemezési sorba kerül be a befűzendő csomag, illetve milyen algoritmus alapján és honnan fűződik ki az ütemezőt elhagyni készülő adatcsomag?

A befűzést a `Qdisc_ops` struktúra `.enqueue` adattagja, az `scheduler_enqueue` függvény hívja meg, mely két bemeneti paramétert vár: az `sk_buff` *buffer*t, és az `sch Qdisc`-et, vagyis egy ütemezési sort. Egyszerűbben fogalmazva ez a függvény felelős azért, hogy egy adott IP csomag egy adott ütemezési sorba befűződjön.

A befűzéshez szükséges információkat az `scheduler_classify` függvény szolgáltatja. Ez az a metódus, amely az IP fejlécből kiszámolja a koncepció ismertetésekor bevezetett változókat, azaz előállítja a `maskedIP`-t, a `subQueueID`-t , a `delayClassID`-t, illetve a `queueWeightID`-t.

```
int ToSsubQueueID;   // Defines subqueue
int ToSdelayClassID; // Defines position in queue
int ToSqueueWeight;  // Defines weight
int globalQueueID;   // Defines global queue number

ToSsubQueueID = ToS >> 6;
globalQueueID = ((DestIP >> 24) & 0x1F) * 4 + ToSsubQueueID;
ToSdelayClassID = (ToS >> 4) & 0x3;
ToSqueueWeight = (ToS & 0xf);
```

A célcím utolsó bájtjának 5 bitre maszkolásával meghatározhatjuk, hogy a 32 eszköz közül melyikbe kerüljön a csomag. Ha ezt néggyel megszorozzuk és hozzáadjuk az alsor számát, akkor megkapjuk a végső, globális sorazonosítót (`globalQueueID`), egy 0 és 127 közötti számot.

A súlyozás mértékének meghatározásához a fentieken kívül azonban egy globális változóra is szükség van, amely eltárolja a koncepció részben ismertetett táblázat (3.1) értékeit (`TosToWeight[16]`).

```
const int Tos_To_Weight[16] = {0, 1, 2, 4, ... };
```

A fenti konstans `int` tömb alapján tudunk tehát súlyt rendelni minden csomaghoz, melyek közül mindig csak a legnagyobbat mentjük el:

```
int weight;
weight = Tos_To_Weight[ToSqueueWeight];
if (weight > q->maxWeight[globalQueueID]) {
        q->maxWeight[globalQueueID] = weight;
}
```

A befűzési metódusban meghívott `scheduler_classify()` függvény visszatérési értéke annak az ütemezési sornak a mutatója, amibe be kell fűznünk az aktuális beérkező csomagot. Ezen kívül a függvény argumentumában lévő mutatók miatt a soron belüli pozíciót is „átmenthetjük" az `scheduler_enqueue()` függvénybe.

A `delayClassID` arra való, hogy megmondja hova fűződjön be az új csomag a soron belül. Feltételvizsgálattal lehet ezt a legegyszerűbben megoldani. A befűzésen kívül itt kell frissíteni a globális adatokat is az ütemezőről. Ez az alábbiak szerint történik:

```
if (delayClassID == 0) {
  ret = qdisc_enqueue_tail(skb, qdisc);
  q->sizePackets[globalQueueID] = q->sizePackets[globalQueueID] + 1;
  q->sizeBytes[globalQueueID] = q->sizeBytes[globalQueueID]+qdisc_pkt_len(skb);
  q->sumSizePackets = q->sumSizePackets + 1;
  q->sumSizeBytes = q->sumSizeBytes + qdisc_pkt_len(skb);

} else if (delayClassID == 1) {
  ret = qdisc_enqueue_head(skb, qdisc);
  q->sizePackets[globalQueueID] = q->sizePackets[globalQueueID] + 1;
  q->sizeBytes[globalQueueID] = q->sizeBytes[globalQueueID]+qdisc_pkt_len(skb);
  q->sumSizePackets = q->sumSizePackets + 1;
  q->sumSizeBytes = q->sumSizeBytes + qdisc_pkt_len(skb);
}

...

if (ret == NET_XMIT_SUCCESS) {
  sch->q.qlen++;
  return NET_XMIT_SUCCESS;
}
if (net_xmit_drop_count(ret))
  sch->qstats.drops++;
return ret;
```

Dequeue (kifűzés)

A kifűzés annak a sornak a végéből történik, amely a legnagyobb súlyú csomagot tartalmazza éppen aktuálisan. De arra is figyelni kell a *round robin*-szerű működés betartása miatt, hogy mindig megjegyezzük honnan fűztünk ki legutoljára csomagot. Ezt a `lastServedQueueID` változóban tároljuk el és a következő kifűzés során mindig az ennél eggyel nagyobb indexű sorból kezdjük a maximális súlyú sor megkeresését. Amit még fontos megemlíteni, hogy egy sor maximális súlyát vissza kell állítani az alapállapotra, ha az a sor teljesen üres, illetve frissíteni kell a globális változókat.

```
for (ii = q->lastServedQueueID + 1; ii <= q->lastServedQueueID + 128; ii++) {
  id = ii % 128;
  actualWeight = q->maxWeight[id];
```

```
  if (actualWeight >= maxWeight && skb != NULL) {
    skb = qdisc_dequeue_peeked(qdisc);
    q->lastServedQueueID = id;
    maxWeight = actualWeight;
    qdisc_bstats_update(sch, skb);


  if (sch->q.qlen == 0) {
    q->maxWeight[id] = DEFAULT_WEIGHT;
  }

  q->sizePackets[id] = q->sizePackets[id] - 1;
  q->sizeBytes[id] = q->sizeBytes[id] - qdisc_pkt_len(skb);
  q->sumSizePackets = q->sumSizePackets - 1;
  q->sumSizeBytes = q->sumSizeBytes - qdisc_pkt_len(skb);
  return skb;
}
```

4. fejezet

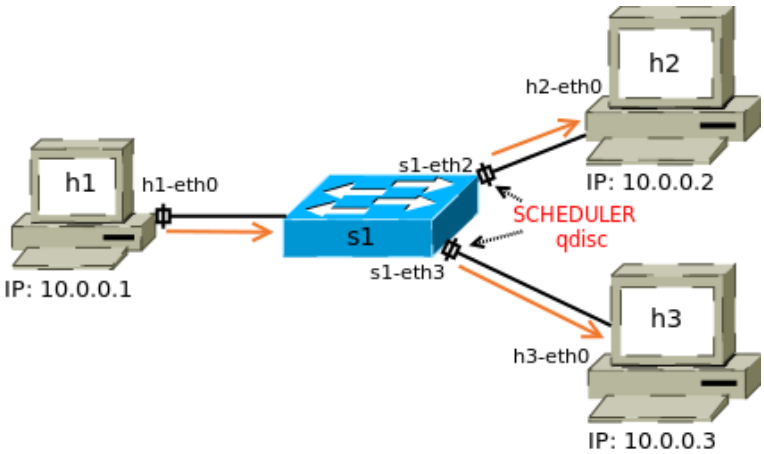
Tesztelés és teljesítményelemzés

Az implementálás során egy már meglévő, működő kódból indultam ki, melyet a kiírt koncepció szerint folyamatosan változtattam. Fontos szempont volt, hogy működő kódból működőt hozzak létre, ugyanis a kernelben való fejlesztés során nem mindig elég pusztán az, ha a fordítás során hibát találunk. Sokszor felmerülhet, hogy a kód ugyan lefordul, viszont az ütemező beüzemelésekor az operációs rendszer magja nem képes tovább futni, vagyis kernelpánik jön létre.

Ahhoz, hogy bármilyen kis változást viszonylag gyorsan tesztelni lehessen, felépítettem egy tesztkörnyezetet, melyben virtuális hosztok virtuális *switch*en keresztül képesek kommunikálni egymással. A választásom egy hálózat-virtualizációs keretrendszerre, a Mininetre esett, amivel realisztikus, valós kernelt futtató emulált hálózatokat lehet megvalósítani.

4.1. Az architektúra

Egy nagyon egyszerű, három hosztból és egy switch-ből álló hálózati architektúrát definiáltam a Mininet Python API [12] segítségével, amely a 4.1. ábrán látható.



4.1. ábra. A megvalósítandó teszthálózat architektúrája

A SCHEDULER *qdisc*-et az `s1` kapcsoló `s1-eth2` és `s1-eth3` interfészein egyaránt üzembe helyeztem. A hosztok között két `nc` (*netcat*) csatorna felépítésével adatforgalmat generáltam a narancssárga nyíllal megegyező irányokban. Két céláll-

omás definiáltam, a h2-t és a h3-at. A h1-es küldőállomás kimenő interfészéről a ToS mezőben manipulált csomagokat küldtem a másik két számítógép felé.

Így figyeltem azt, hogy a csomagok átérnek-e a hálózaton, illetve, hogy abba a sorba kerülnek-e, amely a ToS mezőjük alapján kerülniük kell. Ahhoz, hogy az ütemezőt tesztelni tudjam, le kellett korlátoznom s1-eth2 és s1-eth3 áteresztőképességét a HTB algoritmussal. Azért volt erre szükség, mert egyébként az adatforgalom annyira nagy lett volna, hogy a hálózati ütemező pufferjeinek nem lett volna ideje telítődni és nem látszódott volna az, hogy hogyan működik az algoritmus.

4.2. A SCHEDULER ütemező validálása

Ebben a részben bemutatom a SCHEDULER ütemező validálási eredményeit.

A virtuális tesztkörnyezet elindítása után a *traffic control* program új ütemezőjét, az SCHEDULER *qdisc*-et csatoltam fel az s1-et2 és s1-eth3-as interfészekre egyaránt. A Kernel kódjában elhelyezett printk kiíratófüggvényeknek, és az operációs rendszermag naplózási mechanizmusának köszönhetően látható volt, hogy az ütemezési mechanizmus csatolása mindkét interfészen sikeresen megtörtént.

A virtuális h1-es hoszton tehát két terminál nyitásával egy-egy nc csatornát építettem fel, egymástól eltérő portszámokkal, melyeken keresztül TCP adatforgalmat generáltam és küldtem át. A *netcat* program használata során a [-T] kapcsolóval, hexadecimális számmal megadva közvetlen beállíthatók a kiküldött adatcsomagok Type of Service mezői, így nem kellett a hoszt kimenő interfészére DSMARK *qdisc*-et felcsatolni. (Ez a *qdisc* képes a ToS mezőket különböző szűrők segítségével beállítani.)

Az nc csatornák a h2 (IP cím: 10.0.0.2) és h3 (IP cím: 10.0.0.3) fogadóállomásokon végződtek. A két virtuális gép fogadta a h1 által küldött adatokat. A SCHEDULER *qdisc* algoritmusa szerint (a célcímet és a delayclassID-t felhasználva) a ToS mező beállításait a 4.1. táblázatban olvasható egyedi esetekben teszteltem.

Választott ToS	Célcím	subQueueID	delayClassID	globalQueueID
10 | 0x0A	10.0.0.2	0	0 -> tail enqueue	8
20 | 0x14	10.0.0.3	0	1 -> head enqueue	12
70 | 0x46	10.0.0.2	1	0 -> tail enqueue	9
90 | 0x5A	10.0.0.3	1	1 -> head enqueue	13
130 | 0x82	10.0.0.2	2	0 -> tail enqueue	10
150 | 0x96	10.0.0.3	2	1 -> head enqueue	14
200 | 0xC8	10.0.0.2	3	0 -> tail enqueue	11
220 | 0xDC	10.0.0.3	3	1 -> head enqueue	15

4.1. táblázat. A tesztelésnél használt ToS mezőkből kiszámolt globális sorazonosítók

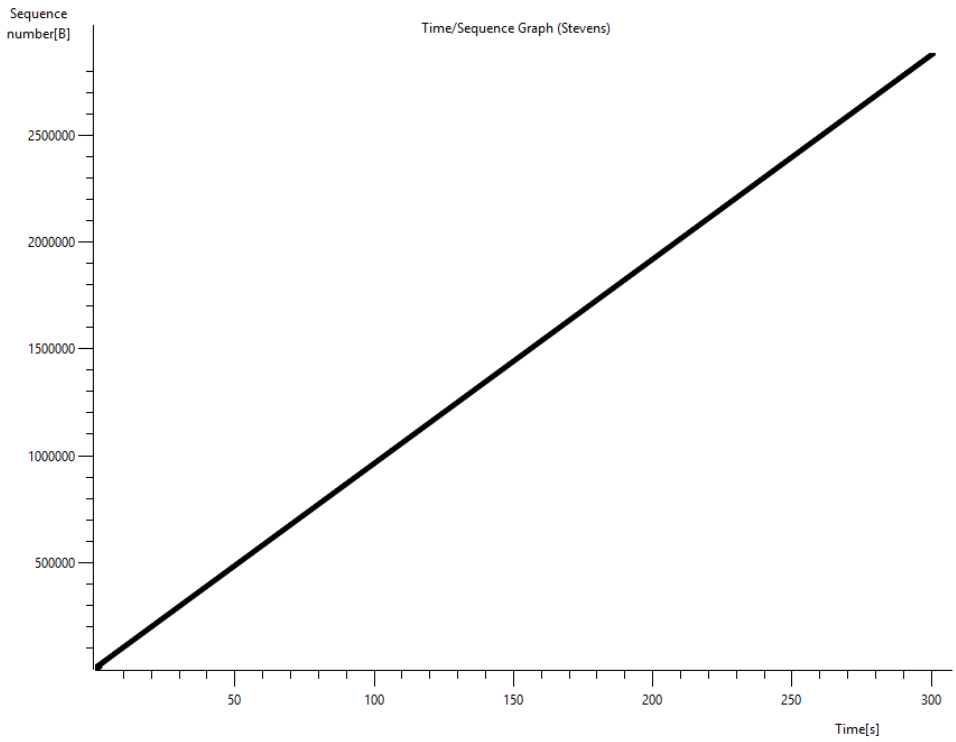
Az adatforgalmakat párosával továbbítottam, vagyis h1 két helyre küldött egyszerre adatot. A Kernel naplózását vizsgálva minden csomag abba a sorba került amelyikbe az algoritmus szerint kerülnie kellett, illetve a sorokon belüli besorolás is a funkcionalitásnak megfelelően ment végbe.

Ezek alapján validáltam a SCHEDULER ütemezőt, amely az elvárásoknak megfelelő helyes működést mutatott.

4.3. Csomagvesztési arány vizsgálata

Ebben a részben a SCHEDULER ütemező teljesítményelemzését végeztem el csomagvesztési arány metrika tekintetében, és az így kapott eredményt összehasonlítottam a HTB és PRIO diszciplínák eredményeivel.

Az előző részben ismertetett teszteléshez hasonlóan változatlanul a 4.1. ábrán látható hálózati architektúra felhasználásával mértem, melynek során a `h1`-es hoszt `h1-eth0` interfészén a Wireshark [16] nevű hálózatanalizáló programmal figyeltem a forgalmat.

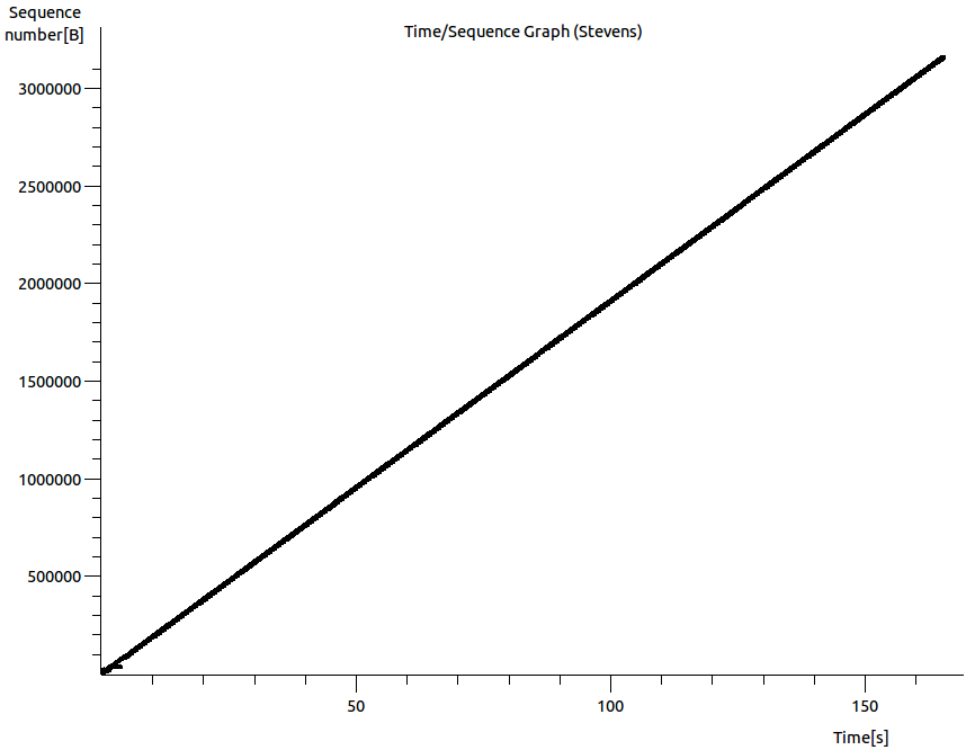


4.2. ábra. A SCHEDULER ütemező Stevens diagramja (80 kbit/s)

A Wireshark programban megtalálható Stevens gráf ábrázolásával a TCP adatfolyamban a két hoszt között sikeresen átmenő csomagok szekvenciaszámait tudtam kirajzolni az idő függvényében. A SCHEDULER ütemező felhasználásával két esetet is megvizsgáltam, melyek Stevens gráfjai a 4.2. és a 4.3. ábrákon láthatók.

Az első esetben a TCP folyamokat a `tcp.analysis.lost_segment` szűrővel vizsgálva, azaz az elveszett csomagok szűrésével nem kaptam találatot, ami azt jelenti, hogy nem volt olyan csomag ami ne ért volna célba.

Összesen nagyjából 12 ezer csomag haladt át a linken, 80 kbit/s-os adatforgalomkorlátozás mellett ez idő alatt. Rossz szekvenciájú (out of order) küldés összesen csak 8 csomagnál volt tapasztalható. Ezekben az esetekben az történik, hogy egy magasabb szekvenciaszámú csomag hamarabb érkezik be, mint egy nála kisebb szekvenciaszámú.A 4.2. ábrán a fentiektől eltekintve sorrendhelyesen érkezett csomagok Stevens diagrammja látható.



4.3. ábra. A SCHEDULER ütemező Stevens diagramja (160 kbit/s)

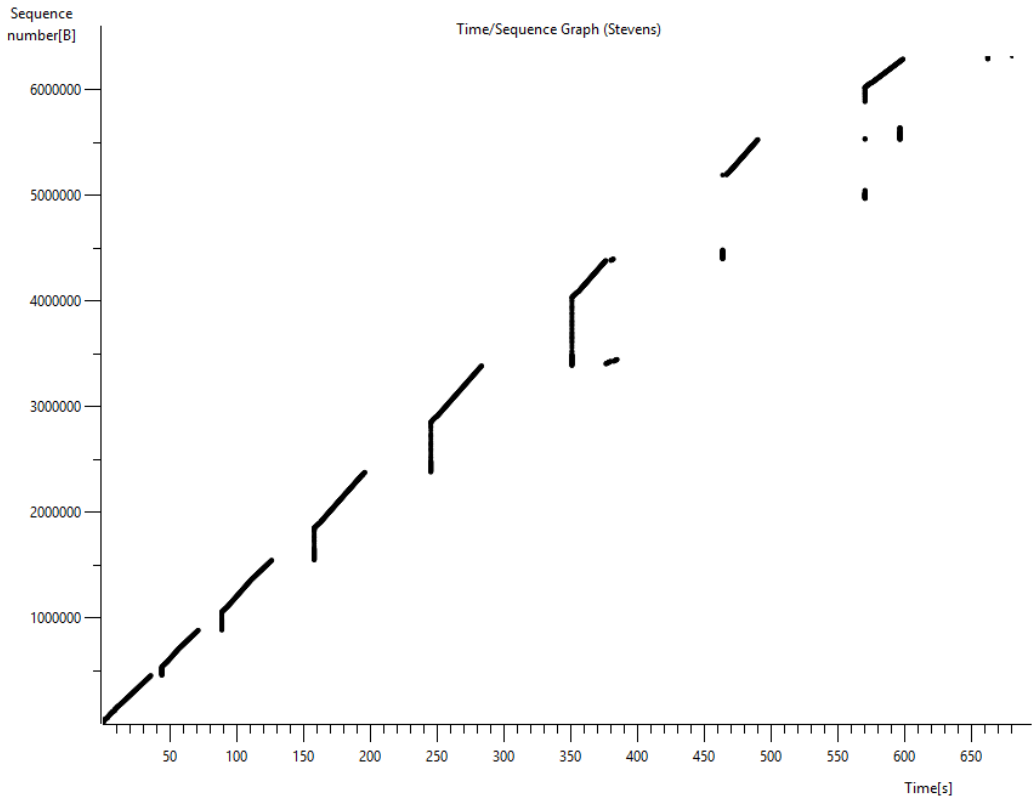
A második esetben megdupláztam a sávszélességet, azaz 160 kbit/s-os adatforgalomkorlátozást vezettem be. Ebben az esetben nagyjából 10 ezer csomagot küldtem át és veszteséget ekkor sem tapasztaltam. A 4.3. ábrán, a Stevens gráfon azonban felfedezhető egy kisebb változás közvetlenül az adatfolyam elkezdése után: néhány csomagot többször is elküldött a `h1`-es hoszt, attól függetlenül, hogy azok megérkeztek-e már a fogadó oldalra.

Várakozásainknak megfelelően az is észrevehető, hogy nagyjából hasonló mennyiségű csomag, mint a 80 kbit/s-os esetben, fele annyi idő alatt ért a céljához a megduplázott sávszélességnek köszönhetően. A Wireshark elmentett adataiból megállapítható, hogy ebben az esetben sem veszett el csomag a TCP folyamból és rossz szekvenciájú (out of order) küldés is csak hat csomagnál volt tapasztalható.

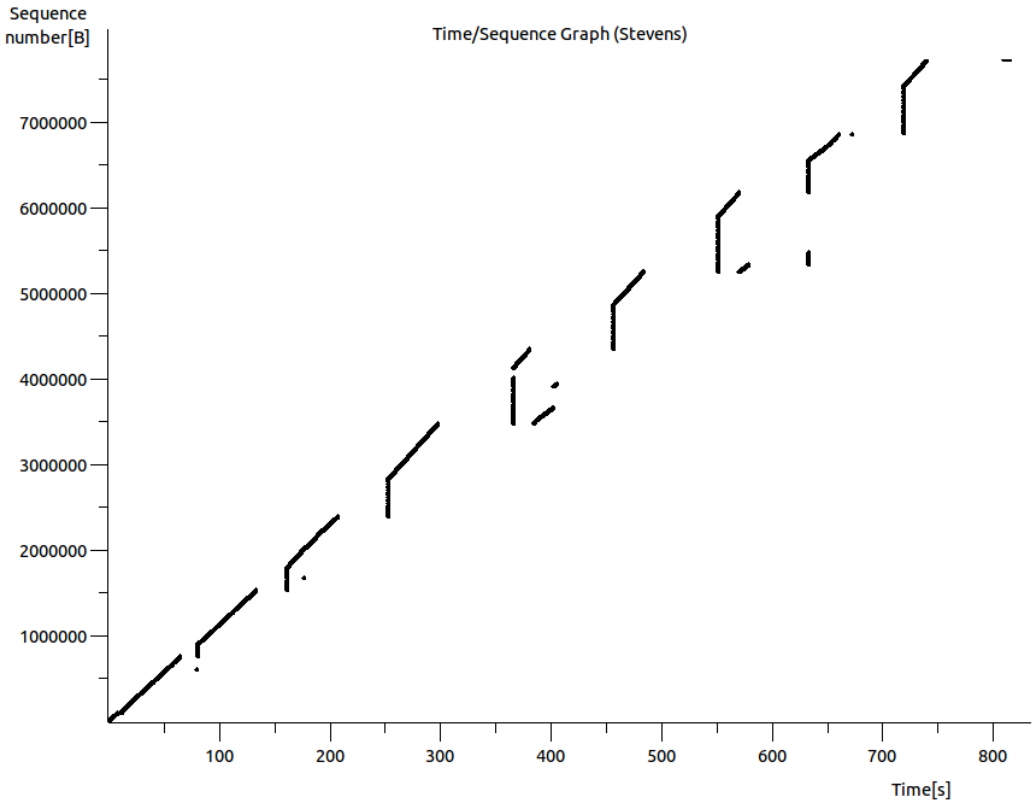
A fenti gráfokat összehasonlítottam a Linux kernelben már implementált HTB (4.4. ábra), és PRIO (4.5. ábra) diszciplínák Stevens gráfjaival is, melyeket szintén 80 kbit/s maximális sávszélesség mellett mértem le. Ezekben az esetekben rosszabb eredményeket tapasztaltam: HTB esetén nagyjából 9000 csomag átküldése során 128 nem érkezett meg egyáltalán a másik oldalra, míg a PRIO algoritmussal ütemezve 11500 csomagból 209 veszett el.

Az ábrákon jól látszik, hogy a SCHEDULER ütemező jobban teljesített, mint a HTB és a PRIO: míg az előbbinél mindkét esetben egy közel lineáris egyenest látunk a képen, addig az utóbbi kettőnél nem folytonos, időben megtörő grafikon ábrázolja a TCP folyam szekvenciaszámait. Ebből egyértelműen látszik, hogy a HTB és a PRIO ütemező elvesztett, és újra is küldött csomagokat.

A fenti teljesítményelemzés alapján megállapítható, hogy a SCHEDULER ütemező az elvárásoknak megfelelő teljesítményt mutat a csomagvesztési arány tekintetében.



4.4. ábra. A HTB ütemező Stevens diagramja



4.5. ábra. A PRIO ütemező Stevens diagramja

5. fejezet

Összegzés és a kutatás lehetséges folytatási irányai

Becslések szerint az évtized végére az internetes forgalom legalább fele videó adatfolyamok megosztására lesz felhasználva. A hálózatokban egyre jobban domináló videó adatforgalmak kezelésére a hálózati ütemezési algoritmusok működését is optimalizálni kell.

Az Ericsson Magyarország Kft. Traffic Lab laboratóriumában folyó ilyen irányú kutatásokhoz kapcsolódva a TDK dolgozatomban egy erre törekvő, újszerű megközelítést mutattam be. A munkám során megismerkedtem a kernelben implementált ütemező algoritmusokkal és a Linux hálózati verem forgalomszabályozási mechanizmusaival. Dolgozatomban megvizsgáltam egy új diszciplína működését, elvégeztem annak implementációját és validációs tesztelését, valamint teljesítményelemzését.

A megvalósult hálózati ütemező képes arra, hogy a különböző prioritású igényeket teljesen egyedileg, egymástól függetlenül kezelje. Ehhez az IPv4 fejlécében megtalálható ToS mezőt veszi alapul, amely a csomag célcímével együttesen határozza meg azt a független ütemező osztályt, amelybe a csomag be fog fűződni. Az osztályból való kifűzés *round robin* szerűen történik, megakadályozva ezzel a sorok kiéheztetését.

Az ütemezési diszciplína a tesztelések során helyesen működött és a csomagvesztési arányai jobbaknak bizonyultak, mint néhány másik vizsgált algoritmusnak.

A kutatási munka folytatásának következő lehetséges lépése az, hogy a Kernel térből *netlink socket*ek megnyitásával információs csatornát állítsunk föl a felhasználói térrel összekapcsolódva. Ekkor az ütemező kódjából el lehetne hagyni a naplózási mechanizmusokat, a felhasználói térben működő programokból kivett adatokból pedig grafikus felületen is lehetne ábrázolni a csomagok fontos adatait és azok be- és kifűzését. Ezzel egy monitorozó rendszert lehetne létrehozni.

Ezen kívül a jövőben az ütemezési diszciplínát valós eszközökön, OpenWRT környezetet futtató útválasztókon is be kellene üzemelni. Ezáltal könnyebbé válna a tesztelés, nem kellene virtuális környezetet alkalmazni hozzá, illetve a végcél is az, hogy az ütemező útválasztókon is helyesen, effektíven és jól működjön.

Rövidítések jegyzéke

3G - Third Generation

4G - Fourth Generation

CBQ - Class Based Queueing

DiffServ - Differentiated Services

DSCP - Differentiated Service Code Point

FIFO - First In, First Out

HD - High Definition

HTB - Hierarchical Token Bucket

IntServ - Integrated Services

IP - Internet Protocol

LTE - Long Term Evolution

Qdisc - Queueing Discipline

QoS - Quality of Service

SFQ - Stochastic Fairness Queueing

TBF - Token Bucket Filter

TC - Traffic Control

TCP - Transmission Control Protocol

ToS - Type of Service

UDP - User Datagram Protocol

Irodalomjegyzék

[1] *Journey to the Center of the Linux Kernel: Traffic Control, Shaping and QoS.* http://wiki.linuxwall.info/doku.php/en:ressources:dossiers:networking:traffic_control, Megtekintve: 2014. október 1.

[2] J. Babiarz, K. Chan, and F. Baker. *Configuration Guidelines for DiffServ Service Classes.* RFC 4594, 2006.

[3] R. Braden, D. Clark, and S. Shenker. *Integrated Services in the Internet Architecture: an Overview.* RFC 1633, 1994.

[4] J. Cimen. Packet Scheduling in Wireless Network IEEE 802.11 Using Linux. Master's thesis, Umeå University, 2004. Megtekintve: 2014. szeptember 28.

[5] Free Electrons. *Linux Cross Reference.* http://lxr.free-electrons.com/source/net/sched/sch_prio.c#L166, Megtekintve: 2014. szeptember 3.

[6] Ericsson. *Ericsson Mobility Report.* http://www.ericsson.com/res/docs/2014/ericsson-mobility-report-june-2014.pdf, 2014. Megtekintve: 2014. október 18.

[7] B. Hubert. *Linux Advanced Routing and Traffic Control HOWTO.* http://lartc.org/howto/index.html, Megtekintve: 2014. szeptember 3.

[8] C. Knowlton. *Growing Demand for High-Quality Video Streaming Across Multiple Devices.* http://www.wowza.com/blog/growing-demand-for-high-quality-video-streaming-across-multiple-device, 2014. Megtekintve: 2014. október 20.

[9] B. Lantz, B. Heller, and N. McKeown. *A Network in a Laptop: Rapid Prototyping for Software-Defined Networks.* ACM, 2010. Megtekintve: 2014. október 11.

[10] G. Lencse and L. A. Csehi. *IP minőség: Oktatási segédanyag.* http://www.tilb.sze.hu/tilb/targyak/NGM_TA011_1/IP_minoseg.pdf, Megtekintve: 2014. szeptember 30.

[11] OpenWRT and Wireless Freedom. *Linux Packet Scheduling.* http://wiki.openwrt.org/doc/howto/packet.scheduler/packet.scheduler.theory, Megtekintve: 2014. október 08.

[12] Topo Class Reference. *Mininet Python API Reference Manual.* http://mininet.org/api/index.html, Megtekintve: 2014. szeptember 5.

[13] R. Rosen. *Linux Kernel Networking: Implementation and Theory*. Apress, 2013.

[14] Cisco Systems Inc. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2013–2018*. `http://www.ciscservice-provider/visual-networking-index-vni/white_paper_c11-520862.pdf`, 2014. Megtekintve: 2014. október 18.

[15] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall, 1st edition, 1987.

[16] The Wireshark team. *Wireshark Wiki*. `http://wiki.wireshark.org`, Megtekintve: 2014. október 12.