Bufferbloat-ilmiö Internetissä

Otto Reinikainen

Pro gradu –tutkielma

ITÄ-SUOMEN YLIOPISTO

Tietojenkäsittelytieteen laitos

Tietojenkäsittelytiede

Marraskuu 2019

ITÄ-SUOMEN YLIOPISTO, Luonnontieteiden ja metsätieteiden tiedekunta, Kuopio
Tietojenkäsittelytieteen laitos
Tietojenkäsittelytiede

Opiskelija, Otto Reinikainen: Bufferbloat-ilmiö Internetissä
Pro gradu –tutkielma, 57 s., 2 liitettä (3 s.)
Pro gradu –tutkielman ohjaaja: FT Matti Nykänen
Marraskuu 2019

Tiivistelmä: Modernit pakettikytkentäiset verkkoyhteydet kärsivät puskuroinnin aiheuttamasta ylimääräisestä verkkoviiveestä ja sen vaihtelusta. Puskurointi aiheuttaa ylimääräistä verkkoviivettä siinä tapauksessa, kun puskureissa pidetään suurta määrää dataa, jota ei voida välittömästi lähettää eteenpäin. Ilmiötä edesauttaa Internetin käytetyimmän tiedonsiirtoprotokollan, TCP/IP:n, pakettien hävikkiin perustuva ruuhkanhallinta, joka ei tunnista todellista ruuhkautumista vaan ainoastaan pakettien hävikin. Pakettien hävikkiin perustuva ruuhkanhallinta yhdistettynä suuriin puskureihin sekä FIFO-tyyppiseen pakettien käsittelyyn tietoliikennelaitteilla saa aikaan tilanteen, jossa verkkoviive nousee merkittävästi silloin, kun verkko on ruuhkautunut. Tilanne johtuu siitä, että ennen hävikin tapahtumista puskureiden täytyy täyttyä niin paljon, että niihin ei enää mahdu uusia paketteja, minkä jälkeen paketteja joudutaan pudottamaan. Puskureissa voi olla jopa satojen millisekuntien edestä dataa. Tässä tutkielmassa esitellään nykyaikaisia menetelmiä, joilla verkkoyhteydelle saadaan hyvä käyttöaste siten, että korkea käyttöaste ei aiheuta merkittävää verkkoviiveen nousua. Näitä menetelmiä ovat muun muassa aktiivinen jononhallinta, kehittyneet kuljetuskerroksen ruuhkanhallintamenetelmät sekä reiluuden hyödyntäminen tietoliikenteessä. Lisäksi tutkielmassa esitellään yksi todellinen käyttötapaus, jossa näitä menetelmiä hyödynnetään verkon suorituskyvyn parantamiseksi verkkopelitapahtumassa.

Avainsanat: aktiivinen jononhallinta, Bufferbloat, ruuhkanhallinta, TCP/IP, vuorontaminen

ACM-luokat (ACM Computing Classification System, 1998 version): C.2, D.4.4

UNIVERSITY OF EASTERN FINLAND, Faculty of Science and Forestry, Kuopio
School of Computing
Computer Science

Student, Otto Reinikainen: Bufferbloat-phenomenon in Internet
Master’s Thesis, 57 p., 2 appendixes (3 p.)
Supervisor of the Master’s Thesis: PhD Matti Nykänen
November 2019

Abstract: Modern packet-switched networks are suffering from unnecessary delay and jitter which are caused by excessive buffering. Buffering adds extra delay to network paths when data is being held in buffers instead of forwarding it to the network. This phenomenon is being contributed by loss-based congestion control algorithms like the one used in TCP/IP. These algorithms are not detecting the real congestion of the network. They only detect the loss of packets which only occurs once the buffers are already full. Loss-based congestion control combined with large FIFO-buffers causes the effect of Bufferbloat where the network congestion causes excessive increase in network delay. This happens because packet loss only occurs when the buffers are already full and contain possibly hundreds of milliseconds worth of data. This Thesis explains the problem and introduces several mitigation methods to improve the delay in modern networks. These methods include active queue management, advanced congestion control algorithms and fair queuing. We also represent a real-world scenario where these methods are used in a competitive LAN-party where real-time is the key to success.

Keywords: active queue management, Bufferbloat, congestion control, scheduling, TCP/IP

CR Categories (ACM Computing Classification System, 1998 version): C.2, D.4.4

Esipuhe

Haluan kiittää vanhempiani kannustamisesta opiskelemaan lähdössä ja erityisesti niiden aikana kaikesta, erityisesti rahallisesta, tuesta. Haluan lisäksi kiittää työni ohjaajaa, Matti Nykästä, uskollisuudesta tieteelle sekä hyvästä ohjauksesta tässä työssä. Haluan myös kiittää Kuopion Opiskelija-asunnot Oy:tä halvasta sähköstä sekä Internetyhteydestä, jotka mahdollistivat tämän tutkielman aiheeseen tutustumisen. Haluan myös kiittää tänä vuonna edesmennyttä tietojenkäsittelytieteilijää, Sally Floydia, hänen merkittävästä työstään aktiivisen jononhallinnan parissa. Kiitokset tuesta myös UEF:in Digiteknologian TKI-ympäristöhankkeelle (Digikeskus-hanke; Hankekoodi: A74338), josta saatiin tarvittavaa tarkastus- ja viimeistelyapua työn viimeisillä hetkillä.

Suuri kiinnostus tietoliikenteeseen ja etenkin viiveen hallintaan sai alkunsa aikoinaan Counter-Strike: Global Offensive-pelipalvelimen ylläpidosta vuonna 2014. Ensin niitä oli yksi, sitten kaksi, kolme, neljä ja pian huomasinkin ylläpitäväni varsin suuren suosion saavuttanutta peliyhteisöä. Jo heti alkumetreillä havahduin ongelmaan tässä pelissä – jokaisen uuden pelaajan tuli ladata palvelimelle liittyessään karttatiedostot tältä samalta palvelimelta. Silloisen pienen kaistanleveyden vuoksi nämä lataukset aiheuttivat suurta vaihtelua jo palvelimelle yhdistyneiden pelaajien keskuudessa. Tämähän ei minulle tietenkään sopinut, sillä halusin pelin toimivuuden olevan suorastaan priimaa omilla palvelimillani, jotka sijaitsivat opiskelija-asunnossa lähes ilmaisen sähkön sekä halvan tiedonsiirron paratiisissa – tiedonsiirtoa nimittäin toteutui teratavuittain joka kuukausi useiden vuosien ajan. Laadukkaan pelikokemuksen toivossa lähdin tutkimaan aihetta lisää ja monen mutkan kautta tässä tutkielmassa esiintyvät aiheet sekä monet muutkin tulivat minulle tutuiksi vuosien varrella. Yhä tänäkin päivänä koen pientä innostusta, kun kuulen jostain uudesta aihetta koskettavasta innovaatiosta.

Pelipalvelinten ylläpito oli mukavaa ja haastavaa mutta kaipasin myös käytännön kokemusta, minkä vuoksi lähdin koittamaan onneani pienen pelitapahtuman parissa, jossa tutustuin sattumalta isompaa pelitapahtumaa järjestäviin kahteen, nykyisin varsin hyviin ystäviini. No, siitähän se sitten lähti. Tänä vuonna nimittäin tuli kohdalleni jo viides Gyostage ja kolmas Skynett Langames, jossa olen ollut suunnittelemassa ja rakentamassa tietoverkkoa. Ensimmäisenä vuonna tapasin tapahtumassa kolmannen hyvän ystävän, jonka kanssa näitä verkkoja edelleen rakennamme. Viisi tänään… vai sittenkin kuusi? – mukana kyllä ollaan, mikäli tapahtuma vielä järjestetään. Hämärästä aamunkoittoon ja kohti asentamisen loputonta riemua!

Lyhenneluettelo

ACK	Acknowledgement; kuittaus onnistuneesta tiedonsiirrosta TCP/IP -protokollassa
ASIC	Application-specific integrated circuit; piiri, joka on suunniteltu suorittamaan tietyn tyyppistä laskentaa mahdollisimman suorituskykyisesti
AQM	Active queue management; aktiivinen jononhallinta
BBR	Bottleneck bandwidth and round-trip propagation time; verkon todellisen ruuhkautumisen havaitseva ruuhkanhallinta-algoritmi
BDP	Bandwidth-delay product; verkkoyhteyden nopeuden ja viiveen tulo
CAKE	Common Applications Kept Enhanced; nykyaikainen kehityksen alla oleva AQM-algoritmi
CoDel	Controlled delay; AQM-algoritmi, joka pyrkii hallitsemaan jonotusviivettä
DHCP	Dynamic host configuration protocol; protokolla, jolla verkkoon liitetyt laitteet saavat automaattisesti verkossa käytettävät asetukset
DNS	Domain name system; palvelu, joka muuntaa verkkotunnukset niitä vastaaviksi IP-osoitteiksi
DOCSIS	Data over cable service interface specification; tiedonsiirtostandardi
DRR	Deficit round robin; vakioajassa toimiva reilu vuoronnin
DSL	Digital subscriber line; tietoliikennetekniikoiden joukko, jolla voidaan siirtää tietoa käyttäen puhelinverkkokäyttöön suunnattuja kaapelointeja
ECN	Explicit congestion notification; IP- ja TCP-protokollien laajennos, jolla voidaan viestiä verkon ruuhkautumisesta
FIFO	First in first out; jonotustapa, jossa jonoa käsitellään purkaen sitä alkupäästä ja lisäten alkioita jonon loppuun
FQ-CoDel	Flow queue CoDel; AQM-tekniikka, joka yhdistää DRR-vuorontimen ja viivettä hallitsevan CoDel-jononhallinnan yhdeksi kokonaisuudeksi
HFSC	Hierarchical fair service curve; tietoliikenneverkoissa käytettäväksi suunniteltu vuorontaja, jolla voidaan hallita kaistanleveyttä sekä jonotusviivettä
HTTP	Hypertext transfer protokol; protokolla, jolla välitetään mediasisältöä Internetissä
IP	Internet protocol; pakettipohjaiseen tiedonsiirtoon käytetty protokolla Internetissä
IRTF	Internet research task force; tutkimusorganisaatio, joka tutkii Internetiin liittyviä aiheita
LEDBAT	Low extra delay background transport; ruuhkanhallinta-algoritmi, joka pyrkii siirtäämään tietoa pienimmällä mahdollisella prioriteetilla muun liikenteen taustalla
LPCC	Low-priority congestion control; yleisnimitys ruuhkahallinta-algoritmeille, jotka siirtävät pienellä prioriteetilla tietoa taustalla
MTU	Maximum transmission unit; suurin mahdollinen L3-paketti, joka voidaan siirtää tietoverkossa pilkkomatta pakettia osiin
NAT	Network address translation; osoitteenmuunnostekniikka, jolla yksittäisen IP-osoitteen takana voi olla kokonaisia aliverkkoja
OSI	Open systems interconnection reference model; tapa kuvata tiedonsiirtoprotokollien toimintaa seitsemällä eri kerroksella

P2P	Peer to peer; palvelimeton verkko, jossa verkon laitteet siirtävät tietoa keskenään
PIE	Proportional integral controller enhanced; eräs jonotusviivettä hallinnoiva satunnaisuuteen perustuva AQM-tekniikka
QoS	Quality of service; tietoliikenteen priorisointi ja luokittelu tietyn palvelun laadun takaamiseksi
RED	Random early detection; AQM-tekniikka, jossa mitataan kertyneen jonon pituutta ja aletaan tarvittaessa pudottamaan paketteja jonon lyhentämiseksi
RTMP	Reaaliaikaiseen videolähetykseen käytettävä tiedonsiirtoprotokolla
RTT	Round-trip time; edestakaiseen matkaan käytetty aika tietoverkossa
TCP	Transmission control protocol; tietoliikenneprotokolla, jolla tietoa voidaan siirtää luotettavasti tietoverkoissa
UDP	User datagram protocol; tietoliikenneprotokolla, jolla tiedonsiirtoviive saadaan pidettyä matalana luotettavuuden kustannuksella
VoIP	Voice over IP; yleisnimitys tekniikoilla, joilla siirretään puheviestintäliikennettä IP-verkoissa

Sisällysluettelo

1 Johdanto ... 1
2 Puskurointi tietoliikenteessä ... 4
3 Jonotusviiveen hallinta ... 8
3.1 Aktiivinen jononhallinta ... 8
3.1.1 RED ... 11
3.1.2 PIE ... 14
3.1.3 CoDel ... 21
3.1.4 FQ-CoDel ... 23
3.2 Kuljetuskerroksen ruuhkanhallinta ... 27
3.2.1 LEDBAT ... 30
3.2.2 BBR ... 32
3.3 Vuorontaminen ... 38
3.3.1 Kapasiteetin hallinta ... 38
3.3.2 Pullonkaulan siirto ... 41
3.3.3 Reiluus tietoliikenteessä ... 42
4 CASE: Skynett Langames 24 ... 44
5 Pohdinta ja jatkotutkimusideat ... 51
Viitteet ... 55
LIITE 1: Qdisc-määritykset ... 58
LIITE 2: Nftables-määritykset ... 59

1 Johdanto

Internet on kehittynyt ja sen käyttäjien määrä on kasvanut räjähdysmäisesti viimeisten kymmenien vuosien aikana. Laaja tietoverkko on mahdollistanut nopean tiedonvälityksen riippumatta käyttäjän sijainnista. Vuosien varrella Internetin palvelut ovat kuitenkin muuttuneet siitä, mitä ne olivat alkuaikoina. Nykyisin Internetiä käytetään niin kaupallisiin kuin myös viihdetarkoitukseen ja verkon välityksellä toimivat palvelut ovat hyvin monipuolisia ja erilaisia käyttötapauksia kehitetään jatkuvasti lisää. Tämä on osaltaan asettanut uusia vaatimuksia modernille verkolle. Nykyisin Internetiä käytetään niin raskaaseen tiedonsiirtoon kuin myös reaaliaikaisiin sovelluksiin kuten esimerkiksi verkkopeleihin sekä puheen ja videon välittämiseen. Tällaisten palveluiden käyttö vaatii käytettävältä tiedonsiirtoverkolta vakautta ja alhaista viivettä saavuttaakseen hyvän käytettävyyden (Grigorescu 2018, RFC 8033).

Nykyaikaiset verkon vaatimukset ja käyttötapaukset ovatkin nostaneet tämän pitkään tunnetun puskuroinnin aiheuttaman viiveen jälleen pinnalle. Verkkoyhteydet ovat kärsineet ja kärsivät yhä edelleen tästä tarpeettomasta viiveestä, mikä laskee verkon yleistä suorituskykyä merkittävästi.

Termiä Bufferbloat käytetään kuvaamaan ongelmaa, jossa liiallinen puskurointi aiheuttaa korkeaa viivettä (eng. latency) ja tämän vaihtelua. Termi esiteltiin ensimmäisen kerran vuonna 2010 (Pan ym. 2013) mutta ongelma on ollut tiedossa jo vuosikymmenien ajan. Termillä tarkoitetaan sellaista tilannetta, jossa liiallinen puskurointi lisää verkon viivettä parantamatta kuitenkaan käytettävää tiedonsiirtonopeutta (Nichols ym. 2012). Tämä viive aiheutuu pakettien ylimääräisestä jonottamisesta verkkoon liitettyjen laitteiden puskureissa (Gettys ym. 2011, Chirichella ym. 2013, Cardozo ym. 2014).

Kuormittamalla verkkoa esimerkiksi raskailla tiedonsiirroilla tai useilla yhtäaikaisilla käyttäjillä voidaan helposti ajautua tilanteeseen, jossa päädytään puskureiden liialliseen käyttöön. Tietyt verkkoyhteystekniikat ovat tälle ilmiölle erityisen alttiita. Esimerkiksi DSL- ja kaapelimodeemiyhteydet voivat pitää sisällään jopa useiden sekuntien mittaisia ylimääräisiä viiveitä pelkästään puskuroinnin vuoksi (Gong ym. 2014). Ongelma on myös havaittavissa suurten etäisyyksien, kuten mannerten välisen, tiedonsiirron aikana. Näissä pitkän matkan yhteyksissä voidaan päätyä tilanteeseen, jossa

kapasiteetiltaan suurista, jopa useiden gigabittien kaistanleveydellisistä, tietoverkoista saadaan päätepisteiden välillä vain megabitteinä sekunnissa mitattavia nopeuksia, vaikka verkon kapasiteetti onkin paljon suurempi (Cardwell ym. 2016).

Viiveen äkillinen vaihtelu ja nousu verkkoa kuormitettaessa johtuu tyypillisesti liiallisesta puskuroinnista, liian pienestä verkon kapasiteetista, tietoliikenneprotokollien heikosti toimivista ruuhkanhallintaominaisuuksista (eng. congestion control), FIFO-tyyppisestä hallitsemattomasta jonotuksesta verkon aktiivilaitteilla sekä erityisesti näiden yhteisvaikutuksesta (Gettys ym. 2011).

Viivettä muodostuu jokaisesta verkon solmusta, joka käsittelee jollain tavalla siihen saapuvia paketteja. Käsittelyn viemä aika riippuu siitä, mitä paketeille tehdään ja kuinka tehokkaasti nämä solmut voivat näitä paketteja käsitellä. Pakettien käsittelyn vaatiman ajan lisäksi solmujen välinen etäisyys vaikuttaa viiveeseen merkittävästi. Bitti liikkuu modernissa tietoverkossa nopeimmillaan valonnopeutta, joten voidaan sanoa, että jokaisella verkon kaarella on jokin ominaisviive. Solmuilla tarkoitetaan verkkoon liitettyjä aktiivilaitteita kuten esimerkiksi tietoliikennekytkimiä ja reitittimiä. Aktiivilaitteiden lisäksi verkon solmut voivat olla myös palvelimia, työasemia tai muita laitteita. Näistä edellä mainituista laitteista jokainen joutuu jollain tavalla käsittelemään tietoliikennepakettia ennen sen lähettämistä seuraavalle solmulle. Pakettien käsittelyyn ja edelleen lähetykseen kulunut aika näkyy suoraan lopulliselle vastaanottajalle viiveenä. Normaalin ominaisviiveen lisäksi verkossa voi esiintyä ylimääräistä viivettä sekä äkillistä viiveen vaihtelua riippuen verkon kuormituksesta. Ominaisviiveeseen ei voida vaikuttaa muuten kuin lyhentämällä reitin pituutta. Siihen puolestaan voidaan vaikuttaa, kuinka paljon matkalla tulee ylimääräistä viivettä puskuroinnista tai pakettien käsittelystä johtuen.

Bufferbloat-ilmiöön voidaan vaikuttaa muun muassa aktiivisella jononhallinnalla, vuorontamisella sekä kehittyneillä ruuhkanhallinta-algoritmeilla tiedonsiirtoprotokollissa. Aktiivisella jononhallinnalla tarkoitetaan nimensä mukaisesti verkkopakettien muodostaman jonon eli puskurin hallintaa. Vuorontamisella puolestaan tarkoitetaan tietyn verkkopaketin, tietovirran tai jonon valintaa saatavilla olevista vaihtoehdoista (RFC 7567). Ongelman perimmäinen syy on TCP:n hävikkiin perustuva ruuhkanhal-

linta (Cardwell ym. 2016) mutta nykyisin tietoverkoissa käytetään myös muita protokollia, joten kuljetuskerroksen ruuhkanhallinnan lisäksi olisi syytä tarkastella myös muita vaihtoehtoja ongelman ratkaisemiseksi. Tämä tutkielma esittelee ongelman, keskeiset tunnetut lähestymistavat siihen sekä vertailee muutamia ongelman ratkaisuun liittyviä nykyaikaisia menetelmiä. Tutkielman keskeinen tutkimuskysymys on miten ja kuinka paljon Bufferbloat-ilmiöön voidaan vaikuttaa nykyaikaisia tekniikoita hyödyntäen.

Aiheen tutkiminen ja ratkaisujen kehittäminen on edelleen ajankohtaista ja tärkeää juuri siitä syystä, että Internetiä käyttävien palveluiden määrä ja vaatimukset kehittyvät jatkuvasti (Grigorescu 2018). Käyttötapauksen mukaan verkolta voidaan vaatia erityyppistä suorituskykyä, minkä vuoksi lisääntynyt viive voi olla todellinen käytettävyysongelma.

Tutkijoiden keskuudessa on ollut kahta lähestymistapaa ongelmaan - joidenkin tutkijoiden mukaan paras ratkaisu liiallisen puskuroinnin aiheuttamiin ongelmiin olisi suoraan puskureihin vaikuttava aktiivinen jononhallinta ja toisten mukaan kannattaisi kehittää toimivampia ruuhkanhallinta -ominaisuuksia kuljetusprotokolliin (Gong ym. 2014). Tutkielmassa esitellään molempia lähestymistapoja ja vertaillaan eri ratkaisuja kirjallisuudesta. Kirjallisuuden lisäksi esitellään yksi oikea käyttötapaus, jossa näitä kehittyneitä tekniikoita käytetään parantamaan tietoverkon viivettä. Tiedonsiirtoprotokollalla tarkoitetaan OSI-mallin (ISO/IEC 7498-4) mukaista neljättä kerrosta ja sovelluskerroksella seitsemättä kerrosta. Tutkielmassa esitetyt tietoliikenneverkon aktiivilaitteet toimivat tyypillisesti OSI-mallin toisella ja kolmannella kerroksella hieman laitteesta riippuen. Tulemme esittelemään tässä tutkielmassa puskuroinnin tarpeen tietoliikenteessä luvussa 2 sekä keskeiset menetelmät jonotusviiveen hallitsemiseksi luvussa 3. Esittelemme käytännön esimerkin luvussa 4, jota seuraa pohdinta ja jatkotutkimusideat luvussa 5.

2 PUSKUROINTI TIETOLIIKENTEESSÄ

Puskureita löytyy lähes jokaisesta verkkoon liitetystä laitteesta. Puskuroinnin tarkoitus on tasoittaa tiedonsiirrossa tapahtuvia hetkellisiä piikkejä pitämällä verkkopaketteja sopiva määrä muistissa ennen niiden edelleen lähettämistä (Chirichella ym. 2013, Alfredsson ym. 2013). Verkkoon liitetyissä laitteissa esiintyy tyypillisesti kahdentyyppisiä puskureita, joita ovat laitteiston rengaspuskurit (eng. ring buffer) ja käyttöjärjestelmän puskurit (Cardozo ym. 2014).

Internetin alkuaikoina suurena ongelmana oli aivan liian pieni puskurointi – jopa kevyt kuormitus saattoi johtaa suureen pakettien hävikkiin (eng. packet loss) yhteyksissä, joissa hetkelliset käyttöpiikit ylittivät verkon kaistanleveyden (eng. bandwidth). Pakettien hävikki aiheutti uudelleenlähetettyjen pakettien tulvan. Varsinkin käytettäessä TCP-protokollaa tiedonsiirtoon aiheuttaa pakettien hävikki tiedonsiirtonopeuden heikkenemistä huomattavasti jopa pienellä määrällä hukattuja paketteja. Ongelma johtui laitteiden muistin määrästä, joka oli liian vähäinen. Nykyisin muisti on kuitenkin niin halpaa, että paremman suorituskyvyn toivossa laitteisiin asennetaan suuri määrä muistia ja otetaan myös tarpeettoman suuret puskureiden koot käyttöön (Cardozo ym. 2014, Nichols ym. 2012). Suuret puskurit yhdistettynä hävikkiin perustuvaan TCP:n ruuhkanhallintaan ovat perimmäinen syy tälle ongelmalle, joka on edelleen ajankohtainen jo 1980-luvulta lähtien (Cardwell 2016).

Internetin alkuaikoina ei huomioitu tarpeeksi verkkopakettien välityksen dynamiikkaa. Tämä aiheutti lähes täydellisen verkon ruuhkautumisen, jonka vuoksi alettiin etsimään ratkaisuja ongelmaan. Ilmiötä kutsuttiin nimellä Internetin sulaminen. Ilmiö havaittiin ensimmäisen kerran Internetin kasvuvaiheessa 1980-luvulla (RFC 7567). Ongelmaan kehitettiin ensimmäisenä ratkaisuna ruuhkanhallinta TCP-protokollaan. Ruuhkanhallinnan tarkoituksena on havaita verkon ruuhkautuminen ja hidastaa lähetysnopeutta ruuhkan havaitessaan. Kuitenkin vuosien tutkimuksen ja Internetin kasvamisen jälkeen voidaan todeta, että tiedonsiirtoprotokollien ruuhkanhallinta ei ole riittävä ratkaisu kaikkiin olosuhteisiin. Kuljetusprotokollan ruuhkanhallinnan lisäksi Internetin reitittimien tulisi täydentää näitä mekanismeja (RFC 7567).

Oletusarvoisesti puskureita hallitaan FIFO-tyylisesti siten, että puskurin täytyttyä aletaan verkkopaketteja pudottamaan jonon loppupäästä sen verran, että puskuriin mahtuu jälleen uusia verkkopaketteja. Muussa tapauksessa pudottamista ei tehdä (Grigorescu 2018). Tämä johtaa siihen, että jonoa on jatkuvasti puskurin koon verran, mikä puolestaan on suoraan verrannollinen yhteydessä esiintyvään viiveeseen.

Verkkopaketteja esiintyy käyttötapauksen mukaan hyvin sekalainen määrä ja niiden sisältö voi olla hyvin vaihtelevaa. Moderni tietoverkko tarvitsee puskureita, sillä kapasiteetti voi ylittyä hetkellisten piikkien seurauksena (Nichols ym. 2012). Mikäli sitä ei tehtäisi lainkaan, jouduttaisiin kaikki kapasiteetin yli menevät paketit pudottamaan. Puskurin tehtävä onkin säilöä paketteja sen aikaa, että ne voidaan käsitellä. Bufferbloat-ilmiö muodostuu siinä vaiheessa, kun puskurointi on jatkuvaa. Puskurointia tapahtuu käytännössä siitä syystä, että verkon solmuun tulee enemmän dataa kuin se pystyy käsittelemään (RFC 7567, Nichols ym. 2012). Tästä syystä puskurointia aiheutuu eniten verkon pullonkaulaan eli Internetin tapauksessa lähettäjän ja vastaanottajan välisen polun hitaimman verkkoyhteyden omaavaan solmuun. Pullonkaulalla tarkoitetaan hitainta yhteyttä tiettyyn suuntaan ja tietyllä polulla verkossa. Nykyaikaisissa verkoissa voidaan lähettää ja vastaanottaa samaan aikaan. Lähetys ja vastaanotto voidaan myös reitittää käyttäen eri polkuja. Tämä aiheuttaa sen, että molemmissa suunnissa voi olla erillinen pullonkaula.

Verkon hajautetusta luonteesta johtuen lähettäjä ei voi suoraan tietää vastaanottajan päässä olevaa tarkkaa kapasiteettia. Tästä ongelmasta tulee erityisen hankala silloin, kun moni lähettäjä lähettää yhdelle vastaanottajalle tietoa samaan aikaan verkon läpi. Kapasiteetti muuttuu aina, kun uusi tietovirta alkaa kulkea verkon läpi. Osana ongelmaa on siis verkon tämänhetkisen kapasiteetin tuntemattomuus lähettäjille. Koska verkkoyhteyttä käyttävät protokollat eivät pysty sopeutumaan riittävän tehokkaasti yhteyden muutoksiin, aiheutuu kapasiteetin hetkellistä ylitystä verkon pullonkaulassa, mikä puolestaan lisää puskuroinnin tarvetta entisestään. Koska hitaimman yhteyden kapasiteetin ylittävää osuutta ei voida välittömästi siirtää vastaanottajalle yhteyden käyttöasteen ollessa täysi, joudutaan saapuvat verkkopaketit tallettamaan edellisen solmun puskuriin odottamaan sopivaa lähetysikkunaa. Odottaminen tunnetusti lisää viivettä (Gettys ym. 2011).

Ilman puskurointia toimiva verkkoyhteys ei kykene sopeutumaan vaihtelevaan tiedonsiirtoon lainkaan ja jokainen kapasiteetin ylittävä verkkopaketti jouduttaisiin jättämään täysin käsittelemättä. Tällainen verkko jättää puskuroinnista aiheutuvan viiveen pois kokonaan mutta aiheuttaa sen sijaan muita ongelmia. Ilman puskuria toimivan tiedonsiirron tulee olla täydellisesti ennakoitavissa eikä verkossa saisi tapahtua mitään muutoksia. Lisäksi pakettien hävikin minimoiminen vaatisi täsmällistä ajoitusta. Tämän kaltainen verkko olisi monimutkainen, kallis ja rajoittunut (Gettys ym. 2011) eikä sellaista näistä syistä kannata toteuttaa.

Liian suurien puskurien aiheuttamat ongelmat havaittiin jo vuosikymmeniä sitten ja ongelmaan on siitä lähtien kehitetty erilaisia ratkaisuja. Ongelma ei kuitenkaan ole niin yksinkertainen, että voisimme vain kytkeä puskurit pois päältä. Kuten aiemmin todettiin, ovat puskurit tärkeitä Internetin toimivuudelle. Se mikä puskureiden käytöstä tekee ongelmallisen, on niiden sopivan kapasiteetin määrittäminen (Gettys ym. 2011). Kapasiteetin määrittämisen lisäksi olennaista on puskurin käyttötapa. Normaali FIFO-tyyppinen puskurin käyttötapa saa varsinkin TCP-protokollaa käytettäessä helposti aikaan verkkoyhteyden ruuhkautumisen ja näin ollen myös viiveen nousun. Tämä korostuu varsinkin suurten puskureiden kanssa. Kun käytössä olevat puskurit verkon pullonkaulassa ovat suuria, pitää pakettien hävikkiin perustuva ruuhkanhallinta puskurit jatkuvasti täynnä aiheuttaen verkkoviiveen nousua (Cardwell 2016). Tämä johtuu siitä, että puskuriin ehtii kerääntymään suuri määrä dataa ennen kuin kuljetuskerroksen hävikkiin perustuva ruuhkanhallinta kytkeytyy päälle ja hidastaa lähetysnopeutta.

Suuret puskurit lisäävät viivettä mutta mahdollistavat korkeanopeuksisen tiedonsiirron (Alfredsson ym. 2013), suuren verkkoyhteyden käyttöasteen sekä vähentävät pakettien hävikkiä (Grigorescu 2018). Pienet puskurit toisaalta mahdollistavat alhaisen viiveen mutta saavutettu tiedonsiirtonopeus voi jäädä pienemmäksi. Lisäksi liian pieni puskuri saa aikaan pakettien hävikkiä. Puskureiden optimaalinen koko määrittyy pitkälti verkon käyttötarkoituksesta. Raskaat tiedonsiirrot, kuten esimerkiksi suurien tiedostojen lataukset HTTP-protokollalla, eivät välitä juurikaan verkkoviiveen määrästä ja näiden tapauksessa suuri kaistanleveys onkin tärkeämpää. Toisaalta esimerkiksi verkkopelit tai VoIP kärsivät viiveestä todella paljon ja lisääntynyt viive voi aiheuttaa käytettävyysongelmia tai jopa sovelluksen toimimattomuutta (Cardozo ym. 2014).

Toimimattomuutta voi esiintyä sellaisessa verkossa, jossa puskurit ovat erityisen suuria ja paketeilla on tietty elinaika, jonka ajan paketit ovat hyödyllisiä niiden vastaanottajalle. Tällaisessa tapauksessa puskureissa odottavan verkkopaketin hyödyllisyys pienenee koko ajan mitä pidempään lähetystä joudutaan odottamaan (Cardozo ym. 2014).

Se, että liiallinen puskurointi lisää viivettä, ei ole asian ainoa haittapuoli. Liiallinen puskurointi lisää myös viiveen vaihtelua (eng. jitter). Ongelmia alkaa esiintyä siinä vaiheessa, kun suuret ja ahneet tiedonsiirrot joutuvat jakamaan saman verkkoyhteyden purskeisen tai aikakriittisen sovelluksen kanssa. Raskas tiedonsiirto aiheuttaa sen, että puskuri on jatkuvasti täynnä. Jotta uusi paketti saadaan lähetettyä solmusta toiseen, tulee puskurin ensin tyhjentyä edellisistä paketeista. Mitä suurempi määrä puskurissa on paketteja, sitä pidempään sen tyhjeneminen kestää. Erityisesti aikakriittiset sovellukset, kuten VoIP kärsivät tästä todella paljon (Alfredsson ym. 2013).

3 JONOTUSVIIVEEN HALLINTA

Kuten luvussa 1 totesimme, voidaan puskuroinnin aiheuttamaan viiveeseen vaikuttaa muun muassa aktiivisella jononhallinnalla, vuorontamisella sekä kuljetuskerroksen ruuhkanhallinnalla. Nykyaikaisilla nopeilla tietoliikenneyhteyksillä olisi syytä kehittää menetelmiä, jotka mahdollistavat suuren tiedonsiirtonopeuden ja pienen viiveen. Toistaiseksi tähän ongelmaan ei ole yhtä ainoaa oikeaa ratkaisua. Jonotuksesta aiheutuvaa viivettä voidaan kuitenkin parantaa käyttämällä aktiivista jononhallintaa (AQM) ja todellisen ruuhkautumisen havaitsevia ruuhkanhallinta-algoritmeja kuljetusprotokollissa. Lisäksi verkon pullonkaulan hallinta on tärkeässä asemassa viiveen hallitsemisen kannalta, sillä suurin osa verkossa syntyvästä viiveestä muodostuu siihen. Luvussa 3.1 tulemme esittelemään muutaman nykyaikaisen ja tunnetun aktiivisen jononhallinnan menetelmän, luvussa 3.2 kuljetuskerroksessa toteuttavan ruuhkanhallinnan toiminnan ja merkityksen sekä luvussa 3.3 vuorontamisen keinot viiveen ja kaistanleveyden hallintaan.

3.1 Aktiivinen jononhallinta

Aktiivisen jononhallinnan tarkoitus on nimensä mukaisesti hallita jonoa eli puskuria. Hallinta perustuu jonon pituuden tai jonotusajan hallintaan ECN-merkitsemällä tai pudottamalla jonossa olevia paketteja tarpeen mukaan (RFC 7567). ECN-merkintä on IP-paketin otsikkotietoon lisättävä merkintä, joka kertoo lähettäjälle, että verkko on ruuhkautunut. Tämän merkin havaitessaan lähettäjän tulisi hidastaa lähetysnopeuttaan käyttämänsä ruuhkanhallinta-algoritmin mukaisesti. Valitettavasti kaikki Internetin reitittimet ja kuljetusprotokollien toteutukset eivät hyödynnä tai tue ECN-merkintää. Sopivan toimintatilan valinta on siis käytettävästä tietoverkosta ja kuljetusprotokollasta riippuva. Koska ECN vaatii tukea käytössä olevalta kuljetuskerroksen protokollalta todellisen ruuhkanhallinnan käyttämiseksi, ei se sovi käytettäväksi kaikkiin mahdollisiin tapauksiin (RFC 3168).

Aktiivista jononhallintaa ehdotetaan tyypillisesti käytettäväksi Internetin reitittimillä. Koska vain nämä verkon reitittimet tietävät tarkkaan oman puskurinsa käyttöasteen ja tunnistavat matkaan käytetyn ajan puskuroinnin aiheuttamasta viiveestä, ovat ne paras

paikka havaita verkon ruuhkautuminen (Floyd ym. 1993). Lisäksi samaa reititintä yleensä käyttävät monet käyttäjät ja reitittimen läpi kulkeva liikenne on hyvin monimuotoista, minkä vuoksi päätökset on paras tehdä reitittimellä (Floyd ym. 1993). AQM-algoritmien suunnittelun keskeisiä tavoitteita on pakottaa reiluutta tietovirtojen kesken, rangaista aggressiivisia ja raskaita tietovirtoja sekä suojata vasta alkaneita ja lyhyitä tietovirtoja (Gong ym. 2014).

Perinteisesti aktiivilaitteiden jonoja on hallittu FIFO-tyyppisesti mahdollisimman suurilla puskureilla. Laitteet välittävät paketteja parhaansa mukaan eteenpäin ja puskuroivat loput. Kun puskurit täyttyvät, aletaan paketteja pudottaa jonon loppupäästä. Tällainen toiminta aiheuttaa joissain tapauksissa sen, että jotkin tietovirrat käytännössä dominoivat käytössä olevaa kapasiteettia siten, että muille tietovirroille ei jää tilaa jonossa. Tämän lisäksi hallitsemattomat FIFO-jonot voivat pysyä suuria ajanjaksoja täynnä, jolloin verkkoon muodostuu ylimääräistä viivettä. Tästä syystä kehitettiin aktiivinen jononhallinta, jonka tarkoitus on hallita näitä jonoja kehittynein menetelmin pitäen jonotuksesta aiheutuvan viiveen matalana mutta verkon virtausnopeuden silti mahdollisimman korkeana (RFC 2309).

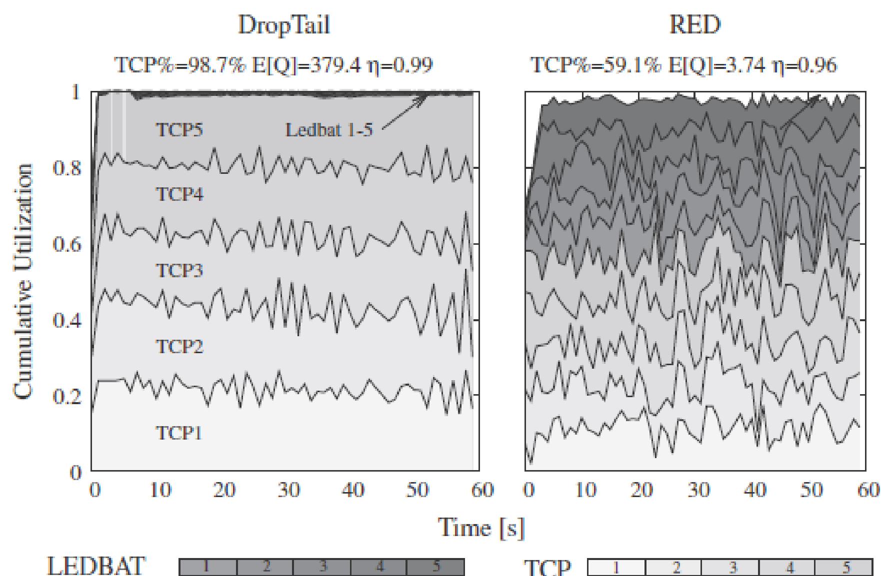
Jatkuvasti täynnä olevien puskureiden aiheuttamat ongelmat havaittiin jo lähes 30 vuotta sitten. Aiheen tutkiminen oli erityisen suosittua 90-luvulla, jonka jälkeen suosio alkoi laskea. Nyt useita vuosia myöhemmin asia on noussut jälleen pinnalle uusien tekniikoiden, kuten CoDel-algoritmin myötä (Gong ym. 2014). Algoritmeja on kehitetty useita, joista eräitä uusimpia ovat CoDel, FQ-CoDel ja PIE, joista viimeinen on sen hyödyllisyyden ja helpon käyttöönoton vuoksi uuden DOCSIS 3.1 -standardin mukaisissa kaapelimodeemijärjestelmissä pakollinen ominaisuus (RFC 8034, White ym. 2014).

Aktiivinen jononhallinta ja sen merkitys puskureiden käytössä on ollut tiedossa vuosikymmenten ajan mutta tekniikat eivät ole saaneet suurta suosiota käytännön toteutusten haasteiden vuoksi. Varhaiset ratkaisut, kuten vuonna 1998 IRTF:n suosittelema RED, olivat haasteellisia ottaa käyttöön, sillä näiden algoritmien parametrien säätäminen oli vaikeaa (Nichols ym. 2012). Parametrien säätämisen haasteellisuuden lisäksi näihin aikoihin myöskin laitteiden laskentanopeus oli huomattavasti nykyisiä laitteita

alhaisempi, mikä osaltaan rajoitti kehittyneempien tekniikoiden käyttöä. Nykyisin tilanne on kuitenkin muuttunut ja jotkin operaattorit ovatkin nykyään ottaneet käyttöön AQM-tekniikoita verkkolaitteissaan parantaakseen loppukäyttäjän kokemusta (Gong ym. 2014).

Tässä luvussa esitellään alkuperäinen RED sekä muutama nykyaikainen vaihtoehto sille. Algoritmeista esitellään niiden keskeiset toimintaperiaatteet, tarkastellaan pseudokoodiesitystä ja lopuksi vertaillaan niiden suorituskykyä. Kaikissa esitetyissä algoritmeissa on kaksi toimintatilaa, joita ovat ECN-merkitseminen ja pakettien pudottaminen. Toteutuksesta riippuen käyttäjä voi valita kumpaa käytetään. Esiteltävät algoritmit on valittu niiden kirjallisuudessa esiintymisen ja Linux-toteutuksen perusteella.

AQM-tekniikat eivät kuitenkaan ole täydellinen ratkaisu. Niiden käyttöön liittyy ainakin yksi tunnettu haittavaikutus, joka on syytä tiedostaa, mikäli aikoo käyttää näitä tekniikoita. Kuvassa 1 (Gong ym. 2014) esitetään ns-2-simulaatio, jossa on käytetty 10 Mbps-nopeuksista verkkoyhteyttä 500 verkkopaketin puskurilla. Tässä tapauksessa puskurin ollessa täysi on sen sisältämä viive 600 ms. Esimerkkitapauksessa käytetään viittä TCP NewReno- ja viittä LEDBAT -ruuhkanhallinnan omaavaa LPCC-yhteyttä. Kuvassa vasemmalla on käytössä perinteinen jonotus ilman AQM-tekniikoita. Kuvan oikealla puolella on muuten sama tilanne mutta käytössä on lisäksi RED-tyyppinen AQM-tekniikka.



Kuva 1. LPCC- ja AQM-tekniikoiden yhteisvaikutus. (Gong ym. 2014)

Kuvasta 1 voidaan havaita, että vasemmalla puolella näkyvät 5 normaalia TCP-yhteyttä jakavat käytössä olevan kapasiteetin reilusti keskenään. Näiden lisäksi 5 taustalla olevaa LEDBAT-ruuhkanhallintaa käyttävät yhteydet jakavat tasaisesti jäljelle jäävän kapasiteetin vaikuttamatta TCP-yhteyksien kapasiteettiin. Tiedonsiirron aikana saavutetaan 99 % käyttöaste verkkolinkille. Raskaat TCP-tiedonsiirrot nostavat toisaalta myös keskimääräisen puskuroinnin viiveen 379 millisekuntiin (Gong ym. 2014).

Kuvan 1 oikean puoleisesta graafista voidaan havaita, kuinka RED-tyyppisen AQM-tekniikan yhteisvaikutus LEDBAT-ruuhkanhallinnan kanssa on saanut aikaan sen, että nyt kaikki yhteydet jakavat käytössä olevan kapasiteetin melko tasaisesti keskenään eikä yhteyksien kesken näytä olevan juuri mitään prioriteettieroja. Tämä on haittavaikutus, joka on saatu aikaan yhdistämällä edellä mainitut LPCC- ja AQM-tekniikat. LPCC-tekniikoiden tarkoitus on siirtää tietoa alhaisemmalla kuin parhaalla mahdollisella prioriteetilla tukkimatta käytössä olevaa verkkolinkkiä. Tämä periaate saavutettiin, kun mukana ei ollut AQM-tekniikkaa. AQM-tekniikan käyttöönoton seurauksena jonotus on lyhentynyt niin paljon, että LEDBAT ei enää kykene mittaamaan luotettavasti viiveen nousua eikä tästä syystä toimi oikein. Sen sijaan, että siirrettäisiin taustalla ruuhkaa aiheuttamatta tietoa, alkaa kapasiteetin käyttö muistuttaa viereistä NewReno-ruuhkanhallintaa (Gong ym. 2014).

Gong ym. 2014 testasivat yli 3000 erilaista simulaatiota omissa tutkimuksissaan ja totesivat, että ongelma voidaan yleistää koskettamaan kaikkien LPCC- ja AQM-tekniikoiden yhdistämistä. AQM-tekniikalla saavutettiin tässä esimerkkitapauksessa jonotuksen aiheuttaman viiveen lasku neljään millisekuntiin mutta saatiin aikaan uusi ongelma. Se, että onko tämä merkittävää, riippuu hyvin pitkälti käyttötapauksesta. Tulemme esittelemään seuraavaksi luvussa 3.1.1 RED-, luvussa 3.1.2 PIE-, luvussa 3.1.3 CoDel- sekä viimeiseksi luvussa 3.1.4 FQ-CoDel-algoritmit aktiivisessa jononhallinnassa.

3.1.1 RED

RED on aktiivinen jononhallinta-algoritmi, joka on tarkoitettu käytettäväksi reitittimissä. RED estimoi puskurin käyttöastetta ja pudottaa tai merkitsee verkkopaketteja satunnaisuuteen perustuen. Todennäköisyys satunnaiselle paketin merkitsemiselle tai pudotukselle kasvaa sitä mukaa, mitä suuremmaksi algoritmi arvioi puskurissa olevan

jonon pituuden. Algoritmin toiminta perustuu kahteen toiminnallisuuteen: jonon pituuden estimointiin ja pakettien pudottamisesta päättämiseen (RFC 2309). Algoritmin tavoitteena on hallita keskimääräistä jonon pituutta (Floyd ym. 1993). RED on ensimmäisiä aktiivisen jononhallinnan algoritmeja, jotka kehitettiin vastaamaan liiallisen puskuroinnin aiheuttamaan viiveen nousuun.

RED käyttää hyödykseen TCP-protokollan hävikkiin perustuvaa ruuhkanhallintaa mutta lähestyy ongelmana varsinaisesti puskurin täyttymistä. Algoritmia voidaan tästä syystä käyttää riippumatta kuljetuskerroksen protokollasta hallitsemaan myös pelkkää puskuria. Algoritmi on alun perin suunniteltu käytettäväksi sellaiseen verkkoon, jossa kuljetuskerroksen protokollat vastaavat ruuhkautumisen merkkeihin verkossa. Tästä syystä se soveltuu erityisen hyvin käytettäväksi TCP-protokollan kanssa. Algoritmin suunnittelu perustuu siihen, että jo yksittäinen pudotettu tai ECN-merkitty paketti välittää kuljetuskerrokselle viestin verkon ruuhkautumisesta (Floyd ym. 1993).

Pidempään jatkunut ruuhkautuminen tunnistetaan keskimääräisen jonon pituuden kasvuna. Koska algoritmi valitsee paketit satunnaisesti tietyllä todennäköisyydellä, pakettien valinnan todennäköisyys on suhteellista tietovirtojen osuudelle käytetystä kokonaiskapasiteetista. Esitellään seuraavaksi RED-algoritmin pseudokoodi kuvan 2 avulla (Floyd ym. 1993).

```
Initialization:
   avg ← 0
   count ← −1
for each packet arrival
 calculate the new average queue size avg:
      if  the queue is nonempty
         avg ← (1 − w_q)avg + w_q q
      else
        m ← f(time − q_time)
         avg ← (1 − w_q)^m avg
   if  min_th ≤ avg < max_th
      increment count
      calculate probability p_a:
         p_b ← max_p(avg − min_th)/(max_th − min_th)
         p_a ← p_b/(1 − count · p_b)
      with probability p_a:
         mark the arriving packet
         count ← 0
   else if max_th < avg
      mark the arriving packet
      count ← 0
   else count ← −1
when queue becomes empty
   q_time ← time
```

Kuva 2. RED-algoritmin pseudokoodiesitys. (Floyd ym. 1993)

Algoritmille annetaan parametreiksi kuvassa 2 esitetyt min_{th}, max_{th} ja max_{p}. Nämä parametrit kuvaavat jonon pienintä ja suurinta raja-arvoa, joiden puitteissa algoritmi pyrkii toimimaan sekä suurinta mahdollista todennäköisyyttä, jolla paketteja valitaan pudotettavaksi tai merkittäväksi.

Kun jonoon saapuu paketti, lasketaan keskimääräinen jonon pituus avg. Keskimääräisen pituuden laskennassa käytetään *liukuvaa keskiarvoa* (eng. exponential moving average), minkä vuoksi algoritmi saadaan keskittymään enemmän jatkuvaan jonon pituuteen eikä niinkään hetkellisiin piikkeihin (Floyd ym. 1993). Mikäli keskimääräinen jonon pituus on suurempi kuin alaraja, min_{th}, lasketaan paketin pudotukselle todennäköisyys, jonka perusteella paketti mahdollisesti valitaan pudotettavaksi. Mikäli jonon keskimääräinen pituus puolestaan ylittää parametrina asetetun max_{th}-arvon, pudotetaan paketti välittömästi. Todennäköisyys kasvaa sitä suuremmaksi, mitä suurempi keskimääräinen jonon pituus on ja mitä enemmän paketteja edellisestä pudotuksesta on. Algoritmin toiminta perustuu siihen, että satunnainen pakettien pudotus aiheuttaa lähettäjän päässä lähetysnopeuden hidastamisen, mikä puolestaan vähentää jonon muodostumista. Jonon koon kasvaessa riittävän suureksi, aletaan paketteja pudottamaan enemmän, jolloin lähetysnopeus hidastuu entisestään. Paketteja ei pudoteta, mikäli jonoa ei pääse kertymään.

Algoritmi on varsin kevyt, sillä sen tarvitsee suorittaa vain vakioaikaisia operaatioita, eikä sen tarvitse muokata jonoon saapuvia paketteja. Tästä syystä se skaalautuu hyvin pakettien määrän suhteen. Laskennallisen vaativuuden lisäksi algoritmi on muistin käytön suhteen myös hyvin kevyt, sillä yksinkertaisessa edellä esitetyn kuvan 2 mukaisessa toteutuksessa sen tarvitsee pitää muistissa kerrallaan vain muutaman yksittäisen muuttujan arvot. Näistä syistä se soveltuu erinomaisesti käytettäväksi verkon aktiivilaitteilla. Esimerkiksi tietoliikennelaitevalmistaja Juniper käyttää RED-algoritmia tietyillä laitemalleillaan oletusarvoisena AQM-tekniikkana (juniper).

RED on yksinkertainen algoritmi, jolla voidaan laskennallisesti hyvinkin kevyesti vaikuttaa puskurissa olevan jonon pituuteen. RED oli pitkään suositeltu AQM-algoritmi käytettäväksi Internetin reitittimillä, kunnes uusien algoritmien kehittymisen myötä suosituksia muutettiin (RFC 2309). Haasteellista algoritmin käytöstä tekee sopivien

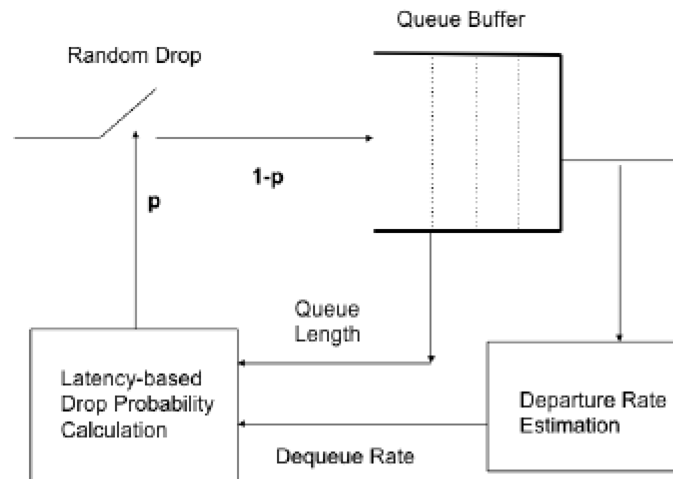
parametrien määrittäminen, mikä on yksi syy sille, että RED ei ole saanut suurta suosiota verkoissa sellaisenaan. Lisäksi se ei sopeudu hyvin sellaisiin verkkoihin, joissa kapasiteetti muuttuu jatkuvasti (Pan ym. 2013). Tällaisia verkkoja ovat esimerkiksi 4G- ja muut mobiiliverkot.

Sopivilla parametreilla käytettynä RED on hyödyllinen AQM-algoritmi, joka soveltuu käytettäväksi erilaisissa tietoverkoissa. Algoritmi on kuitenkin ensimmäisiä laatuaan ja siinä on edellä mainitut rajoitteet. Nykyisin on olemassa myös muita algoritmeja, jotka toimivat paremmin varsinkin muuttuvissa olosuhteissa.

RED:lle on olemassa myös Linux-toteutus, jota voidaan käyttää esimerkiksi sellaisessa tilanteessa, jossa geneeristä Linux-palvelinta käytetään reitittimenä. Toteutus eroaa tässä luvussa kuvatusta geneerisestä toteutuksesta siten, että algoritmia voidaan käyttää myös sopeutuvana eli se kykenee itse laskemaan osan käyttämistään parametreista sopiviksi (tc-red).

3.1.2 PIE

PIE on uudehko algoritmi, jonka toiminta muistuttaa hyvin pitkälti RED:a sillä eroavaisuudella, että PIE mittaa jonon viivettä ja pyrkii suoraan vaikuttamaan viiveeseen. Algoritmin keskeiset suunnittelutavoitteet ovat viiveen hallinta, korkea verkkoyhteyden käyttöaste, yksinkertainen toteutus ja sopeutuvuus erilaisiin verkkoihin (Pan ym. 2013). Lisäksi hetkelliset piikit tulisi kyetä päästämään läpi mutta pitää kuitenkin keskimääräinen jonotusviive mahdollisimman alhaisena. Algoritmin toiminta muodostuu RED:n tapaan samoista kahdesta keskeisestä komponentista, joita ovat satunnainen pakettien valinta ja pudotuksen todennäköisyyden laskenta. Kuvassa 3 (Pan ym. 2013) esitetään algoritmin toimintamalli pakettien saapuessa jonoon.



Kuva 3. PIE-algoritmin toimintamalli. (Pan ym. 2013)

Kuten kuvasta 3 voidaan havaita, noudattaa algoritmi perinteisen jonon loppupäästä pudottamisen sijaan pudottamista alkupäästä. Jonon alkupäästä pakettien pudottamisella saavutetaan nopeampi vaikutus viiveeseen, sillä loppupäästä pudottaessa mahdolliset vaikutukset viiveeseen ilmenevät vasta, kun jonossa edetään kyseisen paketin kohdalle. Kuvassa esitetyn pudotuksen sijaan voidaan käyttää myös ECN-merkintää.

Algoritmille annetaan parametreiksi toteutuksen mukaan jonon pituuden yläraja tavuina (dq_threshold), vertailukohtana käytetty viive millisekunteina (ref_del), tarkasteluväli millisekunteina (T_{update}), skaalauskertoimet α ja β sekä suurin sallittu aika hetkellisille piikeille (burst_max). Tarkasteluväli on aika, jonka kuluessa algoritmi laskee aina uudestaan pakettien pudotukseen käytetyn todennäköisyyden. Näistä parametreista tärkein on kuitenkin ref_del, jota käytetään pudotukseen käytetyn todennäköisyyden laskentaan laskemalla nykyisen viiveen ja parametrin välistä eroa (Pan ym. 2013, Khademi ym. 2014).

Algoritmi soveltaa muiden kehittyneiden AQM-mallien tapaan satunnaista pakettien pudottamista keskeisenä osana toimintaansa. Kun paketti saapuu algoritmin käsiteltäväksi, on sillä nykyisen jonon tilanteen mukaan tietyn suuruinen todennäköisyys tulla valituksi. Satunnainen pakettien pudotus ja merkintä kuitenkin ohitetaan, mikäli jonon viive on alle puolet parametrina annetusta tavoitteesta silloin, kun nykyinen pudotustodennäköisyys on samaan aikaan pieni. Pieni todennäköisyys on tässä tapauksessa pienempi kuin 0,2. Pudotusvaihe ohitetaan myös silloin, jos jonossa on vain muutama

paketti (Pan ym. 2017). Pakettien pudotukseen käytetty todennäköisyys nousee tai laskee sen mukaan, kuinka paljon jonon läpi virtaa paketteja ja miten jonon koko muuttuu. Pakettien pudotukseen käytettävää todennäköisyyttä skaalataan parametreina saatavien α- ja β-arvojen mukaan. Näillä arvoilla algoritmin käyttäytymistä jatkuvan viiveen määrän ja viiveen vaihtelun suhteen voidaan muuttaa (Pan ym. 2013). Toteutuksen mukaan kaikkia parametreja ei ole pakko itse antaa algoritmille vaan voidaan käyttää toteutuksen oletusarvoja.

Todennäköisyys pakettien pudottamiselle lasketaan arvioimalla *liukuva keskiarvo* jonon virtausnopeudelle tavuina. Virtausnopeus voi muuttua joko kapasiteetin muutoksen tai verkon käytön mukaan. Kuvan 4 algoritmissa (Pan ym. 2013) esitetään tapa, jolla PIE laskee tätä virtausnopeutta. Koska virtausnopeutta lasketaan säännöllisesti uudestaan, mukautuu algoritmi muuttuviin olosuhteisiin hyvin.

Departure Rate Calculation:

Upon packet departure

1. Decide to be in a measurement cycle if:

$$qlen > dq_threshold;$$

2. If the above is true, update departure count dq_count:

$$dq_count = dq_count + dq_pktsize;$$

3. Update departure rate once $dq_count > dq_threshold$ and reset counters:

$$\begin{aligned} dq_int &= now - start; \\ dq_rate &= \frac{dq_count}{dq_int}; \\ avg_drate &= (1-\varepsilon) * avg_drate + \varepsilon * dq_rate \\ start &= now. \\ dq_count &= 0; \end{aligned}$$

Kuva 4. Pakettien virtausnopeuden laskeminen PIE-algoritmissa. (Pan ym. 2013)

Kuvassa 4 esitetyt vaiheet 1 ja 2 pitävät huolen siitä, että hetkelliset piikit eivät johda epätarkkoihin mittaustuloksiin. Tämä johtuu siitä, että lyhytkestoinen piikki ei ehdi nostamaan *liukuvaa keskiarvoa* niin paljoa, että se johtaisi pakettien pudottamiseen (Pan ym. 2013), sillä virtausnopeutta mitataan säännöllisesti ja keskiarvoa päivitetään skaalatuilla kertoimilla. Lisäksi tätä mittausta ei suorita lainkaan, mikäli hetkellisen

piikin vaatiman lähetettävän tiedon määrä ei saa kasvatettua jonon pituutta yli sille asetetun dq_threshold-parametrin.

Algoritmin tarkoituksena on hallita jonon viivettä, joten jos mitään jonoa ei ole niin ei ole myöskään tarvetta sitä hallita. Tästä syystä virtausnopeutta lasketaan vain siinä tilanteessa, kun jonossa on sopiva määrä tavuja. Sopivana määränä algoritmi pitää sille parametriksi annettua dq_threshold-arvoa. Näitä mittauksia suoritetaan algoritmille parametrina annetun päivitysvälin mukaisesti laskien eksponentiaalista liukuvaa keskiarvoa. Kuvan 4 dq_count-arvo kuvaa lähetettyjä tavuja edellisestä mittauskerrasta. Tätä parametria suositellaan asetettavaksi 10 KB suuruiseksi sillä olettamuksella, että tyypillinen pakettikoko on väliltä 1 – 1,5 KB (Pan ym. 2013).

Keskimääräisen virtausnopeuden laskemisen jälkeen algoritmi etenee pakettien pudotuksessa käytettävän todennäköisyyden laskentaan. Tämä laskenta suoritetaan T_{update} -parametrina annetuin aikavälein. Laskenta alkaa nykyisen jonotuksen viiveen estimoinnista kaavan 1 mukaisesti (Pan ym. 2013):

$$\text{cur_del} = \frac{\text{qlen}}{\text{avg_drate}} \qquad \text{(kaava 1)}$$

Kaavassa 1 oleva arvo qlen kuvaa jonossa olevan datan määrää ja avg_drate toteutunutta virtausnopeutta verkkoon päin. Kun jonon nykyinen viive on saatu laskettua, skaalataan arvoja $\alpha\ ja\ \beta$ perustuen nykyiseen todennäköisyyteen. Jos todennäköisyys on alle 1 %, jaetaan arvot kahdeksalla. Alle 10 % todennäköisyyden ollessa kyseessä, arvot jaetaan kahdella. Muussa tapauksessa käytetään alkuperäisiä, parametrina saatuja arvoja. Tätä skaalausta suoritetaan siitä syystä, että näin algoritmi saadaan paremmin ja nopeammin sopeutumaan muuttuviin olosuhteisiin (Pan ym. 2013). Lisäksi näin se saadaan itsesääteleväksi. Edellä mainitut luvut on johdettu siitä periaatteesta, että suuret muutokset saavat aikaan epävakautta (Pan ym. 2013). Esimerkiksi silloin, kun p on alle 1 %, halutaan sen arvoa muuttaa korkeintaan noin 0,1 % kerrallaan (Pan ym. 2013). Tämän jälkeen lasketaan varsinainen pakettien pudotukseen käytettävä todennäköisyys p. Laskenta tapahtuu kaavan 2 mukaisesti (Pan ym. 2013):

$$p = p + \alpha * (\text{cur_del} - \text{ref_del}) + \beta * (\text{cur_del} - \text{old_del}) \qquad \text{(kaava 2)}$$

Todennäköisyyden laskennan jälkeen päivitetään edellisen mittauskerran viive vastaamaan juuri laskettua uutta arvoa. Edellä kuvattuja vaiheita toistetaan niin kauan, kun jonoon saapuu uusia paketteja. Hetkellisiin piikkeihin sovelletaan parametrina saatua max_burst-arvoa. Mikäli jonoon saapuvat paketit sopivat piikkeihin sovellettavaan lähetysikkunaan, päästetään paketit suoraan puskuriin pudottamatta niitä lainkaan. Tarkastellaan seuraavaksi tämän vaiheen pseudokoodiesitystä kuvan 5 (Pan ym. 2013) avulla.

Burst Allowance Calculation:

Upon packet arrival

1. If $burst_allow > 0$

   $$enque\ packet\ bypassing\ random\ drop;$$

Upon dq_rate update

2. Update burst allowance:

   $$burst_allow = burst_allow - dq_int;$$

3. if $p = 0$; and both cur_del and old_del less than $ref_del/2$, reset $burst_allow$,

   $$burst_allow = max_burst;$$

Kuva 5. PIE-algoritmin toiminta hetkellisten piikkien kanssa. (Pan ym. 2013)

Mikäli lähetysaikaa hetkellisille piikeille on käytettävissä, päästetään tämän ajan puitteissa paketit suoraan jonoon pudottamatta niitä. Jonosta paketteja poimittaessa päivitetään tätä sallittua aikamäärää vähentämällä parametrina saatu arvo burst_allow nykyisestä jonon viiveestä dq_int. Kun jonon viive palautuu alle puoleen parametrina ref_del annetusta tavoitteesta ja pakettien pudotuksen todennäköisyys $p = 0$, päivitetään aikamäärä parametrina saaduksi arvoksi. Tämä mahdollistaa satunnaiset lyhytkestoiset piikit aiheuttamatta kuitenkaan merkittävää keskimääräisen jonotusviiveen nousua. Tällaisia lyhytkestoisia piikkejä ovat esimerkiksi verkkosivujen lataaminen HTTP-protokollalla. Tietoa siirretään nopeasti pieni määrä, minkä jälkeen tiedonsiirto lopetetaan.

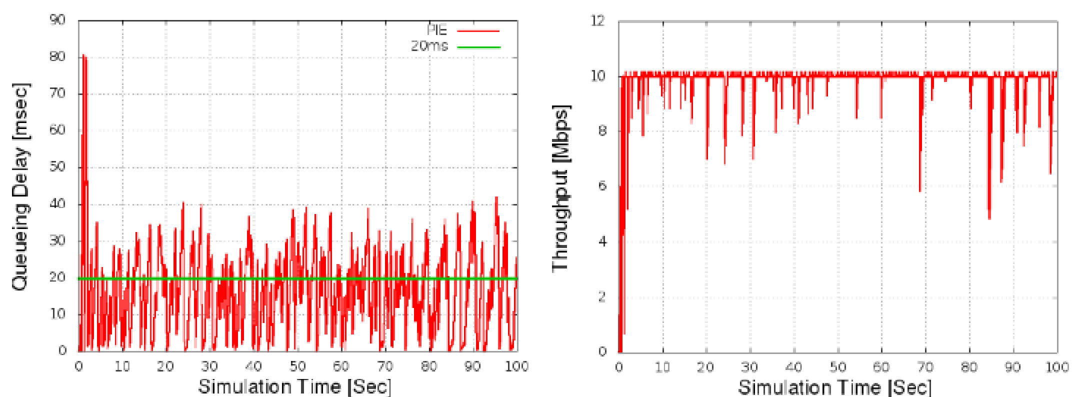
Pan ym. 2013 testasivat tutkimuksessaan algoritmin toimintaa ns-2-verkkosimulaattorilla testiympäristössä, jossa verkon pullonkaulan läpi johdetaan 5 kevyttä ja 50 raskasta TCP-tietovirtaa sekä yhdistelmästä, jossa verkon läpi johdetaan 5 TCP- ja 2

UDP-tietovirtaa. PIE-algoritmia käytettiin taulukon 1 mukaisilla parametreilla testiympäristössä.

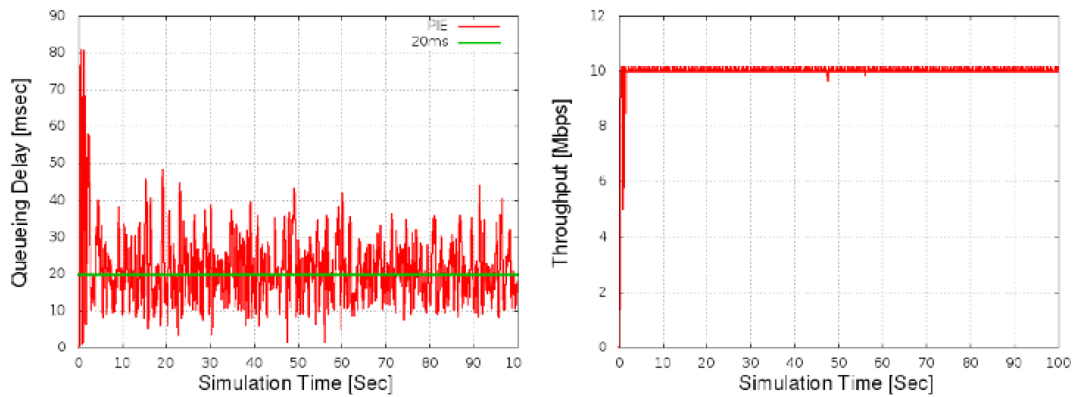
Taulukko 1. PIE-algoritmille annetut arvot ns-2 -simulaatiossa. (Pan ym. 2013)

Parametri:	**Arvo:**
delay_ref	20 ms
T_{update}	30 ms
α	0,125 Hz
β	1,25 Hz
dq_threshold	10 KB
max_burst	100 ms

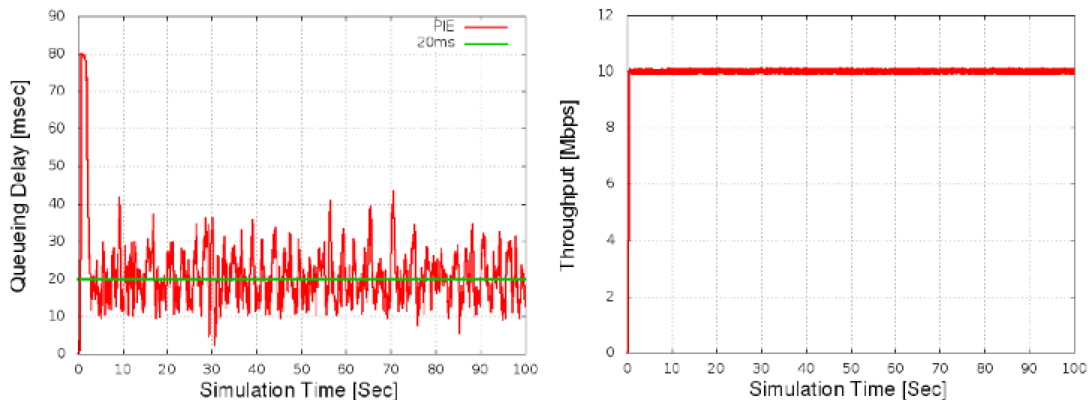
Taulukossa 1 esitetyt arvot ovat alkuperäisen PIE-algoritmin suositellut oletusarvot. Arvot voivat poiketa hieman algoritmin toteutuksesta riippuen. Tarkastellaan seuraavaksi PIE-algoritmin suorituskykyä erilaisissa tilanteissa kuvien 6, 7 ja 8 avulla.



Kuva 6. PIE ja 5 kevyttä TCP-tietovirtaa. (Pan ym. 2013)



Kuva 7. PIE ja 50 raskasta TCP-tietovirtaa. (Pan ym. 2013)



Kuva 8. PIE ja 5 TCP- sekä 2 UDP-tietovirtaa. (Pan ym. 2013)

Kuvista 6, 7 ja 8 voimme havaita, että PIE kykenee saturoimaan käytössä olevan kaistanleveyden lähes jatkuvasti pitäen kuitenkin jonotuksesta aiheutuvan viiveen kohtuullisena eli keskimäärin algoritmille parametrina annetussa tavoitteessa. Kuvien 6, 7 ja 8 vasemmanpuoleiset kuvaajat esittävät jonotuksesta aiheutuvaa viivettä. Vihreä vaakasuora viiva vasemmanpuoleisissa kuvaajissa edustaa parametrina annettua 20 ms tavoiteviivettä. Oikeanpuoleisissa kuvaajissa puolestaan näytetään toteutunut virtausnopeus. Viivettä esittävissä kuvaajissa olevat nousut ja laskut ovat tyypillisiä TCP-protokollalle silloin, kun käytetään pakettien hävikkiin perustuvaa ruuhkanhallintaa.

Simulaation perusteella PIE täyttää sille asetetut vaatimukset hyvällä menestyksellä (Pan ym. 2013). PIE on myös varsin kevyt laskennallisesti ja se voidaan toteuttaa joko laitteiston tasolla tai ohjelmallisesti. Algoritmista on olemassa Linux-toteutus. Kevey-

destä ja jonon tehokkaasta hallinnasta johtuen se on lisätty uusien DOCSIS 3.1 -kaapelimodeemijärjestelmien vaadittujen toiminnallisuuksien listalle, kuten jo aiemmin esitettiin.

3.1.3 CoDel

CoDel (Controlled Delay) on yksi uusimmista tulokkaista AQM-algoritmien rintamalla. CoDel pyrkii tunnistamaan jonotuksen aiheuttaman viiveen mittaamalla pakettien jonotusaikaa. Tätä aikaa mitataan aikaleimaamalla jokainen paketti ja mittaamalla tietyin aikavälein jonossa olevien pakettien todellista jonotusaikaa. Algoritmille annetaan parametreiksi hieman toteutuksen mukaan jonon suurin koko paketteina, suurin hyväksytty jonotusaika ja tarkastelujakso, jolla tätä edellä mainittua aikaa mitataan. Esimerkiksi CoDelin Linux-toteutuksessa algoritmille annetaan arvot limit, target ja interval (tc-codel). Algoritmin keskeinen suunnittelutavoite on toimia ilman monimutkaista parametrien säätämistä toisin kuin esimerkiksi alkuperäinen RED, joka vaatii tarkkaa hienosäätöä toimiakseen optimaalisesti. Tämän tavoitteen lisäksi algoritmin tavoitteena on pitää jonottamisen aiheuttama viive matalana mutta sallia kuitenkin hetkelliset piikit tietoliikenteessä sekä skaalautua hyvin erilaisiin tietoverkkoihin ja näissä käytettäviin laitteisiin (tc-codel).

CoDel mittaa suoraan yksittäisten pakettien jonotusaikaa sen sijaan, että se estimoisi jonon keskimääräistä pituutta RED:n tai PIE:n tapaan. Pakettikohtainen viiveen mittaus mahdollistaa verkkoyhteyden nopeudesta riippumattoman toiminnan ja käyttää puskuria tehokkaasti (Nichols ym. 2012). Algoritmi mittaa paikallisen jonon pienintä viivettä ja vertaa sitä parametrina annettuun arvoon. Niin kauan, kun viive pysyy parametrin sisällä tai jonossa on vähemmän kuin MTU-arvon verran dataa, paketteja ei pudoteta tai merkitä lainkaan. Paketteja aletaan pudottamaan tai merkitsemään siinä vaiheessa, kun jonon pienin viive ylittää tarkastelujakson ajaksi parametrina annetun halutun kohdearvon (tc-codel, Nichols ym. 2012).

Algoritmin toiminta voidaan jakaa kahteen keskeiseen toiminnallisuuteen. Näitä toiminnallisuuksia ovat pakettien lisääminen jonoon (eng. enqueue) ja pakettien poistaminen jonosta (eng. dequeue). Tarkastellaan seuraavaksi näitä toiminnallisuuksia pseudokoodiesityksiä apuna käyttäen kuvien 9 ja 10 avulla (Raghuvanshi ym. 2013).

```
On arrival of every packet :
if current_queue_size < queue_limit then
    Enqueue the packet
    Attach a timestamp in packet header
end
else
    Drop the Packet
end
```

Kuva 9. CoDel-algoritmin paketin lisäys jonoon. (Raghuvanshi ym. 2013)

Paketin saapuessa jonoon tarkastetaan, onko jonossa parametrina saadun raja-arvon ylittävä määrä paketteja. Mikäli paketteja on enemmän kuin jonon suurin sallittu pituus, pudotetaan paketti välittömästi. Muutoin paketti aikaleimataan ja lisätään jonoon. Paketin jonoon lisäämisen jälkeen algoritmi alkaa poimia paketteja jonosta. Kuvassa 10 esitetään poimintavaiheen pseudokoodi.

```
On departure of every packet :
dequeue_time = timestamp for dequeue time
sojourn_time = dequeue_time − enqueue_time
if inside the dropping state then
    if sojourn_time < target  or
    current_queue_size < MTU then
        Do not drop packets
        Come out of dropping state
    end
    else
        while dequeue_time ≥ next_drop_time do
            Drop the packet
            count = count + 1
            next_drop_time + = interval / √count
        end
    end
end
else if outside dropping state and first packet is
being dropped then
    Enter the dropping state
end
```

Kuva 10. CoDel-algoritmin paketin poiminta jonosta. (Raghuvanshi ym. 2013)

Jonosta poimittaessa paketteja. lasketaan jokaiselle paketille sen jonotusaika (eng. sojourn time). Tämä lasketaan vähentämällä nykyisestä ajanhetkestä pakettiin aiemmin liitetty aikaleima. Mikäli tämä oleskeluaika käsiteltävällä paketilla on suurempi kuin parametrina saatu tavoiteaika, siirtyy algoritmi paketteja pudottavaan tilaan. Tässä tilassa paketteja aletaan ennakoivasti pudottamaan tai merkitsemään jonosta poimittaessa vertaamalla poiminnan aloitusaikaa algoritmin laskemaan seuraavaan pudotushetkeen. Tämä hetki määräytyy sen mukaan, kuinka suuri tarkasteluväli on käytössä ja kuinka monta pakettia on jo pudotettu tässä tilassa. Algoritmi pysyy paketteja pudottavassa tilassa niin kauan, kunnes jonon tavoiteviive saavutetaan tai jonossa olevien

pakettien sisältämä datamäärä pienenee pienemmäksi kuin MTU (Raghuvanshi ym. 2013). Tilasiirtymien välillä algoritmin muuttujien arvot nollataan (Nichols ym. 2017).

Raghuvanshi ym. 2013 testasivat CoDel-algoritmin toimintaa ns-2 -verkkosimulaattorilla käyttäen algoritmia suositelluilla oletusparametrilla. Simulaation tuloksena oli, että algoritmia käyttäen saavutetaan korkea verkkoyhteyden käyttöaste ja alhainen puskuroinnin aiheuttama viive. Tutkimuksen perusteella CoDel parantaa merkittävästi verkossa esiintyvää viivettä täyttäen samalla AQM-algoritmien keskeiset tavoitteet. CoDel ei myöskään ole laskennallisesti erityisen raskas, joten se soveltuu hyvin käytettäväksi verkon aktiivilaitteilla.

3.1.4 FQ-CoDel

FQ-CoDel (Flow Queue CoDel) on hybridi, jossa yhdistyy CoDel-tyyppinen aktiivinen jononhallinta ja DRR-tyyppinen vuorontaminen – tarkemmin DRR++ -vuoronnin. Vuorontimen toiminnasta kerrotaan lisää myöhemmin tässä luvussa.

Algoritmin toimintaperiaate on sekoittaa siihen saapuvien tietovirtojen paketit useisiin yksittäisiin jonoihin, joihin sovelletaan CoDel-algoritmia. FQ-CoDelin keskeinen tarkoitus on jakaa käytettävissä oleva kapasiteetti jonojen kesken ja pitää näiden jonojen jonotusviive matalana. Algoritmi jakaa nämä sisäiset jonot erillisiin uusien ja vanhojen jonojen listoihin riippuen jonojen käyttäytymisestä. Listoja hyödynnetään liikenteen priorisoinnissa. Algoritmin tavoitteena on mahdollistaa suurten tiedonsiirtojen ja matalaa viivettä vaativien sovellusten käytön samassa verkossa siten, että ne eivät häiritse toisiaan (RFC 8290).

Algoritmi on geneerinen, tehokas ja lähes parametriton lähestymistapa, jossa yhdistyvät tietovirtoihin perustuva reilu vuorontaminen ja CoDel. Algoritmille annetaan parametreja mutta niiden kanssa ei tarvita erityistä hienosäätöä. Algoritmi on tällä hetkellä yksi tehokkaimmista työkaluista Bufferbloat-ilmiön ratkaisemisessa (RFC 8290).

FQ-CoDel-algoritmille annetaan parametreiksi tarkasteluväli (interval), tavoiteviive (target), jonossa olevien pakettien suurin mahdollinen lukumäärä (limit), poimittavien tavujen suurin määrä vuoroa kohti (quantum), CoDel-jonojen lukumäärä (flows) sekä toteutuksen mukaan se, että käytetäänkö pakettien pudottamista vai ECN-merkintää.

Parametrit interval ja target määrittävät sisäisten CoDel-jonojen vastaavat arvot luvun 3.1.3 mukaisella toimintalogiikalla. Limit-parametrin tarkoitus on rajoittaa algoritmin sisäisten jonojen yhteenlaskettujen pakettien lukumäärää siten, että muistinkäyttöä voidaan hallita (RFC 8290).

Toiminta aloitetaan alustamalla parametrien mukaiset CoDel-tyyppiset jonot. Kaikki FQ-CoDelin sisäiset jonot käyttävät samoja parametreja ja niitä käytetään koko toiminnan ajan. Mikäli arvoja halutaan muuttaa, tulee FQ-CoDel alustaa uudelleen halutuilla parametreilla. (RFC 8290).

Algoritmi jakaa siihen saapuvat paketit niiden otsikkotiedoista laskettavan hajautusarvon perusteella algoritmin sisäisiin jonoihin, joista jokainen käyttää CoDel-algoritmia jonotusviiveen hallintaan. Tietovirta on tyypillisesti viiden monikko, joka muodostuu lähde- ja kohde- IP-osoitteesta, lähde- ja kohde- porttinumeroista sekä kuljetusprotokollan tunnistenumerosta. Tietovirta voidaan määritellä myös näiden osajoukkona tai muilla menetelmillä (RFC 8290).

FQ-CoDel:n vuorontamisen tarkoituksena on antaa jokaiselle tietovirralle oma jono. Tämä ei tietysti todellisuudessa onnistu täydellisesti, sillä jonoja on vain rajallinen määrä. Tästä syystä algoritmi käyttää hajautukseen perustuvaa menetelmää pakettien jakamisessa jonoihin. Suositeltavaa on käyttää luvussa 3.1.3 esitettyä viiden monikkoa hajautusarvon laskentaan. Jonojen lukumäärä voidaan antaa parametrina. Algoritmin Linux-toteutuksessa näitä jonoja on oletusarvoisesti 1024 kappaletta. Suurin toteutuksen tukema jonojen määrä on 65535 kappaletta (RFC 8290).

Paketin lisääminen näihin jonoihin tapahtuu laskemalla Jenkins-hajautusfunktiolla, joka nykyisissä Linux-versiossa on lookup3.c (sch_fq_codel.c), hajautusarvo ja ottamalla tästä jonojen lukumäärällä jakojäännös. Jotta algoritmi ei olisi liian altis palvelunestohyökkäyksille, käytetään suolausta (eng. salt), johon käytettävä arvo valitaan satunnaisesti FQ-CoDelin alustusvaiheessa. Paketti asetetaan tämän jälkeen lasketun arvon mukaisen jonon loppupäähän. Jonoon lisäämisen jälkeen FQ-CoDel päivittää jonon sisältämän tavumääräisen laskurin vastaamaan nykyistä tilannetta. Mikäli kyseinen jono ei tässä vaiheessa vielä ole aktiivinen eli se ei ole uusien tai vanhojen jonojen listalla, lisätään kyseinen jono uusien jonojen listan loppupäähän ja kyseisen

jonon krediitit asetetaan vastaamaan parametrina saatua quantum-arvoa. Jotta algoritmi ei käyttäisi liikaa resursseja, verrataan jonoissa olevien pakettien kokonaislukumäärää parametrina saatuun suurimpaan hyväksyttävään määrään. Jos tämä raja saavutetaan, valitaan jono, jossa on eniten dataa tavumääräisesti ja pudotetaan tästä jonosta puolet sen sisältämistä paketeista kuitenkin niin, että pudotetaan suurimmillaan vain 64 pakettia kerrallaan (RFC 8290).

Jonot jaotellaan uusiin ja vanhoihin jonoihin algoritmin sisällä. Näillä termeillä tarkoitetaan sitä, kuinka aktiivinen jono on ollut viime aikoina. Uusilla jonoilla tarkoitetaan sellaisia jonoja, joiden tietovirrat ovat vasta alkaneet ja jotka eivät vielä kerrytä aktiivisesti jonoa. Vanhat jonot puolestaan ovat olleet aktiivisia jo useammalla vuorontimen vuorolla. Tämä jaottelumalli saa aikaan seurauksen, jossa sellaiset jonot, jotka eivät kerrytä quantum-parametrin määrän vertaa dataa ennen niiden tulemista valituiksi, poistetaan listalta heti, kun ne tyhjenevät. Poiston jälkeen mahdollinen tiedonsiirto näissä jonoissa olleista tietovirroista aiheuttaa sen, että jono päätyy taas uusien jonojen listalle. Tämä tarkoittaa käytännössä sitä, että hetkellisesti pienen määrän dataa tuottavat tietovirrat päätyvät usein näihin uusien jonojen listalle. Tämä puolestaan johtaa siihen, että esimerkiksi DNS-kyselyt, interaktiiviset ja kevyet tietovirrat, kuten DNS, SSH, VoIP ja verkkopelit tapaavat päätyä näihin jonoihin, kun taas pitkään jatkuneet ja raskaat tiedonsiirrot pysyvät yleensä vanhoissa jonoissa (RFC 8290).

Varsinainen pakettien poiminta näistä jonoista koostuu kolmesta välivaiheesta, joita ovat jonon valinta, poiminta tästä jonosta, CoDel-algoritmin käyttö jonon viiveen hallintaan sekä sisäinen kirjanpito. Sopiva jono valitaan käymällä ensin uusien jonojen listaa läpi järjestyksessä. Mikäli listaa läpikäydessä tullaan vastaan jonoon, jonka krediitit ovat loppu, siirretään tämä kyseinen jono vanhojen jonojen listalle ja annetaan sille uusia krediittejä quantum-parametrin verran ja jatketaan uusien jonojen listan läpikäyntiä. Tätä läpikäyntiä jatketaan, kunnes vastaan tulee jono, jolla on krediittejä. Tällöin tämä kyseinen jono valitaan. Tämä aiheuttaa käytännössä sen, että nopeasti tyhjenevät ja kevyttä tietovirtaa sisältävät jonot saavat suuremman prioriteetin kuin raskasta tiedonsiirtoa sisältävät jonot. Tämä puolestaan aiheuttaa sen, että jonotusviive pysyy matalana raskaiden, pitkään jatkuneiden tietovirtojen kustannuksella. Mikäli uusien jonojen listalta ei löytyisi sopivaa jonoa, siirrytään vanhojen jonojen listalle ja aletaan käymään sitä läpi vastaavalla toimintaperiaatteella (RFC 8290).

Kun sopiva jono on löydetty, alkaa kyseisen jonon CoDel hallita tätä jonoa. CoDel valitsee tästä jonosta ensimmäisen sopivan paketin ja palauttaa sen. Jonosta saatetaan pudottaa joitain paketteja, mikäli CoDel näkee sen tarpeelliseksi viiveen hallinnan kannalta. Jono voi mennä myös täysin tyhjäksi tämän pudotuksen seurauksena siten, että yhtään pakettia ei valita lähetettäväksi. Mikäli tällainen tilanne tapahtuu uusien jonojen ollessa kyseessä, siirretään kyseinen jono vanhojen jonojen listan loppuun. Jos jonoa ei siirrettäisi vaan se poistettaisiin, voisi se tulla heti seuraavalla vuorolla jälleen uusien jonojen alkuun, mikäli siihen sattuisi tulemaan paketteja. Tämä voisi pahimmassa tapauksessa jatkua loputtomasti. Huomionarvoista on, että tämä kyseinen jono tyhjeni siinä olleen viiveen vuoksi. Jos tyhjäksi mennyt jono oli jo vanhan listan jono, poistetaan tämä jono listalta. Kun sopiva paketti on löytynyt, poistetaan kyseisen jonon krediiteistä tämän paketin koon verran krediittejä ja lähetetään paketti verkkoon. Tämän jälkeen poimintaprosessi aloitetaan alusta (RFC 8290).

FQ-CoDel:n hienous on siinä, että tietovirrat jaetaan useisiin yksittäisiin jonoihin, joista jokaiseen sovelletaan erikseen CoDel:ia ja DRR-tyyppinen vuoronnin jakaa lähetysvuoroja näille jonoille järjestyksessä. Tällä saavutetaan se hyöty, että yksittäiset tietovirrat eivät dominoi jonoa, viive saadaan keskimäärin pysymään halutussa tavoitteessa ja uudet sekä kevyet tietovirrat saavat korkeamman prioriteetin lähetykselle, jolloin interaktiivisten sovellusten käytettävyys säilyy hyvänä verkon kuormituksesta riippumatta. Huomioitavaa on, että vaikka verkossa olisikin erilaista kuormaa, ei FQ-CoDel pudota paketteja, ellei pudotusta tarvita viiveen hallitsemiseksi. Tässäkin tapauksessa algoritmin suorittamat pudotukset kohdistetaan ensisijaisesti niihin jonoihin, jotka eniten aiheuttavat viiveen nousemista. Mikäli verkon solmu kykenee välittämään kaiken siihen saapuvan tiedon eteenpäin, ei pakettien pudotusta tarvita.

Algoritmin heikkous puolestaan on se, että massiivisilla tietovirtojen määrillä useita tietovirtoja voi päätyä samaan CoDel-jonoon. Tämä johtuu käytössä olevasta hajautusarvon laskennasta ja rajoitetusta jonojen määrästä ja voi aiheuttaa ongelmia siinä tapauksessa, jossa raskaat tietovirrat päätyvät samoihin jonoihin kevyiden tietovirtojen kanssa useasti. Onkin syytä arvioida se, kuinka paljon näitä tietovirtoja algoritmia soveltavan solmun läpi kulkee samaan aikaan ja tarvittaessa hienosäätää jonojen lukumäärää tai käyttää muita menetelmiä ongelman ratkaisemiseksi. Yksi esimerkki tällaisesta menetelmästä on jakaa solmuun saapuvat tietovirrat jollain menetelmällä useisiin

FQ-CoDel:a käyttäviin jonoihin. Esitämme jäljempänä esimerkin todellisesta käyttötapauksesta, jossa käytämme useita FQ-CoDel-jonoja hallitsemaan yksittäistä fyysistä verkkolinkkiä.

3.2 Kuljetuskerroksen ruuhkanhallinta

Yksi syy suureen viiveen nousuun rasituksessa on Internetin alusta asti ollut heikosti toimiva ruuhkanhallinta. Internetin alkuaikoina ei ollut mitään ruuhkanhallintaa, kunnes ongelmien tullessa esiin kehitettiin TCP-protokollaan, joka on käytetyin tiedonsiirtoprotokolla Internetissä, oma ruuhkanhallintaominaisuus. Ruuhkanhallinnan tarkoituksena on tunnistaa tietoverkon kapasiteetin ylitys ja ryhtyä tarvittaviin toimenpiteisiin ruuhkan välttämiseksi. Ruuhkanhallinta tapahtuu tyypillisesti lähettävässä päässä. TCP pyrkii pitämään verkkoyhteyden lähellä sen suurinta mahdollista nopeutta. Perinteistä TCP:n ruuhkanhallintaa käyttäen lähetys aloitetaan hitaasti ja nopeutta kasvatetaan jatkuvasti niin kauan, kunnes häviöitä alkaa esiintyä (Gettys ym. 2011). Tämän jälkeen aloitetaan toimenpiteet ruuhkan välttämiseksi (eng. congestion avoidance). Tämä johtaa siihen, että yhteyden pullonkaulan puskuri ehtii täyttyä ennen kuin lähettäjä ehtii hidastaa lähetysnopeutta (Gong ym. 2014). Tämä puolestaan johtaa lisääntyneeseen viiveeseen. Ruuhkanhallinta on tarpeellista, sillä ilman minkäänlaista ruuhkanhallintaa tukkeutuisi vastaanottajan verkkoyhteys todella helposti, mikäli lähettäjällä vain olisi enemmän kapasiteettia käytettävissään kuin vastaanottajalla (Alfredsson ym. 2013).

Tietoliikenneyhteydellä on aina tasan yksi hitain linkki eli pullonkaula jollain verkon välillä. Yhteyden pullonkaula määrittää suurimman mahdollisen nopeuden, jolla tietoa voidaan siirtää polulla, jossa tämä pullonkaula on osallisena. Tietoa ei voida siirtää solmujen välillä nopeammin kuin niiden välisen polun hitaimman linkin nopeudella. Tämä aiheuttaa sen, että verkon pullonkaulaan muodostuu jonoa. Jono lyhenee vain silloin, kun pullonkaula voi välittää tietoa nopeammin kuin se vastaanottaa sitä. Kun siirtonopeus on suurimmillaan, kaikki ylemmän tason linkit omaavat suuremman lähetysnopeuden kuin pullonkaulan kaistanleveys, joten jono alkaa siirtyä kohti verkon pullonkaulaa (Cardwell ym. 2016).

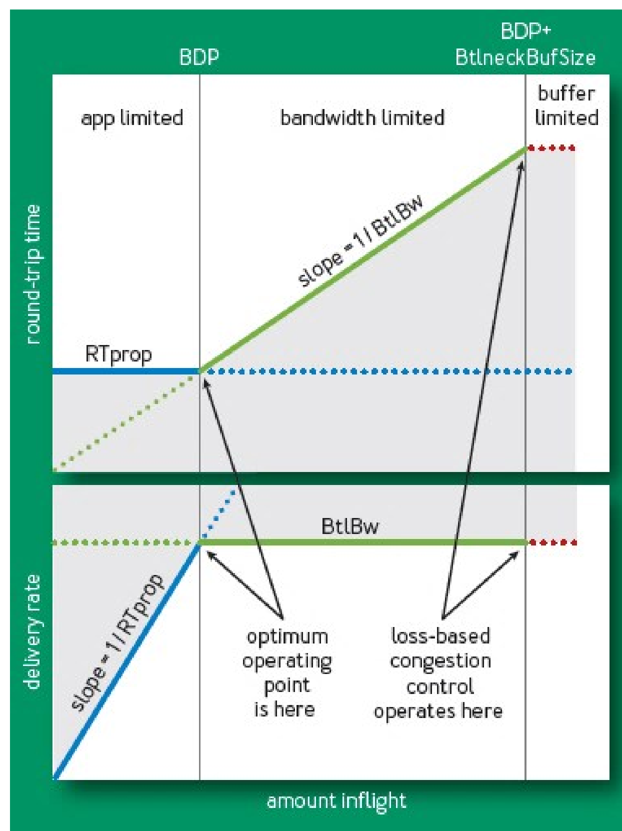
Nykyisin käyttöjärjestelmien oletusasetuksilla toimiva TCP:n ruuhkanhallinta on optimoitu korkeille tiedonsiirtonopeuksille ja pyrkii aina saavuttamaan maksimaalisen kaistanleveyden (Alfredsson ym. 2013). TCP-protokolla pyrkii oletuksena jakamaan kaiken käytössä olevan kapasiteetin tasaisesti muiden TCP-yhteyksien kanssa. Internetissä tietoa välitetään kuitenkin useilla eri protokollilla, joista kaikissa ei ole omaa ruuhkanhallinta -ominaisuutta. TCP ei myöskään ole tietoinen muiden protokollien liikenteen tilasta verkossa. Mikäli tiedonsiirtoprotokollasta puuttuisivat ruuhkanhallintaominaisuudet, jäisi verkon mahdollisen ruuhkautumisen havaitseminen sovelluskerrokselle. Esimerkiksi UDP-protokolla ei pidä sisällään minkäänlaista ruuhkanhallintaa eikä se myöskään ota kantaa siihen, siirtyykö tieto luotettavasti paikasta toiseen (Postel 1980).

Perinteinen ruuhkanhallinta TCP-protokollassa perustuu pakettien hävikin tunnistamiseen. Tyypillisesti paketteja katoaa joko siitä syystä, että paketti vaurioituu lähetyksessä tai verkko on niin ruuhkautunut, että paketti on kadonnut jollain välillä. Pakettien vaurioituminen on kuitenkin melko harvinaista. Jacobson esittää tutkimuksessaan, että todennäköisyys pakettien vaurioitumiselle on alle 1 %. Mikäli pakettien hävikkiä voidaan lähes aina pitää merkkinä verkon ruuhkautumisesta ja aikakatkaisua paketin häviämisenä niin pakettien häviäminen on hyvä ruuhkautumisen merkki (Jacobson 1988). Tällä olettamuksella ruuhkanhallinta on toiminut tähän päivään saakka aiheuttaen ongelmia verkkoviiveen kanssa. 1980-luvulla tämä pakettien hävikkiin perustuva ruuhkautumisen havaitseminen oli relevanttia, mutta nykypäivänä tämä periaate ei ole enää yhtä toimiva. Verkkokorttien nopeudet ovat kasvaneet megabiteistä gigabitteihin ja muistipiirien kapasiteetti kilotavuista gigatavuihin vuosien saatossa, mikä on saanut aikaan sen, että pakettien hävikin ja verkon todellisen ruuhkautumisen välinen yhteys on nykyisin häilyvää (Cardwell ym. 2016). Vaihtoehtoisia ruuhkautumisen havaitsemismenetelmiä olisi siis hyvä tarkastella ja ottaa niitä käyttöön tiedonsiirtoprotokollissa.

TCP:n näkökulmasta jopa monimutkaisen verkon tapauksessa kahden pisteen väli näkyy protokollalle vain yhtenä linkkinä, jolla on yksi edestakaisen matkan määräämä viive ja pullonkaulan määrittämä siirtonopeus. Yhteyden siirtonopeuden määrittää pullonkaulan kaistanleveys (jäljempänä BtlBw) ja päästä-päähän edestakaiseen matkaan vaadittu aika (jäljempänä RTprop). RTprop on verkkoyhteyden fyysinen ominaisuus,

jota voidaan estimoida mittaamalla RTT-arvoa (round-trip time) pakettien toimitusten välillä (Cardwell ym. 2016).

Havainnollistavasta kuvasta 11 (Cardwell ym. 2016) voidaan nähdä näiden rajoitteiden vaikutus tiedonsiirtonopeuteen ja lennossa olevien pakettien määrään. Lennossa olevilla paketeilla tarkoitetaan sellaisia paketteja, jotka on lähetetty mutta joihin ei ole vielä saatu TCP-protokollan mukaista ACK-vastausta. Kuvan 11 tarkoituksena on havainnollistaa optimaalinen tiedonsiirtonopeus näiden protokollan rajoitteiden puitteissa sekä näyttää kuinka ruuhkautuminen tapahtuu.



Kuva 11. TCP-yhteyden optimaalinen toimintapiste. (Cardwell ym. 2016)

Kuvassa 11 RTprop -viivan muodostama alue kuvaa viiveen aiheuttamaa rajoitetta, BtlBw -viiva kaistanleveyden aiheuttamaa rajoitetta ja oikeassa yläkulmassa oleva katkoviiva puskurin koon aiheuttamaa rajoitetta. Varjostetut alueet kuvaavat edellä mainittujen rajoitteiden rikkomista. Näiden protokollan rajoitteiden välillä siirtyminen voidaan jakaa kolmeen eri osa-alueeseen, joita ovat: sovellus-, kaistanleveys- ja puskurointirajoitteisuus. Kun lähetettävää dataa ei ole tarpeeksi täyttämään yhteyttä, määrää viive eli RTprop käytettävän tiedonsiirtonopeuden. Muussa tapauksessa rajoitteena

on kaistanleveys eli BtlBw. Näiden kahden rajoitteen leikkaus kuvastaa arvojen tuloa eli linkin todellista kapasiteettia, jota kutsutaan BDP-arvoksi. BDP on siis datamäärä, jonka pullonkaula voi kuljettaa yhdellä edestakaisella matkalla RTT:n määrittämässä ajassa siten, että data voidaan todella välittää verkon läpi ilman ylimääräistä puskurointia. Tämä arvo on tärkeä käsite, sillä tätä suurempi määrä lennossa olevia paketteja ylittää linkin kapasiteetin ja saa aikaan ylimääräistä puskurointia verkon pullonkaulaan. Tämä tarkoittaa käytännössä sitä, että viive alkaa nousta, mikäli lennossa olevien pakettien määrä ylittää BDP:n. Perinteisesti paketteja aletaan pudottamaan vasta silloin, kun niiden määrä ylittää pullonkaulan puskurin koon. Ruuhkanhallinta pyrkii vaikuttamaan siihen, kuinka paljon pidemmälle etenemme tästä BDP:n määrittämästä optimaalisesta pisteestä (Cardwell ym. 2016).

Ruuhkanhallinta koostuu kahdesta komponentista, joita ovat loppupisteiden välinen mekanismi viestiä verkon ruuhkautumisesta ja politiikka, jota sovelletaan ruuhkautumisen aikana. Koska ruuhkautuminen tapahtuu eksponentiaalisesti, olisi sen tunnistaminen ajoissa tärkeää. Aikaisessa vaiheessa vain pieni muutos lähetysikkunaan riittää korjaamaan ongelman (Jacobson 1988).

Luvuissa 3.2.1-3.2.2 esitellään muutama jonotusviiveen matalana pitävä ruuhkanhallinta-algoritmi. Tiedonsiirtoprotokollana on TCP, ellei toisin mainita. Lukujen 3.2.1-3.2.2 tarkoituksena on esitellä pakettien hävikkiin perustuville ruuhkanhallinta-algoritmeille vaihtoehtoisia menetelmiä, jotka soveltuvat paremmin pitämään verkossa esiintyvän viiveen matalana.

3.2.1 LEDBAT

LEDBAT on ruuhkanhallinta-protokolla, jonka tarkoitus on siirtää tietoa taustalla siten, että tiedonsiirrosta ei aiheudu viiveen nousua ja muut tiedonsiirrot saavat suuremman prioriteetin verkossa. LEDBAT kehitettiin alun perin BitTorrent-protokollan käyttöön mutta nykyisin siitä on olemassa myös TCP-toteutus. Koska LEDBAT siirtää tietoa matalalla prioriteetilla, on se erinomainen vaihtoehto esimerkiksi käyttöjärjestelmäpäivitysten jakamiseen organisaation tietoverkossa. Matalan prioriteetin tiedonsiirto ei aiheuta merkittävää viiveen nousua, joten reaaliaikaisuutta edellyttävät sovellukset voivat jatkaa toimintaansa normaalisti suurtenkin tiedonsiirtojen aikana. Matala prioriteetti tarkoittaa myöskin sitä, että muut yhteydet saavat korkeamman prioriteetin.

Yksi tällainen käyttötapaus voisi olla esimerkiksi P2P-lataus LEDBATilla ja videoneuvottelu. Tässä tapauksessa taustalla oleva lataus ei aiheuta käytettävyysongelmaa esimerkiksi videoneuvotteluun, sillä neuvottelu saa suuremman osan käytettävästä kapasiteetista (Rossi ym. 2010, technet).

LEBAT mittaa tiedonsiirron aikana viivettä vain yhteen suuntaan. Tätä mittaustulosta käytetään estimoimaan jonotuksen aiheuttamaa viivettä. Mikäli viiveessä havaitaan kasvua, hidastetaan lähetysnopeutta. Tämä mahdollistaa nopeamman ruuhkautumisen havaitsemisen kuin perinteiset hävikkiin perustuvat ruuhkanhallinta-algoritmit (Rossi ym. 2010).

Tarkastellaan seuraavaksi hieman tarkemmin tämän ruuhkanhallinnan teknistä toteutusta analysoimalla algoritmin pseudokoodia. Kuvassa 12 (Rossi ym. 2010) esitetään algoritmin pseudokoodi.

```
on data_packet @ RX:
   remote_timestamp = data_packet.timestamp
   acknowledgement.delay =
       local_timestamp() - remote_timestamp

on acknowledgement @ TX:
   current_delay = acknowledgement.delay
   base_delay = min(base_delay, current_delay)
   queuing_delay = current_delay - base_delay
   off_target = TARGET - queuing_delay
   cwnd += GAIN * off_target / cwnd
```

Kuva 12. LEDBAT-algoritmin pseudokoodiesitys. (Rossi ym. 2010)

Kun paketti vastaanotetaan, tarkastetaan paketissa oleva aikaleima ja verrataan sitä nykyiseen aikaan. Tällä menetelmällä saadaan laskettua viive lähettäjän ja vastaanottajan välillä. Tämä ei ole kuitenkaan edestakainen viive vaan pelkästään yksisuuntainen. Syy yksisuuntaisen viiveen käyttämiselle RTT:n sijaan on se, että yksisuuntaisessa viiveessä paluumatkalla esiintyvä muu liikenne tai epäsymmetrinen reititys ei häiritse mittausta (Rossi ym. 2010).

Lähettäjä aikaleimaa paketit lähetysajankohdalla ja arvioi verkon jonotuksesta aiheutuvaa viivettä laskemalla ominaisviiveen ja vertaamalla sitä nykyiseen viiveeseen. Ominaisviive lasketaan edellisen ja nykyisen mittauskerran välillä valitsemalla näistä kahdesta pienempi arvo. Tätä arvoa verrataan parametrina saatuun tavoitteeseen, joka

on tyypillisesti 25 ms (Rossi ym. 2010). Viiveen vaihteluun perustuen muutetaan lähetysikkunaa joko suuremmaksi tai pienemmäksi riippuen viiveen vaihtelun tyypistä.

LEDBAT antaa muille tietovirroille suuremman prioriteetin, joten se häiritsee muuta liikennettä mahdollisimman vähän mutta pystyy kuitenkin käyttämään kaiken käytettävissä olevan kapasiteetin. Tästä syystä se soveltuu erinomaisesti käytettäväksi niin P2P-protokollissa, kuten esimerkkinä annetussa BitTorrentissa kuin myös muissa suuremmissa tiedonsiirroissa, joilla ei kuitenkaan haluta tukkia verkkoa. Koska LEDBAT mittaa aktiivisesti viivettä ja hidastaa lähetystään heti viiveen kasvaessa, ei sen käyttö aiheuta suurta puskurointia, kuten perinteinen TCP:n ruuhkanhallinta. Kaikkea mahdollista liikennettä ei kuitenkaan kannata siirtää käyttäen kyseistä protokollaa, sillä se on suunniteltu nimenomaan taustalla tapahtuvaan matalan prioriteetin tiedonsiirtoon.

3.2.2 BBR

BBR on uusi TCP:n ruuhkanhallinta-algoritmi, joka kehitettiin Googlen toimesta vastaamaan nykyaikaisen tiedonsiirtoverkon tarpeita. Hävikkiin perustuvat ruuhkanhallinta-algoritmit toimivat aiemmin esitetyn kaistanleveyden rajoitteen puolella. Näillä algoritmeilla saavutetaan kyllä pullonkaulan kaistanleveys mutta tämä johtaa tyypillisesti lisääntyneeseen viiveeseen ja hävikkiin. Pitkään uskottiin, että sellaista hajautettua algoritmia ei voitaisi kehittää, joka toimisi kuvan 11 osoittamassa optimaalisessa pisteessä, kunnes Google alkoi tutkimaan ongelmaa uudelleen (Cardwell ym. 2016).

Googlella analysoitiin useita tunteja päivässä vuosien ajan TCP-pakettien otsikkotietoja eri puolilta maailmaa otetuista pakettikaappauksista. Analysoinnissa kiinnitettiin erityisesti huomiota RTprop- ja BtlBw-rajoitteisiin. Koska yksittäisellä mittauksella ei saada tarkkaa kuvaa yhteyden käyttäytymisestä, alettiin tutkia, voisiko ongelmaa lähestyä jatkuvalla mittaamisella. Jatkuvalla mittaamisella ja kehittyneillä hallintajärjestelmillä voitaisiin saada aikaan hajautettu ruuhkanhallintaprotokolla, joka reagoisi todelliseen ruuhkautumiseen eikä pakettien hävikkiin tai pelkkään viiveeseen (Cardwell ym. 2016).

TCP-protokolla saavuttaa suurimman mahdollisen siirtonopeuden ja pienimmän viiveen silloin, kun pullonkaulaan saapuvien pakettien siirtonopeus on sama kuin BtlBw ja lennossa olevan datan määrä on sama kuin yhteyden BDP. Ensimmäinen ehto pitää

huolen siitä, että linkki voi saavuttaa täyden käyttöasteen. Toinen ehto taas varmistaa sen, että lennossa on sopiva määrä dataa, jotta yhteys ei kuihdu datan puutteen vuoksi eikä myöskään tukkeudu liiasta datasta. Se, että siirtonopeus on täsmälleen BtlBw ei tarkoita, etteikö pullonkaulaan tulisi jonoa vaan sitä, että jonon koko ei muutu nykyisestä tilasta juuri tällä kyseisellä mittaushetkellä. Seuraavalla mittauskerralla jonon koko on jo voinut kasvaa, sillä verkon ominaisuudet ovat voineet muuttua tässä ajassa. Kumpikaan näistä ehdoista ei siis yksistään riitä optimaalisen siirtonopeuden ja viiveen saavuttamiseen vaan optimaalinen tilanne TCP-protokollan tapauksessa saavutetaan vain silloin, kun molemmat ehdot täyttyvät samaan aikaan. Koska nämä kaksi keskeistä TCP-yhteyden suorituskykyyn vaikuttavaa suuretta muuttuvat jatkuvasti tietoverkkojen hajautetun luonteen vuoksi, täytyy näitä mitata jatkuvasti. Millä tahansa ajanhetkellä t, mitattu suure RTT muodostuu kaavan 3 mukaisesti:

$$RTT_t = RTprop_t + n_t \qquad \text{(kaava 3)}$$

Esimerkiksi jonottaminen ja vastaanottajan viivästyneet kuittaukset aiheuttavat mitattuun viiveeseen vaihtelua. Tätä vaihtelua kuvastaa kaavan termi n. RTprop on yhteyden fyysinen ominaisuus, joka muuttuu vain silloin, kun polussa tapahtuu muutoksia. RTprop-arvoa voidaan estimoida varsin tehokkaasti kaavalla 4 (Cardwell ym. 2016):

$$\widehat{RTprop} = RTprop + min(n_t) = min(RTT_t)\ \forall t \in [T - W_R, T] \qquad \text{(kaava 4)}$$

Käytännössä RTprop-arvoa siis estimoidaan mittaamalla jollain aikaikkunalla pienintä viivettä paketin toimituksen ja siitä saadun kuittauksen välillä. Tällainen aikaikkuna on tyypillisesti kymmenistä sekunneista minuutteihin.

TCP-protokolla ei itsessään edellytä kaistanleveyden mittaamista, mutta sitä voidaan estimoida pakettien toimituksen seuraamisella. Kun kuittaus lähetettyyn pakettiin saapuu, voidaan tietää, että toimitus onnistui. Mittaamalla tätä onnistuneiden toimitusten siirtonopeutta saadaan hyvä estimaattori pullonkaulan kaistanleveydelle. Mitattu suure on pienempi tai yhtä suuri kuin todellinen pullonkaulan kaistanleveys ja sitä voidaan estimoida kaavalla 5 (Cardwell ym. 2016):

$$\widehat{BtlBw} = max(deliveryRate_t)\ \forall t \in [T - W_B, T] \qquad \text{(kaava 5)}$$

Kaavalla 5 mitataan maksimaalista toimitusnopeutta aikaikkunassa *t*, joka on tyypillisesti noin 6 – 10 kertaa RTT-arvon suuruinen. TCP:n täytyy joka tapauksessa mitata RTT-arvoa pitämällä kirjaa jokaisen paketin lähetysajasta. BBR käyttää tätä hyödykseen sillä lisäyksellä, että RTT:n lisäksi mitataan samalla kertaa toimitusnopeus. Toimitusnopeudella tarkoitetaan onnistunutta siirtonopeutta eli kaikkiin lähetettyihin paketteihin täytyy saada kuittaus, jotta lähetystä voidaan pitää onnistuneena. Näin jokaisella vastaanotetulla kuittauksella saadaan laskettua estimaatit sekä RTprop- että BtlBw-arvoille. Arvot ovat kuitenkin toisistaan riippumattomia – esimerkiksi reititysmuutos voi aiheuttaa RTprop-arvoon muutoksen mutta pullonkaulan kaistanleveys voi pysyä samana (Cardwell ym. 2016).

BBR koostuu kahdesta keskeisestä toiminnallisuudesta: RTprop- ja BtlBw-arvojen estimoinnista sekä tahdistetusta pakettien lähetyksestä. Tahdistus on algoritmin toiminnan kannalta ehdottoman tärkeä, sillä tahdistuksen kerrointa (pacing_gain) muuttamalla säädetään todellista lähetysnopeutta. Tahdistusnopeus on jokin kiinteä tiedonsiirtonopeus riippuen algoritmin toteutuksesta. Tätä kiinteää arvoa skaalataan erikseen laskettavalla kertoimella sen mukaan, mihin suuntaan siirtonopeutta halutaan muuttaa. BBR-toteutusten täytyy käyttää tahdistusta, sillä muutoin tämän kertoimen muuttaminen ei aiheuta toivottua tulosta. Tahdistusta käyttämällä lähetysnopeus saadaan täsmäämään verkon pullonkaulan kanssa. Jokaisen lähetetyn paketin välissä pidetään tauko. Tauon pituus riippuu lähetetyn paketin koosta, BtlBw-arvosta ja tahdistuksen kertoimesta. Pakettien lähetysaika lasketaan kaavan 6 mukaisesti (Cardwell ym. 2016):

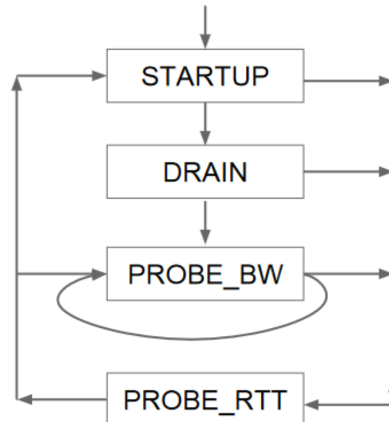
$$nextSendTime = now + packet.size/(pacing_gain * BtlBw) \quad \text{(kaava 6)}$$

Jokaisella saapuneella kuittausviestillä lasketaan RTT vähentämällä lähetysaika nykyisestä ajasta. Mitatulla RTT-arvolla päivitetään samalla RTprop-estimaattoria kaavan 4 mukaisesti. Koska tiedämme tarkkaan lähetetyn datan määrän, voimme laskea toimitusnopeuden tietyllä ajanjaksolla jakamalla kuitatun datan määrän käytetyllä ajalla.

BBR minimoi viiveen pitämällä suurimman osan ajasta vain BDP-arvon verran paketteja lennossa. Tämä aiheuttaa sen, että pullonkaula siirtyy lähettäjän päähän, joten mahdolliset BtlBw:n todelliset muutokset jäävät havaitsematta, ellei tahdistusnopeutta

muuteta hieman säännöllisin väliajoin. BBR muuttaa tietyin aikavälein noin yhden RTprop-arvon ajaksi pacing_gain-arvon suuremmaksi kuin 1. Tämä nostaa lennossa olevien pakettien määrää ja täten myös siirtonopeutta hetkellisesti. Mikäli edellisen toimenpiteen aikana ei havaita muutosta BtlBw:n estimaatissa mutta RTT on noussut, tiedetään, että verkon pullonkaulaan on syntynyt jonoa. Jonon syntyminen tapahtuu käytännössä tämän mittauksen epätarkkuuden vuoksi. Tässä tilanteessa suoritetaan korjaava liike muuttamalla pacing_gain-arvoa pienemmäksi kuin 1 seuraavalla lähetyskerralla yhtä pitkäksi aikaa kuin sitä edellisen kerran nostettiin. Jos taas tilanne onkin se, että pullonkaulan todellinen kapasiteetti onkin kasvanut niin tahdistuksen muutos aiheuttaa pakettien onnistuneiden toimitusten nopeuden kasvua, johon algoritmi vastaa käytännössä kasvattamalla lähetysnopeutta entisestään (Cardwell ym. 2016).

Algoritmi koostuu neljästä tilasta, joita ovat aloitus, tyhjennys, pullonkaulan kaistanleveyden selvitys ja RTT:n mittaus. Kuvassa 13 esitellään algoritmin tilat sekä niiden väliset siirtymät. Kuva 13 on lainattu Cardwellin ym. IETF-muistiosta (Cardwell ym. 2017).

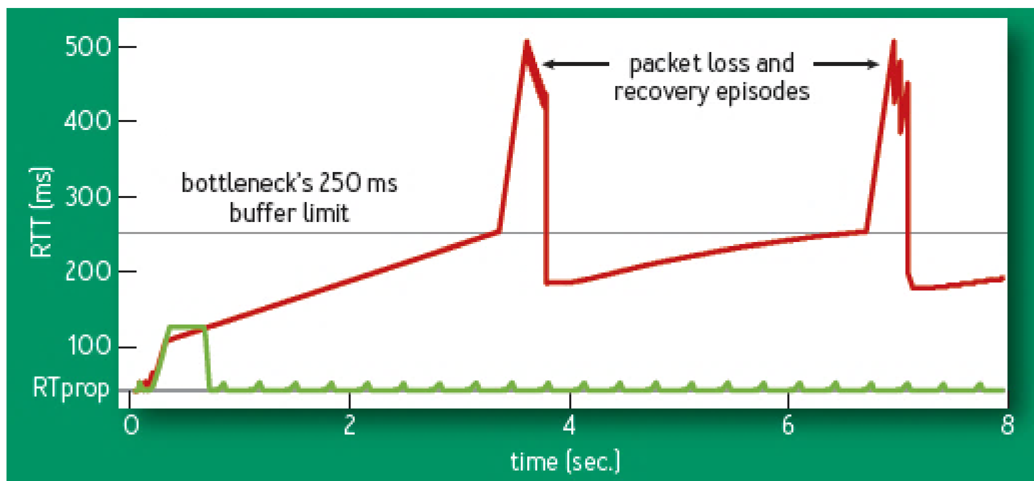


Kuva 13. BBR-algoritmin tilat. (Cardwell ym. 2017)

Algoritmi aloittaa toimintansa lähettämällä paketteja tuplaten lähetysnopeuttaan jatkuvasti siihen saakka, kunnes pakettien toimitusnopeus ei enää kasva (STARTUP-tila). Näin saadaan nopeasti havaittua BtlBw mutta haittana tällä eksponentiaalisella lähetysnopeuden kasvatuksella on jonon muodostuminen. Jono puretaan nopeasti pois tyhjennys -tilassa pudottamalla lähetysnopeutta hetkellisesti (DRAIN-tila). Nopeutta puolitetaan niin kauan, että sopiva nopeus on saavutettu. Kun aloituksessa muodostunut jono on tyhjennetty, siirtyy algoritmi jatkuvaan BtlBw-mittaukseen (PROBE_BW-

tila), jossa se viettää suurimman osan ajastaan. Tässä tilassa algoritmi säätelee pacing_gain-arvoaan aiemmin kuvatun logiikan mukaisesti. Säännöllisin väliajoin käydään myös kuvan viimeisessä tilassa (PROBE_RTT-tila), jonka tarkoitus on selvittää, onko RTprop muuttunut. Tässä tilassa pacing_gain-arvoa pienennetään sen verran, että lennossa olevien pakettien määrä on alle BDP. Tämä hidastaa lähetysnopeutta, jolloin viive saada mahdollisimman tarkasti mitattua. Tästä tilasta siirrytään jälleen takaisin BtlBw-mittaukseen (Cardwell ym. 2016). Aloitustilaan palataan takaisin vain silloin, kun yhteys joudutaan aloittamaan uudestaan.

Tarkastellaan seuraavaksi BBR- ja CUBIC-ruuhkanhallinnan käyttäytymistä ja näiden välistä eroa mitattuun viiveeseen. Kuvassa 14 esitetään TCP-yhteyden ensimmäisten 8 sekunnin mitattu viive edellä mainituilla ruuhkanhallinta-algoritmeilla sekä verrataan näiden algoritmien keskeistä eroa käyttäytymisessä ja suorituskyvyssä. Vertailukohtana voidaan käyttää verkon ominaisviivettä RTprop. Kuvassa punainen väri kuvastaa CUBIC- ja vihreä BBR-ruuhkanhallintaa. Ominaisviivettä kuvataan harmaalla suoralla.

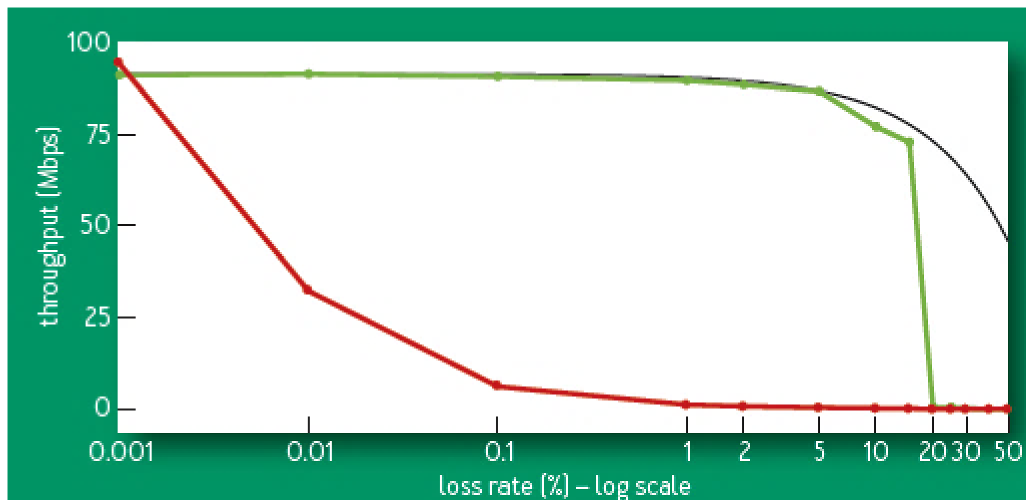


Kuva 14. BBR ja CUBIC suhteessa mitattuun viiveeseen. (Cardwell ym. 2016)

Kuten aiemmin jo todettiin, hävikkiin perustuvat ruuhkanhallinta-algoritmit lisäävät puskuroinnin aiheuttamaa viivettä, sillä ne havaitsevat ruuhkautumisen ja alkavat pudottamaan paketteja vasta silloin, kun puskurit ovat jo täyttyneet. Tämä voidaan havaita kuvassa 14 valtavan kokoisina kahtena piikkinä. BBR eroaa perinteisellä toimintaperiaatteella toimivasta CUBIC:sta siten, että se aiheuttaa vain lyhyeksi aikaa viiveen nousua, jonka jälkeen yhteyden viive on todella tasaista koko mitatun 8 sekunnin

ajan. Kuten kuvasta 14 voidaan havaita, pitää BBR tiedonsiirrosta aiheutuvan viiveen hyvin lähellä verkon ominaisviivettä koko tiedonsiirron ajan, kun taas CUBIC aiheuttaa jatkuvaa vaihtelua.

BRR-ruuhkanhallinta saa aikaan suurimman mahdollisen siirtonopeuden pitäen puskuroinnin aiheuttaman viiveen mahdollisimman pienenä. Tämän lisäksi algoritmi sopii hyvin sellaisiin verkkoihin, joissa on paljon hävikkiä. Perinteisesti esimerkiksi CUBIC-ruuhkanhallinnalla toimiva TCP-protokolla ei siedä hävikkiä varsinkaan suuren viiveen omaavilla verkon väleillä, kuten esimerkiksi mantereiden välisillä yhteyksillä. Tarkastellaan seuraavaksi kuvan 15 avulla BBR- ja CUBIC-ruuhkanhallinnan käyttäytymistä suhteessa pakettien hävikkiin.



Kuva 15. BBR ja CUBIC sekä pakettihävikki verkossa. (Cardwell ym. 2016)

Kuvassa 15 alempi eli punainen viiva kuvastaa CUBIC-ruuhkanhallinnan hävikin ja siirtonopeuden välistä riippuvuutta. Vihreä viiva puolestaan kuvastaa BBR-ruuhkanhallintaa. Kuvasta voidaan havaita, että BBR sietää pakettien hävikkiä huomattavasti paremmin. Jo pieni määrä pakettien hävikkiä saa aikaan huomattavan hidastumisen tiedonsiirrossa silloin, kun käytössä on hävikkiin perustuva ruuhkanhallinta-algoritmi.

Google on käyttänyt BBR-ruuhkanhallintaa omissa palveluissaan ja todennut sen parantaneen siirtonopeutta merkittävästi (Cardwell ym. 2016). Siirtonopeuden lisäksi viive on parantunut merkittävästi varsinkin kehittyvillä maanosilla. Koska BBR toimii puhtaasti lähettäjän päässä eikä se vaadi TCP-protokollaan muutoksia, on sen käyttöönotto helppoa. BBR on implementoitu muun muassa Linux-ytimeen (Cardwell ym.

2016). Käyttö tosin edellyttää vielä toistaiseksi ominaisuuden manuaalista päälle kytkentää, sillä moni Linux-jakelu käyttää TCP:n ruuhkanhallintana edelleen CUBIC:ia.

3.3 Vuorontaminen

Vuorontamisella tarkoitetaan tietoliikenteen tapauksessa järjestystä, jossa verkkopaketteja aletaan käsittelemään niiden saapuessa puskuriin. Paketteja voidaan käsitellä joko yksinkertaisella FIFO-mallilla saapumisjärjestykseen perustuen tai käsiteltävän paketin valinnassa voidaan käyttää kehittyneempiä menetelmiä, kuten esimerkiksi luvussa 3.1 esitettyjä aktiivisen jononhallinnan vuorontimia. Myös aktiivinen jononhallinta on vuorontamista. Luvuissa 3.3.1-3.3.3 käsitellään vuorontamisen muita käyttötapauksia aktiivisen jononhallinnan lisäksi. Näitä käyttötapauksia ovat muun muassa kapasiteetin hallinta ja rajoittaminen sekä mahdollinen reiluus pakettien käsittelyssä. Rajoittamalla verkkoliikennettä sopivassa kohdassa verkkoa voidaan myös toteuttaa verkon pullonkaulan siirto haluttuun kohtaan verkossa. Yksinkertaistettuna käytössä oleva vuoronnin päättää sen, mikä jonossa oleva paketti lähetetään seuraavaksi verkkoon.

3.3.1 Kapasiteetin hallinta

Verkon kapasiteettia voidaan hallita esimerkiksi jakamalla sitä käyttäjien tai käyttötapauksen perusteella. Kapasiteetilla tarkoitetaan tässä tapauksessa käytössä olevaa kaistanleveyttä tietyllä ajanhetkellä. Kapasiteettia voidaan jakaa rajoittamalla verkon nopeutta perustuen sopiviksi katsottuihin ehtoihin. Rajoittamalla verkkoa sopivilla ehdoilla voidaan esimerkiksi priorisoida haluttua käyttäjäryhmää muiden käyttäjien kustannuksella. Samaa fyysistä tietoliikenneverkkoa voidaan käyttää palvelemaan useita käyttötarkoituksia. Verkossa voi kulkea esimerkiksi useiden operaattoreiden asiakkaiden dataa samaan aikaan ja operaattorilla voi olla liiketoiminnallisia intressejä priorisoida verkkoliikennettä asiakkaidensa kesken.

Yksinkertaisessa esimerkkitapauksessa meillä voi olla GNU/Linux-sovelluspalvelin, jolla on käytössä tietyllä nopeudella toimiva vakaa kiinteä verkkoyhteys Internetiin. Sovelluspalvelimen tuottamaa palvelua käyttävät sekä maksavat että ilmaispalvelun asiakkaat. Sovelluksena toimii tiedostonjakopalvelu, jonka ylläpitäjä on luvannut

maksaville asiakkaille nopeampaa tiedonsiirtoa kuin ilmaiskäyttäjille. Tässä tapauksessa olisi mahdollista hyödyntää vuorontamiseen perustuvaa jononhallintaa palvelimella. Palvelimelle voidaan tehdä määritykset, jotka priorisoivat liikennettä valitsemalla maksavien asiakkaiden yhteyksiin liittyvät tietoliikennepaketit suuremmalla todennäköisyydellä kuin ilmaiskäyttäjien. Tällöin maksavat asiakkaat saavat suhteessa suuremman kaistanleveyden käyttöönsä tässä tiedostonjakopalvelussa. Tämä esimerkki tosin vaatii sen, että asiakkaat voidaan jollain keinoin tunnistaa toisistaan. Eri asiakaskunnille voitaisiin esimerkiksi tehdä omat verkko-osoitteensa sovelluspalvelimelle ja tehdä vuorontaminen tähän tietoon perustuen suoraan verkon tasolla sovelluspalvelimelle. GNU/Linux-palvelimen tapauksessa tarvittavat työkalut löytyvät suoraan käyttöjärjestelmästä (tldp-sla).

Äsken mainittu kapasiteetin rajoittaminen voitaisiin suorittaa GNU/Linux-käyttöjärjestelmässä esimerkiksi käyttäen HFSC-vuoronninta. Tässä yksinkertaisessa esimerkissä meille riittäisi kaksi jonoa: maksavat asiakkaat ja ilmaiskäyttäjät. Koska kyseessä on tiedostonjakopalvelu, jossa käyttäjät voivat ladata tiedostoja palvelimelta, viiveellä ei ole suurta vaikutusta palvelun käytettävyyteen eikä meidän tarvitse ottaa siihen sen enempää kantaa. Rajoitin voidaan toteuttaa esimerkiksi seuraavin komennoin:

```
tc qdisc add dev eth0 root handle 1:0 hfsc default 3
tc class add dev eth0 parent 1:0 classid 1:1 hfsc ls rate 100Mbit ul rate 100Mbit
tc class add dev eth0 parent 1:1 classid 1:2 hfsc ls rate 70Mbit ul rate 100Mbit
tc class add dev eth0 parent 1:1 classid 1:3 hfsc ls rate 30Mbit ul rate 100Mbit
```

Tässä esimerkissä kytkemme HFSC-vuorontimen päälle eth0-verkkokorttiin ja asetamme oletusarvoiseksi jonoksi solmun numero 3. Rajoitamme kaistanleveyden 100 megabittiin sekunnissa juurisolmulla ja luomme lisäksi kaksi lehtisolmua. Tässä esimerkissä voidaan olettaa, että palvelimelta lähtevän liikenteen todellinen suurin mahdollinen kaistanleveys on tämä 100 megabittiä sekunnissa. Käytössä oleva kaistanleveys jakautuu tässä tapauksessa siten, että molemmille lehtisolmuille taataan juuressa olevasta kapasiteetista tietty osuus mutta kumpikin solmu voi lähettää täydellä kapasiteetilla, mikäli toinen solmu ei lähetä kyseisellä hetkellä. Lehtisolmujen yhteenlaskettu kaistanleveys voi olla suurimmillaan niiden juurisolmun verran. Seuraavaksi

meidän täytyy tunnistaa käyttäjät toisistaan. Tässä esimerkissä voimme käyttää aiemmin kuvattua tapaa tunnistaa palvelun käyttäjät palvelimen verkko-osoitteen perusteella. Voimme käyttää esimerkiksi osoitteita 10.0.100.20 ja 10.0.100.21 sovelluspalvelimessa. Tarkoitus on, että ilmaiskäyttäjät ottavat aina yhteyttä ensimmäiseen ja maksavat asiakkaat toiseen näistä kahdesta verkko-osoitteesta. Nyt voimme helposti tunnistaa palvelimella kaikki verkkopaketit, jotka kohdistuvat kuhunkin käyttäjäryhmään ja jakaa paketit aiemmin luotuihin jonoihin. Eräs tapa toteuttaa tämä on käyttää nftables-ohjelmiston mangle-toiminnallisuutta. Kyseinen ohjelmisto on saatavissa nykyaikaisiin GNU/Linux-käyttöjärjestelmiin ja sen tarkoituksena on toimia käyttöjärjestelmän pakettikäsittelyn rajapintana. Normaalin pakettisuodatuksen lisäksi sillä voidaan tehdä myös muita tietoliikennettä käsitteleviä toimintoja, kuten esimerkiksi merkitä paketteja tai ohjata niitä tämän esimerkin tavoin haluttuihin jonoihin. Aiemmin kuvatun esimerkkitapauksen jonoihin ohjaus voidaan toteuttaa seuraavin komennoin nftablesin komentokehotteessa (nftables-wiki, nft):

```
add table ip mangle
add chain ip mangle POSTROUTING { type filter hook output priority 0; }
add rule mangle POSTROUTING ip saddr 10.0.100.20 meta priority set 1:3
add rule mangle POSTROUTING ip saddr 10.0.100.21 meta priority set 1:2
```

Komennot luovat säännöt molemmille käyttötapauksille siten, että kaikki palvelimelta tietystä verkko-osoitteesta lähtevä liikenne ohjautuu aiemmin tehtyihin HFSC-jonoihin käyttötapauksensa mukaisesti. Tällä saavutetaan esimerkissä kuvattu käyttötapaus, jossa verkkopalvelun ilmaiskäyttäjät ja maksavat asiakkaat saavat käytössä olevasta kaistanleveydestä eri osan (tc-hfsc, nftables-wiki, nft). Tämä esimerkki ei tosin ota huomioon käyttäjien yhdistymistä tiettyyn osoitteeseen eikä rajoita sovelluspalvelimen osoitteiden käyttöä millään tavoin eli myös ilmaiskäyttäjät voisivat halutessaan ottaa suoran yhteyden maksavien asiakkaiden käyttöön varattuun osoitteeseen. Tässä esimerkkitapauksessa verkkoon tulisikin oikeassa tuotantokäytössä kehittää menetelmä väärinkäytösten estämiseksi.

3.3.2 Pullonkaulan siirto

Kuten aiemmin jo esitettiin, puskuroinnin aiheuttama lisääntynyt viive aiheutuu verkon pullonkaulassa. Joissain tapauksissa ei riitä, että aikaisemmin esitettyjä AQM-algoritmeja vain kytketään päälle niiden oletusarvoilla. Todellisuudessa emme voi suoraan hallita lähettävää päätä, ellemme tietysti itse ole se. Klassinen esimerkki tällaisesta tapauksesta ovat normaalit kuluttajille suunnatut epäsymmetriset Internet-liittymät. Näiden liittymien tapauksessa asiakas ei tyypillisesti voi vaikuttaa siihen, miten operaattori hallinnoi omilla laitteillaan esiintyviä puskureita. Sopivissa olosuhteissa tälle operaattorin laitteelle pääsee kertymään jonoa, sillä näissä yhteyksissä muodostuu pullonkaula tyypillisesti operaattorin ja asiakkaan välille silloin, kun asiakkaaseen päin kohdistuu raskasta tiedonsiirtoa Internetistä päin. Asiakkaan verkkoon liittämistä laitteista operaattorin suuntaan voi puolestaan aiheutua toinen pullonkaula. Tämä tapahtuu sellaisessa tilanteessa, jossa asiakkaan päästä lähetetään enemmän dataa verkkoon kuin mikä tämän polun kapasiteetti on. Edellä kuvattuun tilanteeseen päästä helposti epäsymmetrisillä yhteysnopeuksilla, jotka poikkeavat verkkokorttien kättelynopeudesta.

Tarkastellaan seuraavaksi esimerkkitapausta verkkoyhteydestä, jossa muodostuu helposti pullonkaula loppukäyttäjän päähän ja siirretään pullonkaula helpommin hallittavaan paikkaan. Tässä esimerkkiyhteydessä käytetään normaalia kaapelimodeemia, johon käyttäjä liittää korkeanopeuksisen verkkokortin omaavan palvelimen, joka kykenee helposti lähettämään verkkokortin kättelynopeudella dataa modeemille mutta modeemi ei kykene lähettämään operaattorin suuntaan samalla nopeudella. Tässä tapauksessa modeemi joutuu puskuroimaan siihen tulevaa dataa ja puskurin täytyttyä paketteja aletaan pudottamaan. Esimerkin vuoksi tätä modeemia ei voida millään tavoin hallita ja käytössä on yksi FIFO-jono. Esimerkin tapauksessa myös operaattorin pään seuraava aktiivilaite toimii vastaavalla tavalla eli käytössä ei ole mitään aktiivista jononhallintaa. Tässä kuvitteellisessa esimerkkitapauksessa asiakkaan ja operaattorin välinen yhteys on kaistanleveydeltään ja viiveeltään vakaa. Yhteyden nimellinen ja toiminnallinen nopeus on 100/10 Mbps.

Yksi alkuperäisen FQ-CoDel-algoritmin kehittäjistä, Dave Täht, esittää Cerowrt-projektissaan (Cerowrt) lähestymistavaksi edellisessä kappaleessa esitettyyn tilanteeseen

keinotekoisen pullonkaulan luomista ja sen hallitsemista. Tämä tarkoittaa käytännössä sitä, että rajoitamme käytössä olevaa kaistanleveyttä ohjelmallisesti jollain muulla pullonkaulaan kytketyllä laitteella hieman todellista kapasiteettia pienemmäksi. Tällöin hallitsematon pullonkaula siirtyy sellaiseen paikkaan, jossa voimme hallita sitä ja soveltaa siihen sopivia menetelmiä. Ajatuksena on siis rajoittaa yhteysnopeutta molempiin suuntiin, jolloin vältämme jonon muodostumisen niihin paikkoihin, joissa emme voi hallita sitä. Tämä tapahtuu käytännössä liittämällä jokin hallittava laite tämän modeemin ja siihen liitettyjen muiden laitteiden väliin ja rajoittamalla tällä hallittavalla laitteella yhteysnopeutta. Rajoitetaan siis nopeus olemaan esimerkiksi 95/9,5 Mbps, jolloin pullonkaula siirtyy sekä asiakkaalle saapuvan liikenteen että asiakkaalta lähtevän liikenteen osalta tälle laitteelle. Nyt voimme soveltaa liikenteen molempiin suuntiin haluamaamme AQM-tekniikkaa pitämään tämän keinotekoisen pullonkaulan jonotusviiveen sopivana.

Huomionarvoista edellisessä kappaleessa kuvatussa menetelmässä tosin on se, että emme oikeasti voi vaikuttaa siihen, kuinka nopeasti Internetistä päin meille saapuu dataa. Keinotekoiseenkin pullonkaulaan voi silti saapua niin paljon dataa kuin sitä edeltävän yhteyden kaistanleveys on. Tämän ratkaisun keskeinen ajatus onkin aiheuttaa ruuhkanhallinnan päälle kytkeytyminen lähettävissä päissä ennen jonon syntymistä alkuperäiseen pullonkaulaan. Tätä yhteyden keinotekoista rajoittamista tulisi tosin käyttää vain silloin, kun emme voi vaikuttaa jonkin fyysisen verkkoyhteyden molempien päiden jonoihin muilla menetelmillä. On hyvä ottaa huomioon, että jotkin operaattorit voivat käyttää AQM-tekniikoita omissa laitteissaan, jolloin jonotuksesta aiheutuvaa viivettä ei välttämättä ole helppo havaita eikä se aiheuta merkittävää käytettävyysongelmaa. Hallitsemattomien puskureiden kanssa ongelma on huomattavasti helpommin havaittavissa.

3.3.3 Reiluus tietoliikenteessä

Reiluudella tietoliikenteessä on tarkoituksena saavuttaa se tilanne, että jokaisella tietovirralla olisi mahdollisuus saada osuus käytössä olevasta kokonaiskapasiteetista. Reiluus on vuorontimen ominaisuus, sillä vuoronnin päättää sen, mikä paketti lähetetään seuraavaksi verkkoon. Keskeiset suunnitteluperiaatteet vuorontimelle ovat ag-

gressiivisten tietovirtojen sopeuttaminen muihin tietovirtoihin siten, että ne eivät häiritse toisiaan. Tämän lisäksi palvelun laatu tulee pysyä korkeana eli verkkoyhteyden käyttöaste voidaan pitää korkeana kärsimättä käytettävyysongelmista (MacGregor ym. 2000). Tässä luvussa tarkastellaan reiluuden yleisiä toimintaperiaatteita.

Tietovirta koostuu paketeista, jotka reititetään lähettäjän ja vastaanottajan välillä käyttäen yhtä tai useampaa reititintä tietyllä polulla verkossa. Näille paketeille tulisi taata tasalaatuinen palvelutaso jokaisella polun reitittimellä. Tietovirrat on mahdollista erottaa toisistaan pakettien otsikkotiedoilla, joten otsikkotiedoista voidaan esimerkiksi laskea hajautusarvo, jonka perusteella tietovirrat voidaan tunnistaa toisistaan. Reiluuden toteuttaminen on ollut aiemmin laskennallisesti raskasta mutta nykyisin laitteiden suorituskyky on kehittynyt sen verran, että tämä ei ole enää ongelma (MacGregor ym. 2000).

Esimerkiksi aiemmin luvussa 3.1.4 esitetty FQ-CoDel käyttää toiminnassaan DRR++ -tyyppistä vuoronninta (RFC 8290), joka pyrkii reiluuteen tietovirtojen kesken. Pelkän reiluuden toteuttamisen lisäksi DRR++ kykenee hallitsemaan hetkittäiset piikit sekä tunnistaa jo reaaliaikaiset tietovirrat raskaiden tiedonsiirtojen joukosta (MacGregor ym. 2000).

4 CASE: SKYNETT LANGAMES 24

Tässä luvussa esitetään vuonna 2018 järjestetyn SKYNETT LANGAMES 24 -verkkopelitapahtuman tarpeita vastaava tietoverkko, jonka rakentamisessa hyödynnettiin aikaisemmin tässä tutkielmassa kuvattuja menetelmiä verkkoviiveen hallitsemiseksi. Luvun tarkoituksena on näyttää käytännön esimerkki aiemmin opituista menetelmistä.

SKYNETT LANGAMES on Parikkalassa Skynett ry:n vuosittain järjestämä verkkopelitapahtuma. Tapahtuman tarkoituksena on kokoontua yhteiseen tilaan pääasiassa pelaamaan videopelejä. Nykyisin videopeleille tyypillistä ovat erilaiset moninpeliominaisuudet, jotka vaativat tietoverkkoyhteyksiä. Tätä tarkoitusta varten tapahtumaa varten on rakennettu oma tietoverkko, josta on myös yhteys Internetiin. Kävijät kytkevät laitteensa paikalla oleviin tietoliikennekytkimiin tai liittyvät langattomaan verkkoon. Verkon käyttötarkoitus on mahdollistaa verkkopelien pelaaminen paikallisesti sekä Internetin käyttö.

Verkkopelien erityisvaatimukset tietoverkolle ovat alhainen viive ja pieni pakettien hävikki. Pelit eivät varsinaisesti käytä suurta määrää kaistanleveyttä mutta vaativat alhaisen viiveen reaaliaikaisen luonteensa vuoksi. Pelien lisäksi tällaisissa tapahtumissa usein käytetään raskasta tiedonsiirtoa erilaisten sovellusten, varsinkin pelien, lataamiseen omille päätelaitteille. Kuten aikaisemmin jo esitettiin, raskas tiedonsiirto yhdessä reaaliaikaisuutta vaativien sovellusten kanssa voi tietyissä tilanteissa aiheuttaa ongelmia.

Tapahtuman Internet-yhteytenä oli paikallisen teleoperaattorin tarjoama symmetrinen 1/1 Gbit/s valokuituyhteys, joka oli liitetty suoraan erilliseen tapahtumaverkon reitittimeen. Tapahtumaverkon reitittimenä toimi normaali GNU/Linux-palvelin, jolla hoidettiin reitityksen lisäksi verkon palomuuri-, QoS-, DHCP-, DNS-, sekä NAT-toiminnot. Palomuurin tarkoituksena oli suodattaa Internetiin päin suuntautuvaa liikennettä estämällä yleisesti haitalliseksi havaitut yhteydet. NAT-toimintoa käytettiin siitä syystä, että operaattorilta ei saatu kokonaista julkista aliverkkoa tapahtuman käyttöön. Syy käyttää geneeristä palvelinta tähän tarkoitukseen oli ratkaisun edullisuus ja ohjelmallisesti toteutettavien toimintojen laajuus. Edellä mainitut ominaisuudet ovat voittoa tavoittelemattomalle yhdistykselle hyvä syy käyttää ohjelmallista ratkaisua tähän

käyttötarkoitukseen. Palvelimeen oli asennettu uusin testausversio Debian GNU/Linux-jakelusta (asennettu 27.12.2018).

Koska tapahtumaverkkoon liitetyt laitteet käyttävät verkkoa eri tavoin ja käytössä olevat sovellukset eivät ole verkon ylläpitäjien tiedossa, on ainoa järkevä vaihtoehto hyödyntää geneerisiä lähestymistapoja verkon tasolla puskuroinnin hallintaan. Emme siis pysty tunnistamaan yksittäisten sovellusten liikennettä riittävällä tarkkuudella muusta liikenteestä. Tämän lisäksi ongelmaksi muodostuu se, että verkkoon liitetyt laitteet eivät ole verkon ylläpitäjien hallinnassa. Tämä johtaa siihen, että laitteisiin ei voida tehdä muutoksia. Edellä mainituista syistä johtuen lähestymistavaksi valittiin verkon tasolla tehtävä segmentointi, kapasiteetin jakaminen eri segmenttien kesken sekä aktiivisen jononhallinnan käyttöönottaminen.

Tapahtumaverkon kävijöille oli jokaiselle varattu gigabitin nopeudella toimiva portti heille lähimmästä pöytäkytkimestä, josta yhteys jatkui kierretyllä parikaapelilla kohti runkoverkkoa. Kytkinten väliset yhteydet toimivat gigabitin nopeudella ja runkokytkimestä oli gigabitin valokuituyhteys edellisessä kappaleessa kuvattuun palvelimeen. Tapahtumapaikalla oli 192 konepaikkaa kävijöille ja tämän lisäksi tapahtuman järjestäjillä oli joitain omia päätelaitteita, kuten esimerkiksi maksupäätteitä, verkossa. Kaikki verkon käyttäjät jakoivat saman fyysisen yhteyden Internetin suuntaan ja suurin osa kaikesta liikenteestä oli Internet-liikennettä. Sisäverkossa tapahtui myös jotain liikennöintiä mutta se oli huomattavasti vähäisempää kuin Internet-liikenne. Tässä tapauksessa Internet-yhteys on selkeä pullonkaula, jossa mahdollinen ruuhkautuminen esiintyisi. Tässä käyttötapauksessa ruuhkautuminen aiheuttaisi käytettävyysongelmia käytettävien sovellusten kanssa. Suurimmat ongelmat muodostuisivat reaaliaikaisuutta vaativien verkkopelien toimivuuden heikkenemisestä.

Varsinaisen pelaamisen lisäksi tapahtumassa lähetettiin reaaliaikaista videokuvaa Internetin suoratoistopalveluun, joka oli tässä tapauksessa Twitch.tv. Kyseinen palvelu ottaa vastaan RTMP-protokollalla lähetettävää reaaliaikaista videokuvaa. RTMP-protokolla lähettää varsinaisen datansa TCP/IP-paketteina vaikka kyseessä onkin reaaliaikainen sovellus. Jotta tämä videokuva siirtyisi sulavasti Internetiin, oli tätä käyttöä varten varattu järjestäjille vuorontamisen keinoin osuus käytössä olevasta kaistanle-

veydestä. Tämä tarkoittaa käytännössä sitä, että järjestäjille oli taattu tietty kaistanleveys käytettäväksi kaikissa olosuhteissa. Kapasiteetti jaettiin liitteessä 1 esitetyn skriptin keinoin kolmeen luokkaan käyttäen GNU/Linuxin HFSC-toteutusta. Nämä luokat muodostavat loogisia kokonaisuuksia, joihin voidaan tarvittaessa soveltaa esimerkiksi haluttuja palomuurisääntöjä. Loogiset verkkosegmentit olivat erotettuina toisistaan verkon L2-tasolla käyttäen tietoliikennelaitteiden VLAN-toiminnallisuutta. Reitittimenä toimiva palvelin oli liitetty kaikkiin verkossa oleviin VLAN-segmentteihin. Luokittelu toimii tässä verkossa käytetyn HFSC-vuorontajan vuoksi siten, että jokaiselle luokalle varataan tietty nopeus, joka taataan kaikissa tilanteissa kyseiselle luokalle. Tämän lisäksi määritellään suurin mahdollinen nopeus, jolla luokka voi lähettää verkkoon, mikäli muut luokat eivät sillä hetkellä käytä niille määritettyä pienintä taattua nopeutta. Taulukossa 2 esitetään nämä aiemmin mainitut kolme kaistanleveyden hallinnan kannalta oleellista luokkaa sekä niille varatut kaistanleveydet. Edellä kuvattujen kolmen luokan lisäksi taulukossa esitetään kokonaiskapasiteettia kuvaava juurisolmu.

Taulukko 2. Skynett 24 -verkkopelitapahtuman HFSC-luokat.

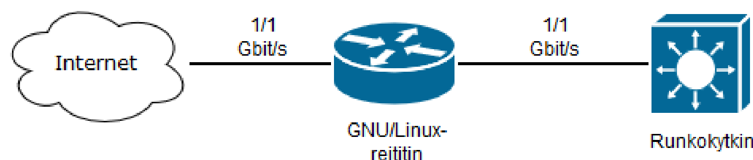
Luokka:	**Selite:**	**Pienin nopeus:**	**Suurin nopeus:**
1:1	Runkoyhteys	920 Mbit/s	920 Mbit/s
1:2	Pöytäpaikat	700 Mbit/s	920 Mbit/s
1:3	Järjestäjät	150 Mbit/s	920 Mbit/s
1:4	Langaton verkko	70 Mbit/s	920 Mbit/s

Kuten taulukosta 2 voidaan havaita, on juurisolmun kapasiteetti 920 Mbit/s. Tämä johtuu siitä, että halutaan välttää ylimääräinen puskurointi Internet-operaattorin päässä, jossa tätä prosessia ei voida hallita. Koska tämä sama reititin reitittää myös sisäverkon liikenteen, pätevät samat luokat myös sisäiseen liikenteen. Tässä tapauksessa luodaan verkkoon luvussa 3.3.2 kuvatulla menetelmällä uusi keinotekoinen pullonkaula symmetrisen yhteyden molemmin puolin. Tähän uuteen pullonkaulaan voidaan nyt soveltaa kehittyneitä AQM-tekniikoita hallitsemaan jokaiseen luokkaan muodostuvaa jonoa

siten, että loppukäyttäjillä olisi mahdollisimman hyvin toimiva verkkoyhteys tapahtuman käyttötapaukset huomioiden.

HFSC on puumainen tietorakenne, jossa lehtisolmut määrittävät luokan ominaisuudet mutta niitä voidaan periä solmujen vanhemmilta. Tässä tapauksessa luokat 1:2, 1:3 ja 1:4 ovat näitä lehtisolmuja, joiden mukaan kapasiteettia jaetaan ja luokka 1:1 on näiden solmujen vanhempi, joka jakaa kapasiteettinsa lastensa kesken. HFSC-vuorontimella kaistanleveyden lisäksi jokaisella lehtisolmulla voi olla myös viiveluokitus määriteltynä (tc-hfsc). Käytämme tapahtumaverkossa kuitenkin pelkkää kaistanleveyden hallintaa HFSC:n linkshare-toiminnallisuudella ja jätämme viiveen hallinnan FQ-CoDel-algoritmille. AQM-tekniikaksi päätettiin valita FQ-CoDel, sillä se soveltuu teoriassa hyvin käytettäväksi tällaiseen verkkoon, jossa on paljon erityyppistä liikennettä useilta eri laitteilta ja viiveellä on merkitystä sovellusten käytettävyyteen.

FQ-CoDel-algoritmia hyödynnetään siten, että luodaan sekä verkkoon saapuvalle (eng. ingress) että siitä lähtevälle liikenteelle (eng. egress) ylimääräiset tc-luokat taulukossa 2 esitetyn luokitteluperiaatteen mukaisesti ja liitetään kuhunkin näistä juuri luoduista luokista FQ-CoDel-vuoronnin liitteessä 1 esitetyllä skriptillä. Koska käytössä on symmetrinen Internet-yhteys ja reitittimessä on kaksi verkkokorttia, voidaan käyttää sekä ingress- että egress-liikenteeseen soveltuvia AQM-tekniikoita. Käytännössä tämä tapahtuu siten, että toinen verkkokorteista on liitetty Internetiin ja toinen tapahtuman sisäverkon runkokytkimelle. Kaikki Internet-liikenne siis kulkee molempien verkkokorttien läpi siten, että toiseen verkkokorttiin saapuva liikenne lähtee aina ulospäin toisesta. Edellä kuvatulla kytkentätavalla kaikkeen Internet-liikenteeseen voidaan soveltaa HFSC- ja FQ-CoDel-tekniikoita. Esitetään seuraavaksi kuvan 16 avulla karkeasti tämä kytkentämalli, jolla verkkoyhteys oli toteutettu.

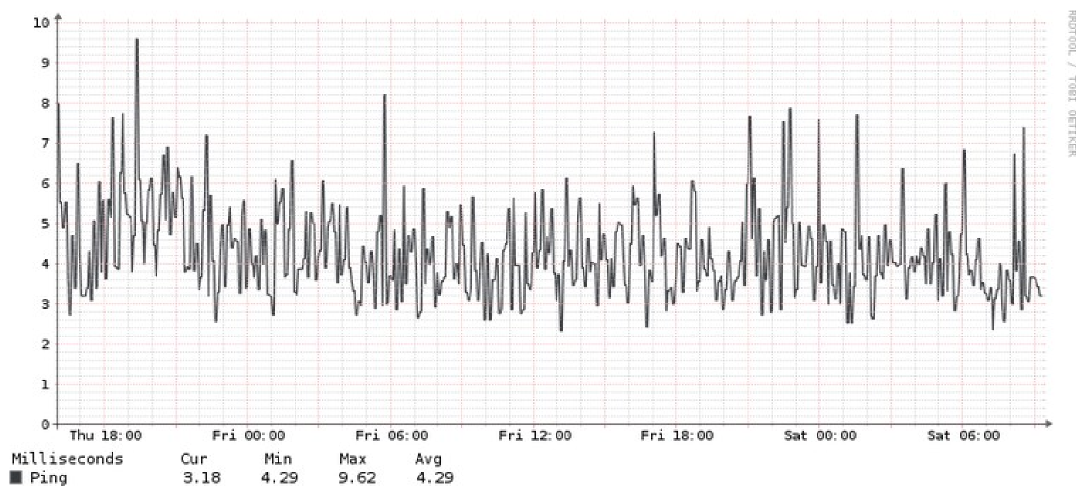


Kuva 16. Skynett-tapahtuman Internet-reitittimen kytkentä.

Kuten kuvasta 16 voimme havaita ja kuten aiemmin tässä luvussa mainittiin, oli operaattorilta tuleva Internet-yhteys kytketty suoraan tapahtumaverkon reitittimen toiseen verkkokorttiin. Tapahtumaverkon runkokytkimestä puolestaan oli suora kytkentä toiseen verkkokorteista.

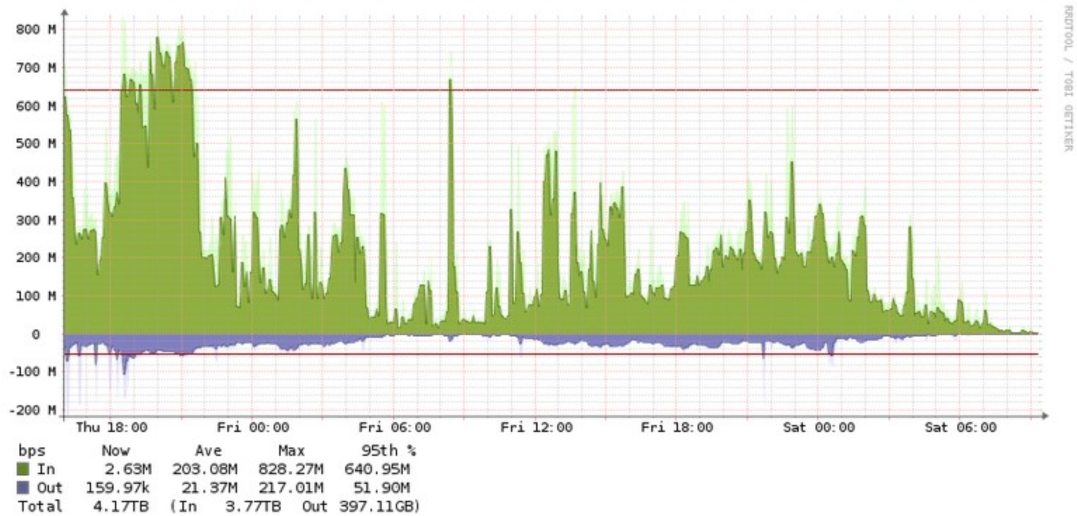
Verkon kaistanleveyttä ja keskimääräistä viivettä operaattorin pään reitittimeen mitattiin erillisellä palvelimella, johon oli asennettu LibreNMS-ohjelmisto. Ohjelmisto oli määritetty siten, että se mittasi tapahtumaverkon reitittimen Internetiin päin olevan verkkokortin käyttämää kaistanleveyttä sekä normaalia ping-viivettä operaattorin reitittimeen. Näiden edellä mainittujen lisäksi jokaisessa tapahtumaverkon kytkimessä sekä Internet-reitittimessä oli SNMP-palvelu päällä, jonka avulla näistä laitteista saatiin muutakin tietoa kerättyä. Verkon kannalta kiinnostavimmat tiedot tosin olivat viive sekä kaistanleveys tietyllä ajanhetkellä.

LibreNMS-ohjelmistossa on oletuksena 5 minuutin viive mittauksissa. Tätä arvoa ei muutettu tapahtuman aikana, joten ohjelmiston tuottamat mittaustulokset eivät pidä sisällään yksittäisiä ja nopeita piikkejä. Tarkastellaan seuraavaksi ohjelmistolla tehtyjä mittauksia verkon toiminnasta kuvien 17 ja 18 avulla.



Kuva 17. Ping-mittaus LibreNMS-ohjelmistolla.

Kuten kuvasta 17 voidaan havaita, asettuu keskimääräinen viive hyvin lähelle 4 millisekuntia eikä Bufferbloat-ilmiölle tyypillistä suurta viiveen nousua tai vaihtelua havaittu.



Kuva 18. Kaistanleveyden mittaus LibreNMS-ohjelmistolla.

Keskimääräinen kaistanleveys tällä oletusarvoisella 5 minuutin mittausvälillä oli noin 209 Mbps sisään- ja 21 Mbps ulospäin, kuten kuvasta 18 voidaan havaita. Keskiarvojen perusteella tämä verkko ei olisi muutenkaan erityisen kovalla kuormalla, joten vielä olisi mahdollista lisätä konepaikkojen määrää vastaavalla yhteydellä ilman suurempia käytettävyysongelmia.

LibreNMS-mittausten lisäksi reitittimen suorittimen käyttöä, verkkoliikennettä sekä ping-viivettä operaattorin reitittimelle seurattiin reitittimeltä suoraan sekunnin välein käyttäen SSH-yhteyttä reitittimelle ja ajamalla siinä sopivia hallintatyökaluja. Näitä mittaustuloksia ei kuitenkaan tallennettu mihinkään vaan niitä käytettiin vain yleisen toimivuuden seuraamiseen reaaliajassa. Sekunnin välein mittauksissa näkyi selvästi se, että verkkoyhteyden käytössä oli varsinkin kaistanleveyden osalta suuria muutaman sekunnin mittaisia piikkejä toistuvasti. Näiden piikkien aikana mitattu verkkoviive ei kuitenkaan noussut merkittävästi, mikä olikin koko suunnitellun ratkaisun keskeinen tavoite.

Tapahtuman aikana opittiin se, että geneeristä palvelinlaitteistoa ja GNU/Linuxia käyttäen on mahdollista rakentaa hyvin toimiva reititin kevyeen käyttöön. Kevyenä käyttönä voidaan pitää tässä esimerkissä kuvattua 1/1 Gbit/s yhteyttä. Vastaaviin käyttötapauksiin on olemassa myös kokonaisia avoimen lähdekoodin käyttöjärjestelmiä sekä kaupallisia laitteita, joilla voidaan toteuttaa vastaava ratkaisu. Syy käyttää omaa rat-

kaisua valmiin ratkaisun sijaan oli osittain oppia GNU/Linuxin reititys- ja verkkotoiminnallisuuksia sekä testata näiden toimintaa heterogeenisessa ympäristössä aidolla datalla. Tapahtumassa havaittiin myös se, että siellä pelatut verkkopelit viestivät pitkälti käyttäen UDP-protokollaa ja paketit olivat pieniä. Tällaiseen käyttötarkoitukseen aiemmin esitetty FQ-CoDel on teoriassa erittäin hyvä vaihtoehto ja se havaittiin myös käytännössä toimivaksi tapahtuman aikana. Myös muun tyyppisessä käyttötapauksessa voitaisiin soveltaa vastaavaa ratkaisua siirtämällä pullonkaula tarvittaessa hallittavaan sijaintiin sekä käyttämällä AQM-tekniikkaa, joka pyrkii hallitsemaan viivettä ja toimii reilusti tietovirtojen kesken. Yhteenvetona tapahtuman aikana verkkoyhteys toimi hyvin sille asetettujen vaatimusten mukaisesti. Jälkeenpäin ajateltuna tapahtumassa käytettiin jopa liian tiukkoja palomuurisääntöjä etenkin ICMP-liikenteen kohdalla, mutta tästä on hyvä ottaa oppia tulevia verkkoja silmällä pitäen. Käytetyt palomuurisäännöt, joilla muun muassa verkkoliikenteen priorisointi käytännössä toteutettiin, olivat liitteen 2 mukaiset. Tapahtuman jälkeen asetuksia on jatkokehitetty huomattavasti seuraavaa tapahtumaa varten. Myös käytetyt laitteet ovat vaihtuneet huomattavasti suorituskykyisempiin malleihin.

5 POHDINTA JA JATKOTUTKIMUSIDEAT

Internet-yhteyksissä pääsääntöisesti käytetty tiedonsiirtoprotokolla on TCP/IP, joka kehitettiin Internetin alkuaikoina. Kyseisinä aikoina yhteydet olivat nykyisillä standardeilla hitaita eikä verkon laitteissa ollut paljoa muistia, joten myös puskurit näissä laitteissa olivat pieniä. Lisäksi verkon laitteissa käytettiin tyypillisesti FIFO-tyyppistä jonotusta. Koska pakettikytkentäisessä tietoverkossa lähettäjä määrittää itse oman lähetysnopeutensa, ajauduttiin tilanteisiin, joissa vastaanottajan verkkoyhteys ylikuormittui ja paketteja katosi matkalle, sillä puskurit olivat pieniä eivätkä voineet pitää sisällään suurta määrää paketteja kerralla, mikä johti siihen, että yhteyden ruuhkautuminen voitiin tuolloin havaita helposti pakettien hävikistä. Tämän vuoksi protokollaan kehitettiin ruuhkanhallintaominaisuus, joka pienentää lähetysnopeutta havaitessaan pakettien häviämistä. Nykyisin kuitenkin yhteysnopeudet ja muistin määrä laitteissa on kasvanut huomattavasti. Lisäksi myös käyttötapaukset ovat muuttuneet siten, että nykyisin käytetään paljon reaaliaikaisuutta vaativia sovelluksia, kuten esimerkiksi verkkopelejä, videokonferensseja ja VoIP-sovelluksia. Kun nämä reaaliaikaiset käyttötapaukset, pakettien hävikkiin perustuva ruuhkanhallinta, suuret puskurit ja FIFO-tyyppinen jonotus verkon laitteissa yhdistetään, saadaan aikaan käytettävyysongelma. Tämä käytettävyysongelma johtuu siitä, että ennen kuin ruuhkanhallinta ehtii kytkeytyä päälle, on puskureihin kertynyt jo huomattava määrä dataa. Puskureissa oleva data joudutaan lähettämään FIFO-jonoissa kokonaan verkkoon ennen kuin uudet jonoon saapuvat paketit saavat oman lähetysvuoronsa. Tämä johtaa siihen, että jonoissa pitkään roikkuvat paketit nostavat verkkoviivettä ja sovelluksen mukaan viive voi aiheuttaa käytettävyysongelman. Käytettävyysongelmia näissä sovelluksissa ovat muun muassa videon ja äänen häiriöt ja pätkiminen. Verkkopelien kohdalla käytettävyysongelmat riippuvat pelistä mutta näitä ongelmia ovat muun muassa osumatarkkuuden heikentyminen ja liikkeiden rekisteröimättä jääminen.

Ongelmiin on kehitetty erilaisia ratkaisuja, joista merkittävimpiä ovat AQM-tekniikat sekä siirtoprotokollien kehittyneet ruuhkanhallintamenetelmät. AQM-tekniikoiden tarkoituksena on hallita puskuria FIFO-mallia älykkäämmin siten, että jonosta voidaan tarvittaessa pudottaa paketteja, mikäli se on tarpeellista viiveen hallitsemisen kannalta. Tämän lisäksi kehittyneet AQM-tekniikat, kuten esimerkiksi FQ-CoDel, hyödyntävät

reiluutta toiminnassaan, joten yksittäiset tietovirrat eivät pääse dominoimaan verkkoa vaan jokaisella tietovirralla on mahdollisuus päästä lähettämään dataa verkkoon. Kehittyneemmät protokollien ruuhkanhallintaominaisuudet puolestaan havaitsevat verkon todellisen ruuhkautumisen mittaamalla pakettien hävikin sijaan verkkoviivettä ja todellista toteutunutta siirtonopeutta.

Yleisesti katsottuna molemmilla lähestymistavoilla päästään parempaan lopputulokseen kuin perinteisillä menetelmillä, mutta menetelmillä on selkeästi omat paikkansa verkoissa. Ruuhkanhallinta toteutuu käytännössä aina lähettäjän päässä, kun taas AQM-tekniikat soveltuvat paremmin käytettäväksi verkon reitittimillä ja muilla laitteilla. Liiallisen puskuroinnin aiheuttamaan lisääntyneeseen verkkoviiveeseen voidaankin vaikuttaa useilla eri keinoilla.

Tiivistettynä tietoverkossa kannattaa käyttää todellisen ruuhkautumisen havaitsevaa ruuhkanhallintaa silloin, kun se on mahdollista sekä käyttää aktiivista jononhallintaa tietoliikennelaitteilla hallitsemaan niiden puskureita. Esimerkiksi TCP/IP-protokollaa käyttäessä hyvin toimiva ruuhkanhallintatekniikka on BBR, joka voidaan kytkeä suoraan käyttöjärjestelmän puolelta päälle ilman muita muutoksia käytössä olevaan järjestelmään tai vastaanottajan päähän. Tietoliikennelaitteilla puolestaan on suotavaa käyttää aktiivista jononhallintaa perinteisen FIFO-tyyppisen jonotuksen sijaan. Hyviksi havaittuja AQM-tekniikoita tietoliikennelaitteilla käytettäviksi ovat muun muassa RED ja PIE, sillä ne pystyvät hallitsemaan jonotusviivettä tehokkaasti ja niille löytyy toimiva toteutus laitteistolta suoraan. Myös muita menetelmiä on mahdollista käyttää, mutta niille ei ole välttämättä tukea verkkolaitteilla itsellään, joten niiden vaatima laskenta voidaan joutua suorittamaan käyttäen esimerkiksi geneeristä palvelinta tai verkkolaitteen suoritinta varsinaisten ASIC-piirien sijaan. Mikäli verkkoliikennettä reititetään esimerkiksi geneerisellä GNU/Linux-palvelimella, voidaan AQM-tekniikkana käyttää muun muassa FQ-CoDel:ia, joka yhdistää viiveen hallinnan ja reiluuden tietovirtojen kesken.

Bufferbloat-ilmiö pohjautuu pitkälti huonoihin valintoihin Internetin alkuajoilta saakka eikä ongelmaa ole vieläkään saatu täysin korjattua. Toisaalta nämä valinnat olivat parhaita siihen aikaan tunnettuja ja Internetistä on myöskin tullut jotain aivan

muuta kuin mitä kyseisinä aikoina odotettiin. Pakettien hävikkiin perustuva ruuhkanhallinta TCP/IP-protokollassa on edelleen käytetyin ruuhkanhallintamenetelmä. Tämän lisäksi monet tietoliikennelaitteet käyttävät edelleen normaalia FIFO-jonotusta puskureissaan ja puskureiden koot ovat suuria, jotta kaistanleveys kyettäisiin pitämään mahdollisimman korkeana jatkuvasti. Internetiä käyttävät nykyaikaiset sovellukset ovat yksi syy siihen, että tämä muinainen ongelma on noussut jälleen pinnalle ja siihen on alettu kehittämään uusia ratkaisuja. Nykyisin moni sovellus tarvitsee toimiakseen vakaan ja reaaliaikaisen yhteyden. Jotta tähän tilanteeseen päästään, tulee verkon tasolla pitää huoli siitä, että verkossa ei tapahdu kohtuutonta määrää puskurointia eli näitä puskureita tulisi hallita jollain tavalla. Tämä tilanne saadaan toteutettua joko kuljetusprotokollan tasolla käyttämällä vaihtoehtoisia ruuhkanhallintamenetelmiä tai verkon tasolla käyttämällä AQM-tekniikoita. Kuljetuskerroksen tasolla toimiva vaihtoehtoinen tapa pyrkii tunnistamaan verkon ruuhkautumisen viiveen ja kaistanleveyden vaihteluna perinteisen pakettien hävikin sijaan. Toinen hyvin toimiva vaihtoehto on käyttää AQM-tekniikoita, jotka käsittelevät puskureihin saapuvaa tietoliikennettä älykkäämmin kuin perinteiset FIFO-jonot ja tarpeen tullen voivat jättää joitain paketteja kokonaan käsittelemättä, jotta puskurin koko ei pääse kasvamaan liian suureksi. Käytännössä tämä pakettien käsittelemättä jättäminen aiheuttaa varsinkin TCP/IP-protokollan tapauksessa ruuhkanhallinnan kytkeytymisen päälle ennen kuin puskuri on liian täynnä. Tämä puolestaan aiheuttaa sen, että tällä keinotekoisella pakettien hävikillä voimme vaikuttaa verkkoviiveeseen hyvinkin tehokkaasti pitäen varsinaisen protokollan toteutuksen muuttumattomana. Yhdistämällä reiluuden tehokkaaseen jonotusviiveen hallintaan, saamme rakennettua tietoverkon, jossa yhdistyy matala viive, korkea kaistanleveys ja hyvä käyttöaste. Lisäksi tätä verkkoa voidaan soveltaa useille kuljetusprotokollille puuttumatta kuljetusprotokollan itsensä toteutukseen. Voidaankin siis sanoa, että AQM-tekniikat ovat geneerinen lähestymistapa Bufferbloat-ongelmaan.

Vaikka Bufferbloat-ongelma on ollut tiedossa jo vuosikymmenten ajan, on merkillistä, että ongelmaan ei ole aikaisemmin kehitetty riittävän hyviä menetelmiä, joita olisi otettu laajemmin käyttöön. Tähän voi vaikuttaa osittain se, että Internetin alkuaikoina käyttötapaukset olivat yksinkertaisempia kuin nykyään. Lisäksi varsinkin kuljetusprotokolliin tehtävät muutokset ovat hankalia ottaa käyttöön, mikäli ne eivät ole taaksepäin yhteensopiva vanhempaa versiota käyttävien laitteiden kanssa. Nykyään TCP/IP-

protokollaan on saatavilla useita ruuhkanhallintamenetelmiä, joista osa hyödyntää perinteistä pakettien hävikkiin perustuvaa lähetysnopeuden hidastamista ja osa puolestaan mittaa viivettä, todellista siirtonopeutta tai näiden yhdistelmää. Protokollassa on mahdollista tehdä muutos pelkästään lähettäjän päässä, joten näiden uusien ja paremmin toimivien menetelmien käyttöönotto on mahdollista ja voidaan toteuttaa asteittain säilyttäen yhteensopivuus olemassa olevien järjestelmien kanssa.

Sekä AQM- että ruuhkanhallintamenetelmät kehittyvät jatkuvasti ja uusia menetelmiä on alettu ottamaan laajemmin ja nopeammin käyttöön kuin Internetin alkuaikoina. Esimerkiksi tässä tutkielmassa aiemmin mainittu FQ-CoDel-tekniikka on useissa nykyisissä GNU/Linux-jakeluissa oletusarvoisesti päällä. Tämä muutos tapahtui vain muutamia vuosia FQ-CoDel:n julkaisusta. Menetelmä ei ole täydellinen mutta se on toistaiseksi yksi parhaista mahdollisista AQM-tekniikoista, joita on mahdollista käyttää näissä GNU/Linux-jakeluissa ennen seuraavan korvaavan tekniikan saapumista. On vain ajan kysymys, kunnes sama tapahtuu kehittyneille TCP/IP:n ruuhkanhallintamenetelmille – ehkä vielä jonain päivänä suurin osa TCP/IP:tä käyttävistä laitteista hyödyntää muuhun kuin hävikkiin perustuvaa ruuhkanhallintaa tai kenties koko protokolla on korvattu jollain muulla. Tämä ei toisaalta poista tarvetta hallita tietoliikennelaitteiden puskurointia, joten AQM-tekniikoiden kehitys tulee luultavasti jatkumaan vielä pitkään.

Käytettävyyden ja paremman suorituskyvyn vuoksi olisi toivottavaa kehittää aktiivilaitteiden ASIC-piireillä toimiva nykyaikainen ratkaisu, joka tunnistaisi luotettavasti verkkoliikenteen todellisen aikakriittisyyden, toimisi reilusti sekä pitäisi jonotusviiveen hallinnassa. Tämän työn kirjoittamishetkellä nousi pinnalle eräs mielenkiintoinen ratkaisu, CAKE, joka ei tosin vielä kirjoittamishetkellä ollut kovin tunnettu eikä siitä löytynyt kattavaa tieteellistä tutkimusta. CAKE löytyy nykyään jo käyttämästäni Debian GNU/Linux-jakelusta ja siinä on paranneltu monia FQ-CoDelin heikkouksia, kuten hajautusarvojen mahdollista törmäystä sekä liikenteen luokittelua käyttäen muun muassa DiffServ-arvoja IP-pakettien otsikkotiedoista. Tämänkaltainen ratkaisu olisi varsin lupaava, mikäli se voitaisiin toteuttaa geneeristen palvelimien ja työasemien ohella myös verkon aktiivilaitteilla.

Viitteet

Alfredsson ym. "Impact of TCP congestion control on bufferbloat in cellular networks." 2013 IEEE 14th International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM). IEEE, 2013

Baker Fred ja Gorry Fairhurst. “IETF recommendations regarding active queue management.” RFC 7567, 2015

Braden Bob ym. “Recommendations on queue management and congestion avoidance in the Internet.” RFC 2309, 1998

Cardozo ym. "Bufferbloat systematic analysis." 2014 International Telecommunications Symposium (ITS). IEEE, 2014.

Cardwell Neal ym. “BBR: Congestion-based congestion control.” Queue 14.5 (2016): 50

Cardwell Neal ym. “BBR congestion control.” Working Draft, IETF Secretariat, Internet-Draft draft-card-well-iccrg-bbr-congestion-control-00, 2017

Cerowrt. “Beating bufferbloat with home routers.” http://blog.cerowrt.org/post/bufferbloat_vs_quality/, viitattu 24.9.2018

Chirichella Chiara ja Dario Rossi. "To the Moon and back: are Internet bufferbloat delays really that large?." 2013 Proceedings IEEE INFOCOM. IEEE, 2013.

Floyd Sally ja Van Jacobson. ”Random early detection gateways for congestion avoidance.” IEEE/ACM Transactions on networking 1.4 (1993): 397-413

Grigorescu Eduard. Reducing internet latency for thin-stream applications over reliable transport with active queue management. Väitöskirja. University of Aberdeen, 2018

Hoeiland-Joergensen Toke ym. “The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm.” RFC 8290, 2018

Jacobson Van. "Congestion avoidance and control." ACM SIGCOMM computer communication review. Vol. 18. No. 4. ACM, 1988

Jim Gettys ja Kathleen Nichols. ”Bufferbloat: Dark buffers in the internet.” Queue, 9(11):40, 2011

Juniper. “Managing Congestion Using RED Drop Profiles and Packet Loss Priorities” https://www.juniper.net/documentation/en_US/junos/topics/concept/red-drop-profile-overview-cos-config-guide.html, viitattu 12.6.2019

MacGregor ym. "Deficits for bursty latency-critical flows: DRR++." Proceedings IEEE International Conference on Networks 2000 (ICON 2000). Networking Trends and Challenges in the New Millennium. IEEE, 2000

Nftables-wiki, https://wiki.nftables.org/wiki-nftables/index.php/Main_Page, viitattu 23.6.2019

Nft, https://manpages.debian.org/testing/nftables/nftables.8.en.html, viitattu 12.9.2019

Nichols Kathleen ja Van Jacobson. "Controlling queue delay." Communications of the ACM 55.7 (2012): 42-50

Pan ym. "PIE: A lightweight control scheme to address the bufferbloat problem." High Performance Switching and Routing (HPSR), 2013 IEEE 14th International Conference on. IEEE, 2013

Postel Jon. “User datagram protocol.” RFC 768, 1980

Raghuvanshi ym. "On the effectiveness of CoDel for active queue management." Advanced Computing and Communication Technologies (ACCT), 2013 Third International Conference on. IEEE, 2013

Ramakrishnan ym. “The addition of explicit congestion notification (ECN) to IP.” RFC 3168, 2001

Rene Adams. “Active queue management: a survey.” Communications Surveys & Tutorials, IEEE, 15(3):1425–1476, 2013

Rossi ym. "LEDBAT: the new BitTorrent congestion control protocol." Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on. IEEE, 2010

sch_fq_codel, https://github.com/torvalds/linux/blob/master/net/sched/sch_fq_codel.c, viitattu 9.9.2019

technet, https://blogs.technet.microsoft.com/networking/2018/07/25/ledbat/, viitattu 24.9.2018

tc-codel, https://manpages.debian.org/buster/iproute2/tc-codel.8.en.html, viitattu 12.9.2019

tc-hfsc, https://manpages.debian.org/buster/iproute2/tc-hfsc.8.en.html, viitattu 12.9.2019

tc-red, https://manpages.debian.org/buster/iproute2/tc-red.8.en.html, viitattu 12.9.2019

tldp-sla, https://www.tldp.org/HOWTO/Adv-Routing-HOWTO/lartc.cookbook.sla.html, viitattu 27.1.2019

White ym. "Active queue management in DOCSIS 3. X cable modems." CableLabs Technical Report (2014)

LIITE 1: QDISC-MÄÄRITYKSET

```
#!/bin/bash

### Muuttujien määrittely ###
wan="enp5s0f0"
lan="enp5s0f1"
up=920
down=920
tc=/sbin/tc
egress="fq_codel noecn flows 8192 target 5ms interval 30ms"
ingress="fq_codel noecn flows 8192 target 5ms interval 30ms"

### Poistetaan mahdollisesti olemassa olevat qdiscit ###
tc qdisc del dev $wan root
tc qdisc del dev $lan root

### INGRESS HFSC-luokat ###
$tc qdisc add dev ${lan} root handle 1:0 hfsc default 2
$tc class add dev ${lan} parent 1:0 classid 1:1 hfsc ls rate ${down}Mbit ul rate ${down}Mbit
$tc class add dev ${lan} parent 1:1 classid 1:2 hfsc ls rate 700Mbit ul rate ${down}Mbit
$tc class add dev ${lan} parent 1:1 classid 1:3 hfsc ls rate 150Mbit ul rate ${down}Mbit
$tc class add dev ${lan} parent 1:1 classid 1:4 hfsc ls rate 70Mbit ul rate ${down}Mbit

### INGRESS FQ-CoDel aiemmin luotuihin luokkiin ###
$tc qdisc add dev ${lan} parent 1:2 ${ingress}
$tc qdisc add dev ${lan} parent 1:3 ${ingress}
$tc qdisc add dev ${lan} parent 1:4 ${ingress}

### EGRESS HFSC-luokat ###
$tc qdisc add dev ${wan} root handle 1:0 hfsc default 2
$tc class add dev ${wan} parent 1:0 classid 1:1 hfsc ls rate ${up}Mbit ul rate ${up}Mbit
$tc class add dev ${wan} parent 1:1 classid 1:2 hfsc ls rate 700Mbit ul rate ${up}Mbit
$tc class add dev ${wan} parent 1:1 classid 1:3 hfsc ls rate 150Mbit ul rate ${up}Mbit
$tc class add dev ${wan} parent 1:1 classid 1:4 hfsc ls rate 70Mbit ul rate ${up}Mbit

### EGRESS FQ-CoDel aiemmin luotuihin luokkiin ###
$tc qdisc add dev ${wan} parent 1:2 ${egress}
$tc qdisc add dev ${wan} parent 1:3 ${egress}
$tc qdisc add dev ${wan} parent 1:4 ${egress}
```

LIITE 2: NFTABLES-MÄÄRITYKSET

```
#!/usr/sbin/nft -f
flush ruleset

define lan-if = enp5s0f1
define wan-if = enp5s0f0

################################################################################

## Taulut ja chainit
add table ip filter
add chain ip filter INPUT { type filter hook input priority 0; }
add chain ip filter FORWARD { type filter hook forward priority 0; }
add chain ip filter OUTPUT { type filter hook output priority 0; }

## Sisäverkon VLANit
add set ip filter lan_vlans { type iface_index; }
add element ip filter lan_vlans {enp5s0f1.10}
add element ip filter lan_vlans {enp5s0f1.40}
add element ip filter lan_vlans {enp5s0f1.50}
add element ip filter lan_vlans {enp5s0f1.60}
add element ip filter lan_vlans {enp5s0f1.100}
add element ip filter lan_vlans {enp5s0f1.200}

## Sallitut ICMP-tyypit
add set ip filter sallitut-icmp { type icmp_type; flags interval; }
add element ip filter sallitut-icmp {0,1,6,8,11-14}

## Estetyt osoitteet ja portit
add set filter drop-wan-udp { type inet_service; flags interval; }
add set filter drop-wan-tcp { type inet_service; flags interval; }
add set filter drop-wan-dst-ip { type ipv4_addr; flags interval; }
add element ip filter drop-wan-tcp {0,25,135-139,179,445,593,1433-1434,7547}
add element ip filter drop-wan-udp {25,135-139,161,445,593,1433-1434,1900}
add element ip filter drop-wan-dst-ip {10.0.0.0/8, 100.64.0.0/10, 169.254.0.0/16, 172.16.0.0/12, 192.0.0.0/24, 192.0.2.0/24,
192.168.0.0/16, 198.51.100.0/24, 203.0.113.0/24, 224.0.0.0/4, 240.0.0.0/4}

################################################################################

## Vmapit
# Conntrack
add map ip filter ct_map { type ct_state : verdict; }
add element ip filter ct_map { established : accept }
add element ip filter ct_map { related : accept }
add element ip filter ct_map { invalid : drop }

### Input filter
add rule ip filter INPUT iif lo counter accept
add rule ip filter INPUT ct state vmap @ct_map
add rule ip filter INPUT icmp type @sallitut-icmp counter accept
add rule ip filter INPUT iif @lan_vlans udp dport {53,67} counter accept
add rule ip filter INPUT iif enp5s0f1.100 udp dport 161 counter accept
add rule ip filter INPUT iif @lan_vlans tcp dport {22,3000} counter accept
add rule ip filter INPUT counter drop

### OUTPUT FILTER [ Tältä hostilta muualle menevä liikenne ]
add rule ip filter OUTPUT oif $wan-if tcp dport @drop-wan-tcp log prefix "OUTPUT TCP ESTO1: " counter drop
add rule ip filter OUTPUT oif $wan-if udp dport @drop-wan-udp log prefix "OUTPUT UDP ESTO1: " counter drop
add rule ip filter OUTPUT oif $wan-if ip daddr @drop-wan-dst-ip log prefix "OUTPUT SRC ESTO: " counter drop

### FORWARD filter
add rule ip filter FORWARD ct state vmap @ct_map
add rule ip filter FORWARD oif $wan-if tcp dport @drop-wan-tcp log prefix "FORWARD TCP ESTO1: " counter drop
add rule ip filter FORWARD oif $wan-if udp dport @drop-wan-udp log prefix "FORWARD UDP ESTO1: " counter drop
add rule ip filter FORWARD oif $wan-if ip daddr @drop-wan-dst-ip log prefix "FORWARD WAN SRC ESTO: " counter drop
add rule ip filter FORWARD oif $wan-if icmp type != @sallitut-icmp counter drop
add rule ip filter FORWARD iif @lan_vlans oif $wan-if counter accept
add rule ip filter FORWARD iif {enp5s0f1.100} oif enp5s0f1.99 counter accept
add rule ip filter FORWARD iif @lan_vlans oif @lan_vlans counter accept
```

```
add rule ip filter FORWARD counter drop

######### NAT
add table ip nat
add chain ip nat PREROUTING { type nat hook prerouting priority 0; }
add chain ip nat POSTROUTING { type nat hook postrouting priority 0; }
add rule ip nat POSTROUTING oif $wan-if masquerade fully-random

######### MANGLE
add table ip mangle
add chain ip mangle PREROUTING { type filter hook prerouting priority 0; }
add chain ip mangle FORWARD { type filter hook forward priority 0; }
add chain ip mangle POSTROUTING { type filter hook postrouting priority 0; }

# Langaton (queue :4)
add rule ip mangle FORWARD iif {enp5s0f1.50} counter meta priority set 1:4
add rule ip mangle FORWARD oif {enp5s0f1.50} counter meta priority set 1:4

# Adminit (queue :3)
add rule ip mangle FORWARD iif {enp5s0f1.60} counter meta priority set 1:3
add rule ip mangle FORWARD oif {enp5s0f1.60} counter meta priority set 1:3
```