TEKNILLINEN KORKEAKOULU

Elektroniikan, tietoliikenteen ja automaation tiedekunta

Tietoliikenne- ja tietoverkkotekniikan laitos

Eero Solarmo

Luotettava tiedonsiirtomenetelmä hybridiverkkoihin

Diplomityö, joka on jätetty opinnäytteenä tarkastettavaksi diplomi-insinöörin tutkintoa varten Espoossa 25.4.2008

Työn valvoja: Professori Raimo Kantola

Työn ohjaaja: Marko Luoma

TEKNILLINEN KORKEAKOULU **Diplomityön tiivistelmä**

Tekijä: Eero Solarmo

Työn nimi: Luotettava tiedonsiirtomenetelmä hybridiverkkoihin

Päivämäärä: 25.4.2008 Sivumäärä: 10+76

Tiedekunta: Elektroniikan, tietoliikenteen ja automaation tiedekunta

Professuuri: Tietoverkkotekniikka koodi: S-38

Työn valvoja: Prof. Raimo Kantola

Työn ohjaaja: TkL Marko Luoma

Nykyaikaisten tietoverkkojen monipuolistuessa käytettyjen teknologioiden määrä tulee lisääntymään. Langattomista ja langallisista komponenteista rakentuvat liityntäverkot ovat entistä laajemmassa käytössä jo nykyään. Tämä asettaa tavoitteita paitsi laitteiden yhteistoiminnalle, myös tiedonsiirron luotettavuudelle. Erityisesti langattomalla siirtotiellä esiintyvät häiriöt ja niistä johtuvat pakettihukat ovat uhka luotettavalle tiedonsiirrolle. Näiden pakettihukkien estäminen ja niistä palautuminen ovat tärkeitä keinoja luotettavuuden parantamiseksi.

Tässä työssä on mallinnettu ja simuloitu eräs menetelmä luotettavan tiedonsiirron toteuttamiseksi. Menetelmä on nimeltään HBH, ja se on hyppykohtaisiin negatiivisiin kuittauksiin perustuva menetelmä. Sen tarkoituksena on varmistaa luotettava tiedonsiirto korjaamalla linkillä tapahtuvat pakettihukat nopeasti ja tehokkaasti. Mallinnukseen on käytetty OPNET Modeler -ohjelmaa.

Simulaatioiden perusteella saatiin selville, että menetelmä parantaa linkkikohtaista luotettavuutta huomattavasti. Erityisesti 10-30% pakettihukilla toiminta tehostuu huomattavasti verrattuna varmistamattomaan tilanteeseen. Kvalitatiivinen analyysi TCP:n toiminnasta menetelmän kanssa antoi erittäin lupaavia tuloksia suorituskyvyn parantumisesta erityisesti kun pakettihukkaa alkoi esiintyä huomattavasti.

Avainsanat: moniteknologiaverkot, hybridiverkot, luotettava tiedonsiirto, hyppykohtainen kuittaus

HELSINKI UNIVERSITY OF TECHNOLOGY **Abstract of the Master's Thesis**

Author: Eero Solarmo

Name of the Thesis: Reliable Transport Method for Hybrid Networks

Date: April 25th, 2008 Number of pages: 10+76

Faculty: Faculty of Electronics, Communications and Automation

Professorship: Networking Technology code: S-38

Supervisor: Professor Raimo Kantola

Instructor: Lic.Sc.(Tech.) Marko Luoma

As modern communication networks become more diverse, the amount of utilised technologies will increase in the future. Access networks comprised of wireless and wired components are in even wider use than before. This sets goals not only for device interoperability, but also for transmission reliability. Especially wireless medium interference and packet loss caused by it is a threat to reliable communication in environments using those technologies. Avoidance of and recovery from these packet losses are key methods to improve reliability.

In this Thesis a method for reliable transmission is modeled and simulated. The method is called HBH (hop-by-hop) and it is a method based on hop-by-hop negative acknowledgements. Its purpose is to secure reliable transmission by repairing link-based packet losses quickly and efficiently. The method is modeled using OPNET Modeler.

Based on simulations we discovered that the method improves link-based reliability significantly. Especially when experiencing packet loss of 10 to 30 percent the communication is substantially improved compared to the case without any reliability support. Qualitative analysis on TCP operation gives very promising results on improving performance, especially when significant packet loss is present.

Keywords: mixed technology networks, hybrid networks, reliable communication, reliable transport, hop-by-hop acknowledgement

Esipuhe

Kiitokset diplomityön valmistumisesta kuuluvat työn valvojalle, Professori Raimo Kantolalle. Suuret kiitokset kuuluvat myös Marko Luomalle erittäin asiantuntevasta palautteesta ja ohjauksesta. Kiitokset myös muille projektin jäsenille (ei erityisessä järjestyksessä): Timo-Pekka Heikkinen, Juha Järvinen, Mika Ilvesmäki, Piia Töyrylä, Olli-Pekka Lamminen, Visa Holopainen ja Markus Peuhkuri.

Lisäksi kiitos perheelleni, erityisesti avopuolisolleni Elinalle tuesta tämän koitoksen aikana.

Espoossa, 25. huhtikuuta 2008.

Eero Solarmo

Lyhenteet

3G	3rd Generation
ABE	Available Bandwidth Estimator
AIMD	Additive Increase, Multiplicative Decrease
ATCP	TCP for Mobile Ad Hoc Networks
ATP	Ad hoc Transport Protocol
CSMA	Carrier Sense Multiple Access
CTS	Clear to Send
CW	Congestion Warning
DACK	Delayed Acknowledgement
DSA	Direct Spectrum Access
ECN	Explicit Congestion Notification
ENIC	Enhanced Interlayer Control
ESRT	Event to Sink Reliable Transport
FBcast	Fountain Broadcast
FEC	Forward Error Correction
hACK	hop-by-hop Acknowlegement

HBH	Hop by Hop
HRS	Hop by hop Reliability Support Scheme
ICMP	Internet Control Message Protocol
I/O	Input/Output
MAC	Medium Access Control
MSS	Maximum Segment Size
MTU	Maximum Transfer Unit
NACK	Negative Acknowledgement
PIC	Physical Interface Card
PSFQ	Pump Slowly, Fetch Quickly
RMST	Reliable Multi-segment Transport
RTO	Retransmission Timeout
RTS	Request to Send
RTT	Round-trip Time
SACK	Selective Ack
SMTP	Simple Mail Transfer Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
TCP-ELFN	TCP Explicit Link Failure Notification
TTL	Time to Live
WLAN	Wireless Local Area Network
WSN	Wireless Sensor Network

Sisällysluettelo

Kuvat

Taulukot

1. Johdanto

1.1. Tausta

Luotettava tiedonsiirto on elintärkeä ominaisuus monelle verkkosovellukselle. Sen toteuttaminen ei kuitenkaan ole aina täysin triviaali tehtävä. Nykyaikaiset verkot koostuvat usein hyvin monista erilaisista palasista, joiden vaikutus verkon toimintaan jää usein loppukäyttäjälle läpinäkyväksi.

Tämän työn tarkoituksena on esitellä ratkaisu luotettavan tiedonsiirron varmistamiseen tietoverkoissa, joiden rakenteista ei välttämättä ole aiempaa tietämystä. Toisin sanoen menetelmän on tarkoitus olla riippumaton verkon infrastruktuurista, oli tämä sitten täysin langaton, täysin kiinteä, tai sekä kiinteitä että langattomia elementtejä sisältävä hybridiverkko.

Tarkoituksena on toteuttaa tiedonsiirto verkon kahden pisteen välillä luotettavasti. Tiedonsiirrolla tarkoitetaan tässä yhteydessä nimenomaan pakettien mukana kulkevan informaation siirtoa, eikä niinkään sitä, että jokaisen yksittäisen paketin olisi päästävä perille. Luotettavuus tarkoittaa siirron onnistumisen lisäksi myös sitä, että tiedon eheys säilyy päätepisteiden välillä. Menetelmän tavoite on ainoastaan toteuttaa luotettava tiedonsiirto, joten palvelunlaatuun liittyviä seikkoja ei oteta erityisesti huomioon. Toisin sanoen tiedonsiirrolle ei anneta mitään takeita viiveen, viiveenvaihtelun tai kapasiteetin suhteen. Ainoastaan varsinaisen informaation eheys pyritään takaamaan.

Koska verkon rakenteesta ei tehdä mitään ennakko-oletuksia, menetelmän tulee olla mahdollisimman riippumaton reititys- ja siirtoprotokollista. Toisaalta joissakin erikoistapauksissa kommunikointi eri protokollakerrosten välillä saattaa olla välttämätöntä, joten menetelmällä olisi syytä olla jonkinlainen rajapinta siirtoyhteyskerroksen protokollan kanssa.

Verkko, jossa toimitaan, saattaa koostua mielivaltaisesta määrästä reitittimiä ja päätelaitteita. Tämän vuoksi hallintatiedon määrää tulee pyrkiä rajoittamaan, jotta menetelmä olisi skaalautuva.

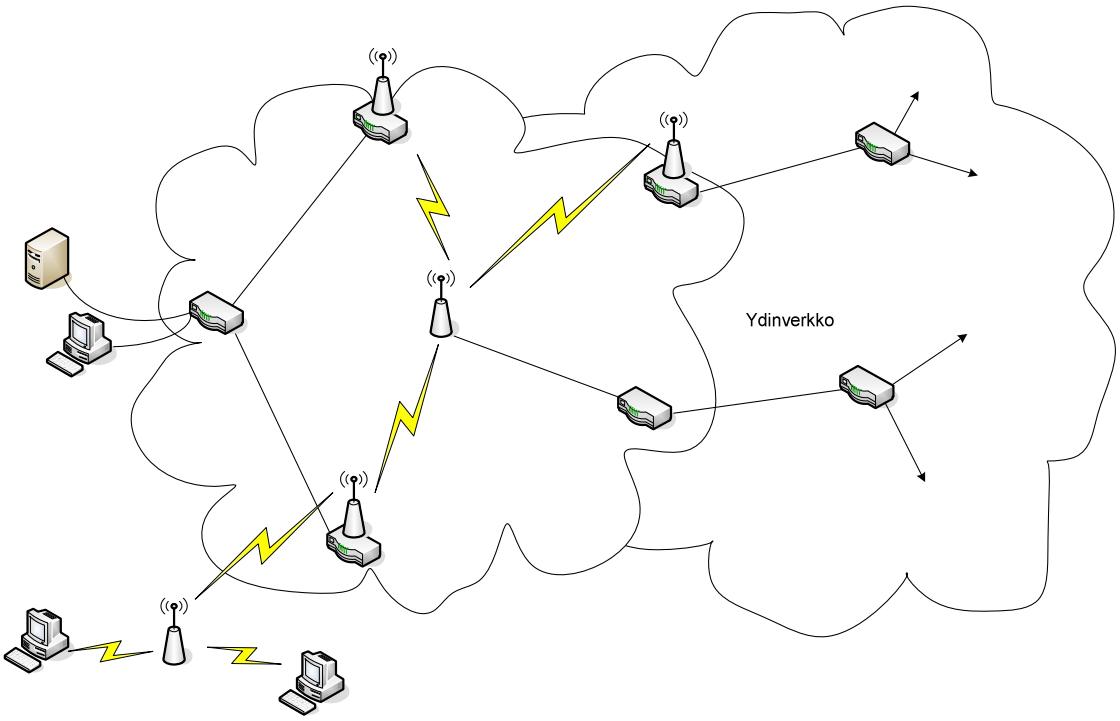
Skaalautuvuus tarkoittaa sitä, että verkon kasvaessa protokollan tai menetelmän suorittamat toimenpiteet tai prosessori- ja muistivaatimukset eivät lisäänny merkittävästi. Verkon ”äly” voidaan toteuttaa yhteen pisteeseen, jossa kaikki laskenta tapahtuu. Toisaalta toiminnallisuutta voidaan hajauttaa kaikkiin verkon pisteisiin. Jos verkossa on paljon liikennettä tai useita solmuja, keskitetty palvelin lisää ylimääräistä hallintaliikennettä. Myös skaalautuvuus muodostuu ongelmaksi, jos kaikki laskenta tehdään yhdessä pisteessä. Menetelmässä voi olla jokin ”keskipiste”, josta toimintaa johdetaan jollain tasolla, mutta menetelmä ei saa olla täysin riippuvainen tästä pisteestä. Parhaimmillaan tämä keskipiste olisi dynaaminen ja se voisi vaihtaa paikkaa sen mukaan, mistä on parhaimmat toimintaedellytykset.

Oletuksia viiveistä tai siirtokapasiteeteista ei voida tehdä, koska verkon linkit voivat olla joko kiinteitä tai langattomia. Radiolinkeillä voi esiintyä häiriöitä tai päällekkäistä liikennettä, jolloin niiden tarjoama kapasiteetti saattaa vaihdella huomattavastikin. Kapasiteettia on mitattava siis usein, jotta kapasiteetin pienentyessä ei lähetetä suurta pursketta tietoa, josta suuri osa todennäköisesti katoaisi.

Hybridiverkkojen vaihtelevan rakenteen vuoksi luotettavan tiedonsiirron varmistaminen on mielenkiintoinen haaste. Jos verkossa on useita langattomia kiinteitä tukiasemia, voidaan sen rungon ajatella olevan tavallaan ”kiinteä” ad hoc -verkko ilman liikkuvuutta. Tämän vuoksi työn lähteenä on käytetty useita ad hoc -verkkoihin kehitettyjä tiedonsiirron varmistusmenetelmiä. Nykyisissä langattomissa tekniikoissa (esim. 3G, WiMAX, WLAN) kapasiteetti on teoriassa vakio. Vaihtelua kapasiteetissa on tietysti radioympäristön häiriöiden tai muiden kuuluvuutta heikentävien seikkojen vuoksi. Tulevaisuudessa verkkoteknologiat kuitenkin kehittyvät, ja kasvava trendi on käyttää DSA:ta (Direct Spectrum Access) pääsynhallintaan ja niin sanottuja kognitiivisia radioita tämän tekniikan toteuttamiseen. DSA:n tullessa käyttöön nykyisten eri tekniikoille varattujen taajuuskaistojen välit hiipuvat pois, jolloin koko taajuusalue saadaan käyttöön. DSA:lla voidaan käyttää vapaata taajuuskaistaa dynaamisesti etsimällä vapaat taajuudet, joilla tietoa voidaan siirtää. Riippuen radioliikenteen määrästä, DSA:n tarjoama kapasiteetti saattaa vaihdella suuresti jolloin on entistä tärkeämpää, että kapasiteettimuutokset havaitaan nopeasti ja mahdolliset pullonkaulat pystytään kiertämään tarvittaessa.

Etenemistien valintaa voidaan optimoida eri kriteerien perusteella. Valinta voidaan tehdä esimerkiksi pienimmän viiveen, suurimman kapasiteetin, pienimmän virhetodennäköisyyden tai lyhimmän polun perusteella. Verkon ollessa epästabiilissa tilassa myöskään monitievälitystä ei kannata sulkea pois vaihtoehdoista. Monitievälityksessä liikenne lähetetään kohteeseen eri polkuja

ja kohde on yhteenkokoamispiste, jossa monesta suunnasta saapuvat paketit kootaan järkeväksi tietovirraksi.



Kuva 1: Liityntäverkko saattaa koostua lähes lukemattomasta määrästä erilaisia laite- ja linkkiyhdistelmiä

1.2. Tavoite

Tämän työn tavoitteena on esitellä ratkaisumenetelmä luotettavan tiedonsiirron toteuttamiseksi. Menetelmän tarkoitus ei ole olla koko Internetin kattava kokonaisvaltainen ratkaisu, vaan rajallisen kokoisessa liityntäverkossa toimiva ratkaisu. Liityntäverkon kokoa ei sen tarkemmin määritellä, vaan sen voidaan olettaa olevan mitä tahansa muutaman ja muutaman kymmenen laitteen väliltä. Menetelmä on nimetty HBH:ksi (*hop-by-hop*, *hyppy hypyltä*) sen hyppykohtaiseen kuittaukseen perustuvan toiminnallisuuden vuoksi. Menetelmä mallinnetaan ja simuloidaan OPNET Modeler ohjelmalla, ja saatujen tulosten perusteella analysoidaan sen toimintaa ja hyödyllisyyttä. TCP:n toimintaa menetelmän kanssa tarkastellaan teoreettisella tasolla. Lähtökohtana on erityisesti luotettavuuden toteutuminen.

1.3. Työn rakenne

Loput tästä työstä on jaoteltu seuraavasti: Luvussa 2 esitellään taustatietona käytettyjä aikaisemmin kehitettyjä menetelmiä samankaltaiseen ongelmaan. Luvussa 3 esitetään olemassa olevien tekniikoiden pohjalta ratkaisuja eri ongelmakohtiin, sekä pohditaan eri vaihtoehtoja näille ratkaisuille. Lisäksi esitellään välitysprosesseja. Luvussa 4 esitellään ratkaisun toteutukseen liittyvät yksityiskohdat. Luvussa 5 esitetään simuloinneista saadut tulokset menetelmälle. Lopuksi luvussa 6 esitetään johtopäätökset.

2. Tausta

2.1. Aikaisempi tutkimus

Aikaisemman tutkimuksen esittelyssä perehdytään langattomille linkeille ja langattomiin verkkoihin suunniteltuihin virheenkorjausmenetelmiin. Tämä valinta on tehty siksi, että kiinteillä linkeillä ei tapahdu radioympäristöstä johtuvia pakettien katoamisia. Kaikki katoamiset kiinteillä linkeillä johtuvat pääasiassa katkenneista linkeistä tai jonojen ruuhkautumisen aiheuttamista pakettien putoamisista. Käytännössä myöskään törmäyksiä ei tapahdu kiinteillä linkeillä luotettavien siirtotien saantimenetelmien (MAC, medium access control) ansiosta.

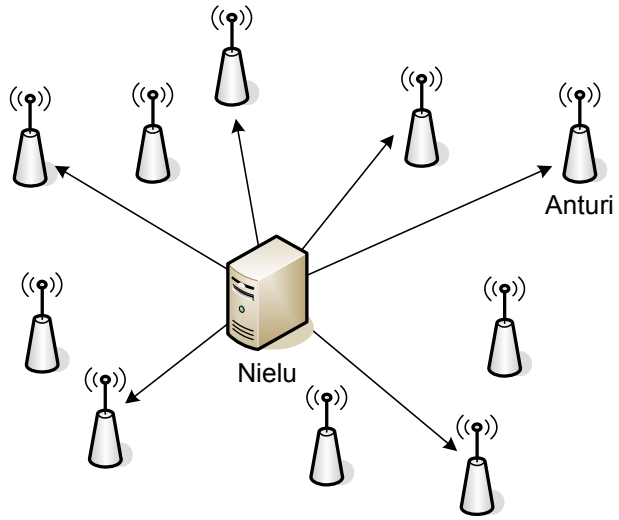
Tarkasteltuja menetelmiä ovat negatiivisiin kuittauksiin (NACK) perustuvat PSFQ, RMST ja HRS sekä lähetystiheyden säätelyyn perustuva ESRT. FBcast on virheenkorjauskoodeihin (Forward error correction) perustuva menetelmä. TCP-Jersey, ATP sekä ENIC ovat pääosin kuljetustasolla toimivia päästä päähän menetelmiä.

2.1.1. PSFQ

PSFQ[1] (Pump-Slowly, Fetch-Quickly) on langattomiin sensoriverkkoihin kehitetty robusti negatiivisiin kuittauksiin (NACK) perustuva kuljetusprotokolla, joka on suunniteltu toteuttamaan luotettava tiedonsiirto. PSFQ on melko yksinkertainen lähestymistapa, koska se tekee vähimmäisoletukset reititysinfrastruktuurista sekä on skaalautuva ja energiatehokas. Lisäksi se reagoi nopeasti verkossa tapahtuviin virhetiloihin, joten se toimii myös virhealttiissa ympäristössä. PSFQ:n pääidea on lähettää tietoa suhteellisen hitaasti (pump-slowly), mutta toisaalta mahdollistaa tietoa menettäneiden solmujen nopea tiedon palauttaminen aggressiivisesti lähimmiltä naapurisolmuilta (fetch-quickly).

Toimintaympäristö

PSFQ on tarkoitettu käytettäväksi langattomissa sensoriverkoissa (Wireless Sensor Network, WSN). Langattomat sensoriverkot ovat verkkoja, joissa erilaiset sensorit lähettävät tietoja tilastaan (esimerkiksi lämpötila) niin kutsutulle nielulle (sink). Nielu on mikä tahansa laite tai keskusyksikkö, jossa tiedot käsitellään ja tallennetaan. Usein sensorilta nieluun siirrettävän tiedon varmistus ei ole kriittistä, sillä satunnainen tiedon puuttuminen (esimerkiksi lämpötila jollain hetkellä) ei välttämättä haittaa koko järjestelmän toimintaa. PSFQ onkin kehitetty pääasiassa siksi, että nielusta voidaan siirtää tietoa luotettavasti sensoreille. Tieto voi tässä tapauksessa olla vaikka pätkä koodia, jolla sensorit ohjelmoidaan uudelleen. Tällaisessa tilanteessa paketin katoaminen johtaa todennäköisesti koko uudelleenohjelmoinnin epäonnistumiseen. Kaikki sensoriverkot ovat sovelluskohtaisia, joten protokollaa on vaikea yleistää kaikkiin tapauksiin soveltuvaksi. PSFQ on kuitenkin räätälöitävissä eri sovellusten tarpeisiin sopivaksi.



Kuva 2: PSFQ toimii pääsääntöisesti tiedonsiirrossa nielusta sensoreille

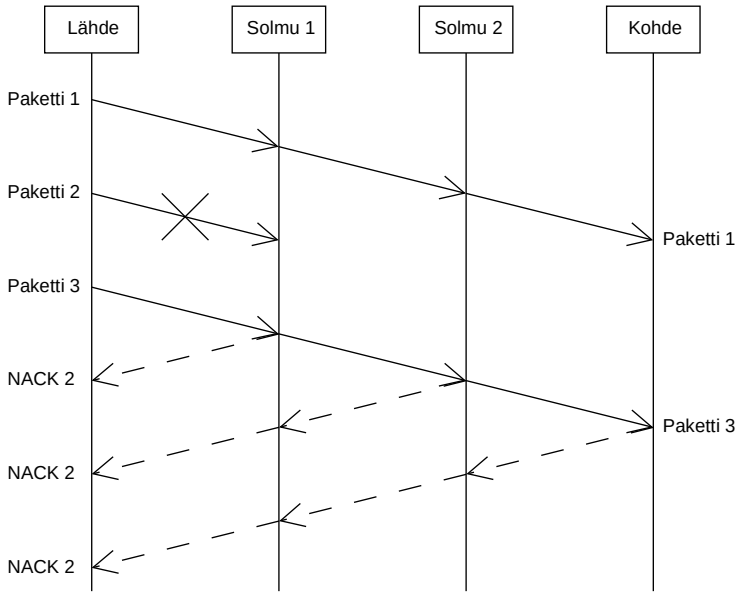
PSFQ:n kehittäjien mukaan suurin ongelma päästä päähän virheenkorjaukselle on langattomien verkkojen fyysiset ominaisuudet. Sensoriverkot saattavat toimia ankarissa radioympäristöissä ja luottavat usean hypyn välitystekniikoihin. Virheet kasaantuvat eksponentiaalisesti usean hypyn yli, jolloin pakettien katoamisen ja uudelleenjärjestymisen todennäköisyys kasvaa hyppymäärän kasvaessa. Jos pakettien katoamistodennäköisyys on *p*, todennäköisyys saada paketti perille päästä päähän putoaa nopeasti kaavan $(1-p)^n$ mukaisesti, jossa *n* on hyppyjen määrä. PSFQ:n kehittäjät

arvioivat, että on lähes mahdotonta saada paketti perille suuressa verkossa käyttäen päästä päähän lähestymistapaa, jos pakettien katoamistodennäköisyys on enemmän kuin 10 %.

Toiminnallisuus

PSFQ tarkkailee hyppy hypyltä pakettien sekvenssinumeroita ja päättelee niiden perusteella onko paketteja kadonnut. Tällöin lähteen ja kohteen välissä olevat solmutkin osallistuvat virheiden havaitsemiseen ja korjaukseen. Tämä lähestymistapa itse asiassa paloittelee usean hypyn välityspolun sarjaksi yhden hypyn mittaisia lähetysprosesseja. Hyppy hypyltä lähestymistapa skaalautuu paremmin ja on virhesietoisempi kuin päästä päähän lähestymistapa. Lisäksi se vähentää pakettien uudelleenjärjestymistä.

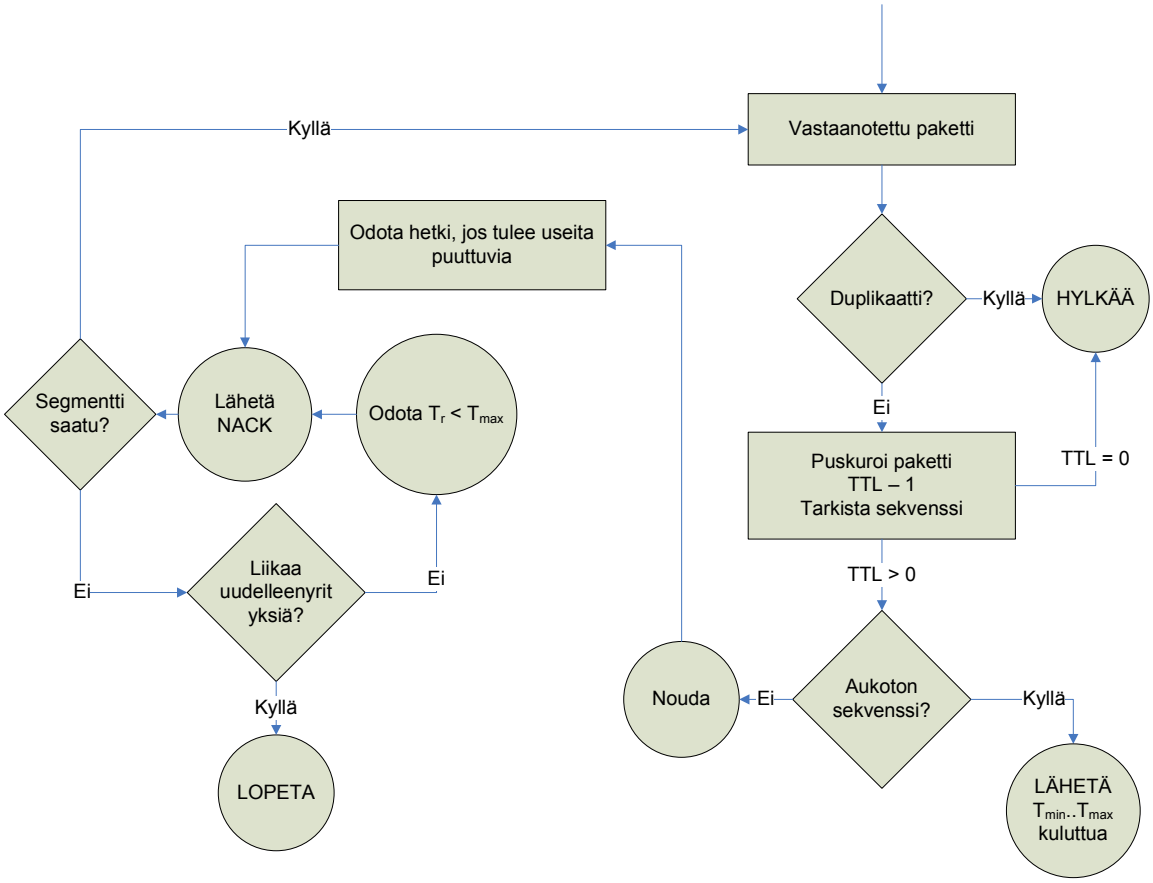
Normaalissa NACK-järjestelmässä paketin katoaminen saattaa levitä alavirrassa oleville solmuille, jos kadonnutta pakettia korkeamman sekvenssinumeron sisältäviä paketteja välitetään jatkuvasti. Tällöin jokaisella seuraavalla solmulla välistä puuttuvan paketin huomaaminen käynnistää hakuoperaation alavirrasta. Millään solmulla alavirrassa ei pakettia kuitenkaan ole, joten NACK-viestit saattavat edetä jopa lähteelle asti. Tätä tapahtumaketjua kutsutaan PSFQ:n kehittäjien mukaan ”NACK-tapahtuman etenemiseksi”. Tämän takia on välttämätöntä varmistaa, että solmut välittävät vain paketit, jotka jatkavat sekvenssiä.



Kuva 3: NACK-tapahtuman eteneminen.

Kuvassa 3 on esitetty NACK-tapahtuman eteneminen. Kuvassa havaitaan miten paketti 2 katoaa ensimmäisellä hypyllä. Paketti 3 välitetään kuitenkin eteenpäin, jolloin myöhemmät solmut lähettävät negatiivisen kuittauksen paketista 2, joka etenee aina lähteelle asti.

PSFQ sisältää NACK-tapahtuman etenemisen vuoksi myös välimuistiominaisuuden, jolla varmistetaan sekä pakettien lähettäminen oikeassa järjestyksessä että täydellinen virheenkorjaus kaikille hakuoperaatiolle alavirtaan päin. Lokalisoimalla katoamistapahtumat ja olemalla lähettämättä (kadonnutta pakettia) korkeamman sekvenssinumeron paketteja, mekanismi toimii store-and-forward -lähestymistavalla. Tämä auttaa entisestään toimintaa virhealttiissa ympäristössä, koska lähetykset pilkotaan periaatteessa yhden hypyn mittaisiksi lähetysprosesseiksi.



Kuva 4: PSFQ:n toimintaperiaate tilakaaviona.

Kuvassa 4 on esitetty PSFQ:n toiminta yksinkertaistettuna tilakaaviona. Lyhyesti sanottuna vastaanotetusta paketista tarkistetaan löytyykö se jo laitteen puskurista ja duplikaatit hylätään. Jos saapunutta pakettia aiempia paketteja puuttuu, käynnistetään nouto-operaatio ja kadonneista paketeista lähetetään NACKit.

PSFQ:ssa on implementoitu lisäksi proaktiivinen nouto sen varalta, että datan viimeinen paketti katoaa. Tällöin vastaanottajalla ei ole mitään tietoa, kuuluisiko dataerään vielä lisää paketteja. Siksi vastaanottaja lähettää ajan T_{pro} (joka valitaan radio-olosuhteista ja sovelluksesta riippuen) kuluttua NACKin viimeisintä vastaanotettua pakettia seuraavasta paketista. Jos paketti on todella kadonnut, tällä paikataan puuttuva paketti, mutta NACKin ollessa aiheeton eikä paketteja enää ole NACK hylätään.

2.1.2. RMST

RMST[2] eli Reliable Multi-Segment Transport on kuljetustason protokolla suunnatulle diffuusiolle[3] (Directed Diffusion) langattomissa sensoriverkoissa. RMST tarjoaa taatun toimituksen sekä sirpalointi ja uudelleenjärjestely (fragmentation/reassembly) -toiminnan sovelluksille, jotka niitä tarvitsevat. RMST perustuu selektiivisiin NACKeihin, ja se voidaan konfiguroida verkonsisäisen välimuistin käyttöön ja korjaukseen. RMST:ssä on implementoitu komponentteja sekä siirtoyhteys- että siirtokerroksille.

RMST-protokolla on implementoitu eräänlaisena suodattimena, joka voidaan liittää mihin tahansa solmuun tarvittaessa, ilman että reititysprotokollaan tarvitsee kajota. RMST:ssä luotettavuus tarkoittaa sitä, että uniikkiin RMST-olioon liittyvä data toimitetaan lopulta kaikille tilaajasolmuille. Uniikki RMST-olio tarkoittaa dataerää, johon kuuluu yksi tai useampi samasta lähteestä tuleva paketti (fragment). RMST ei takaa pakettien oikeaa toimitusjärjestystä eikä tarjoa mitään reaaliaikatakeita.

RMST:ssä vastaanottajat ovat vastuussa pakettien mahdollisesta uudelleenlähetyksestä. Välimuistittomassa toimintatavassa ainoastaan nielut tarkkailevat RMST-olioiden eheyttä vastaanotettujen pakettien perusteella. Välimuistillisessa toimintatavassa RMST-solmu ”kerää” paketteja ja pystyy lähettämään puuttuvia paketteja polun seuraavalle solmulle sekä myös pyytämään puuttuvia paketteja edelliseltä solmulta.

2.1.3. ESRT

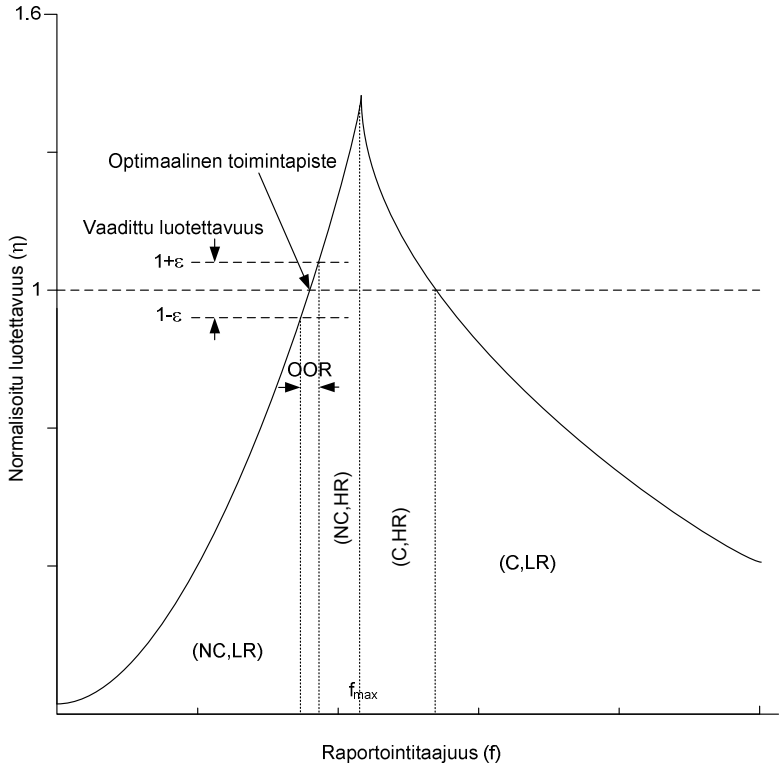
ESRT[4] (Event-to-Sink Reliable Transport) on myös eräs langattomiin sensoriverkkoihin suunniteltu protokolla. Sen keskeinen teema on se, että päästä päähän tiedonsiirtovarmistus ei ole sopiva sensoriverkkojen maailmaan, vaan se tuhlaa turhaan rajallisia resursseja. PSFQ[1] on samankaltainen kuin ESRT, mutta se toimii pääsääntöisesti eri suuntaan. Kun PSFQ toimii nielusta sensoreille, on ESRT:n tarkoitus varmistaa sensoreiden lähettämän tiedon päätyminen nieluun.

ESRT:n tarkoitus on toteuttaa luotettava tapahtumien havaitseminen vähäisimmällä mahdollisella energiankulutuksella. Lisäksi se sisältää ruuhkanhallintakomponentin, jonka avulla luotettavuus ja energiansäästö toteutuvat. ESRT:n algoritmit ajetaan pääsääntöisesti nielussa, jolloin sensoreissa tarvittava toiminnallisuus saadaan minimoitua. ESRT:n toiminta määräytyy verkossa havaitun luotettavuuden ja ruuhkatilanteen perusteella. Tämä itsekonfiguroituva luonne tekee ESRT:stä robustin satunnaiseen ja dynaamiseen topologiaan sensoriverkoissa.

Toimintaperiaate

ESRT:n toimintaa varten on määritelty muutamia avainlukuja. τ on aikamääre, jonka välein nielu tekee päätöksiä, toisin sanoen päätöksentekoväli. Havaittu tapahtuman luotettavuus r_i on nielussa vastaanotettujen pakettien määrä tarkasteluvälillä i. Tavoiteltu tapahtuman luotettavuus R taas määrittelee, montako pakettia pitää vastaanottaa, jotta havainto tapahtumasta olisi luotettava. R valitaan sovelluskohtaisesti. Raportointitaajuus f sensorilla on lähetettyjen pakettien määrä aikayksikössä. Menetelmän tavoitteena on siis säädellä sensoreiden raportointitaajuutta f siten, että nielussa saavutetaan tavoiteltu luotettavuus R tapahtuman havainnosta ilman, että resursseja kuluu turhaan.

Tapahtuman luotettavuuden mittaria merkitään symbolilla $\eta = r/R$. Tavoitteena on siis operoida niin lähellä arvoa $\eta = 1$ kuin mahdollista minimoiden samalla resurssien käyttö. Arvo 1 tarkoittaa siis sitä, että otetaan vastaan tarkalleen niin monta pakettia kuin tapahtuman luotettava havaitseminen edellyttää. Tätä kutsutaan optimaaliseksi toimintapisteeksi. Arvon $\eta = 1$ ympärille määritellään 2ε:n levyinen toleranssialue, jossa ε on eräs protokollan parametri. Raportointitaajuudelle on löydettävissä arvo f_{max}, joka on suurin mahdollinen sensoreiden raportointitaajuus, jonka verkko pystyy käsittelemään ilman ruuhkautumista. Kuvassa 5 on esitetty tapahtuman luotettavuus raportointitaajuuden funktiona sekä edempänä kuvaillut tunnusomaiset toiminta-alueet. Kuvassa nähdään miten luotettavuus kasvaa kun raportointitaajuus kasvaa, mutta f_{max}:n jälkeen luotettavuus alkaa laskea.



Kuva 5 ESRT:n tapahtuman luotettavuus raportointitaajuuden funktiona ja toiminta-alueet

Järjestelmälle voidaan määritellä viisi tunnusomaista toiminta-aluetta *f*:n ja η:n funktiona, käyttäen seuraavia rajoja:

(**NC,LR)**: $f < f_{max}$ ja $\eta < 1 - \varepsilon$ (No Congestion, Low Reliability eli ei ruuhkaa, matala luotettavuus)

OOR: $f < f_{max}$ ja $1 - \varepsilon \leq \eta \leq 1 + \varepsilon$ (Optimal Operating Region eli optimaalinen toiminta-alue)

(NC,HR): $f \leq f_{max}$ ja $\eta > 1 - \varepsilon$ (No Congestion, High Reliability eli ei ruuhkaa, korkea luotettavuus)

(C,HR): $f > f_{max}$ ja $\eta > 1$ (Congestion, High Reliability eli ruuhkaa, korkea luotettavuus)

(C,LR): $f > f_{max}$ ja $\eta \leq 1$ (Congestion, Low Reliability eli ruuhkaa, matala luotettavuus)

(NC,LR) tarkoittaa sitä, että raportointitaajuus on niin harvaa, että ruuhkautumista ei synny, mutta tapahtumien havaitsemisen luotettavuus ei kuitenkaan ole riittävällä tasolla. **(NC,HR)** tarkoittaa sitä, että raportointitaajuus on vähemmän kuin f_{max}, mutta tuhlaa resursseja, koska luotettavuuden saavuttamiseen riittäisi pienempikin taajuus. **(C,HR)**:ssä raportointitaajuus on jo liian korkea, jolloin verkko ruuhkautuu. Luotettavuus on kuitenkin vielä korkeampi kuin haluttu luotettavuus. **(C,LR)**:ssä raportointitaajuus on niin suuri, että verkko ruuhkautuu täysin, eikä paketteja saada

nielulle asti, jolloin luotettavuustavoite ei toteudu. Tavoite on siis säädellä sensorien raportointitaajuutta siten, että pysytään **OOR**-tilassa. Tällöin saavutetaan vaadittu luotettavuus minimaalisella energiankulutuksella.

Nielu tarkkailee luotettavuutta ja ruuhkaa jokaisella tarkkailuvälillä, ja lähettää sen perusteella sensoreille päivitystiedot. Käytännössä jos ollaan tilassa **(NC,LR)** tai **(C,LR)**, jolloin luotettavuustaso on liian pieni, nielu muuttaa raportointitaajuutta aggressiivisesti, jotta vaadittava luotettavuustaso saavutettaisiin mahdollisimman nopeasti. Tiloissa **(NC,HR)** ja **(C,HR)** ESRT pyrkii säästämään niukkoja energiaresursseja. Päätavoite on kuitenkin luotettava tapahtumien havainnointi, joten näissä tiloissa ESRT laskee *f*:ää hallitusti, kunnes saavutetaan **OOR**-tila. Ruuhkan havaitseminen on toteutettu yksinkertaisesti siten, että sensorit lähettävät nielulle tiedon ruuhkautumisesta, jos niiden reitityspuskuri täyttyy paketeista.

2.1.4. HRS

HRS[5] (Hop-by-Hop Reliability Support Scheme) on aiemmin esiteltyjen langattomissa sensoriverkoissa toimivien siirtoprotokollien kaltainen ratkaisu, mutta se ottaa huomioon myös reititysmuutokset ja skaalautuu paremmin. HRS perustuu hyppykohtaisiin sekvenssinumeroihin.

HRS:n pääominaisuuksia ovat hyppykohtaiset sekvenssinumerot virheenhavaitsemista ja -korjausta varten erityisesti reittimuutostilanteissa, pakettihukan havaitseminen hypyllä jolla se tapahtuu, sekä istunnonhallintatiedon vähentäminen usean lähettäjän tilanteissa. HRS toteuttaa myös kaksi toimintatapaa: unicast-toimintatapa ”sensoreilta nieluun” ja broadcast-toimintatapa ”nielusta sensoreille”. Näille toimintatavoille on toteutettu hieman erilainen luotettavuuden varmistaminen. Sensorit lähettävät tietoa nieluun vain unicastina, koska kohteena on vain yksi solmu, eli nielu. Toisaalta taas nielun lähettäessä tietoa sensoreille, solmujen on parempi lähettää tieto eteenpäin broadcastina, koska kohteena on monta solmua.

Aiemmissa esitellyissä protokollissa käytetään päästä päähän sekvenssinumeroita hyppy hypyltä virheenkorjaukseen. Jos jokin solmu liittyy kesken istunnon, sillä ei tällöin ole tietoa viimeisimmän paketin sekvenssinumerosta, joten se ei voi heti osallistua virhekorjaukseen. Solmujen tulisi pystyä osallistumaan virheenkorjaukseen, vaikka ne liittyisivät keskellä istuntoa.

Päästä päähän sekvenssejä käytettäessä ei voida olla varmoja, tapahtuuko paketin katoaminen solmun ja sen edeltäjän välillä vai aiemmin. Tästä voi aiheutua ”NACK-tapahtuman eteneminen”[1]. HRS:ssä virheenkorjaus tapahtuu sillä välillä, jolla paketti on kadonnut, joten tätä

tapahtumaa ei pääse syntymään. NACKeihin perustuvat menetelmät ovat tehokkaita pakettihukasta toipumiseen, mutta ne kärsivät joistakin ongelmista, kuten kokonaisten (pienten) viestien tai sekvenssin viimeisten pakettien katoamisista. Yksi HRS:n tavoitteista on toteuttaa NACK-pohjainen toiminta ja samalla välttää kyseiseen toimintatapaan liittyvä viimeisten pakettien katoaminen.

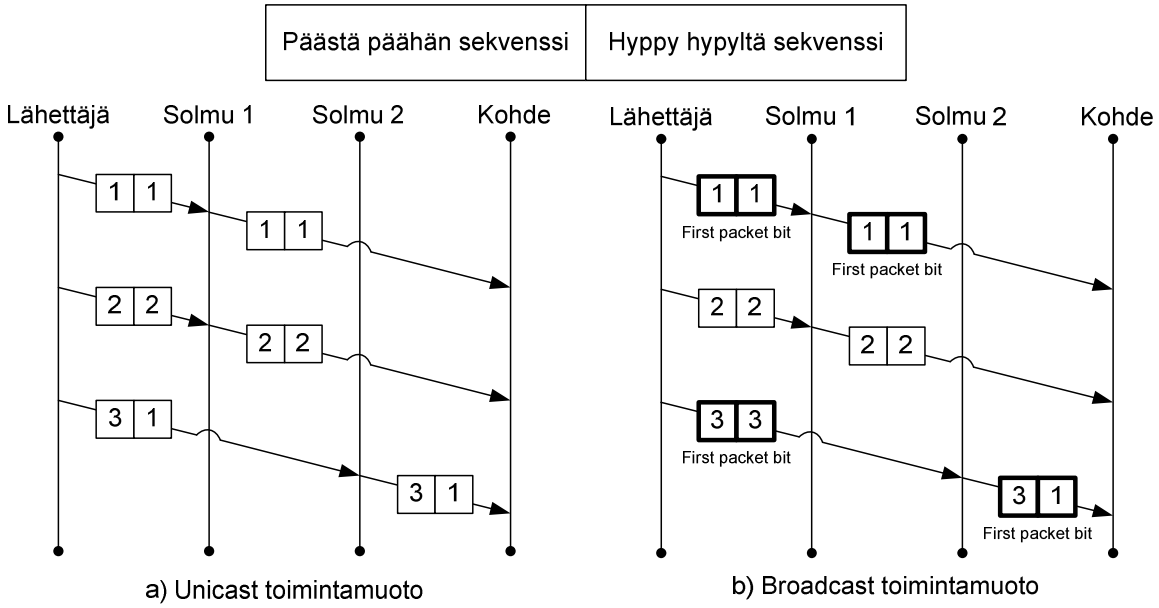
Unicast-toimintatapa

Unicast-toimintatavassa jokainen solmu ylläpitää hyppykohtaista sekvenssinumeroa kaikille naapurisolmuille. Lähettäjä alustaa sekvenssinumeron nollaksi kun se lähettää paketin kyseiselle naapurille ensimmäistä kertaa. Sekvenssinumeroa ylläpidetään jatkuvasti ja se on merkityksellinen vain solmun ja seuraavan hypyn solmun välillä huolimatta istunnoista. Tämän vuoksi solmujen ei tarvitse ylläpitää tietoa istunnoista ja lähettäjistä. Tämä lupaa skaalautuvuutta tiedonsiirrossa sensoreilta nieluun.

Koska solmut eivät ylläpidä istuntotietoa, ne voivat liittyä reitille välittömästi. Kun reitti muuttuu, lähettävä solmu huomaa reittimuutoksen ja alustaa uuden reitin seuraavan hypyn sekvenssinumeron nollaksi. Nyt pakettien hyppykohtaiset sekvenssinumerot alkavat alusta, joten uuden solmun ei tarvitse ottaa kantaa siihen, mitä aiemmin on lähetetty, vaan lähettävä solmu on hoitanut lähetyksen entisen hypyn kautta. Lisäksi yhdellä linkillä useastakin lähteestä tuleva data aggregoituu saman juoksevan hyppykohtaisen sekvenssinumeron alle.

Broadcast-toimintatapa

Broadcast-toimintatavassa solmut ylläpitävät kaikille naapurisolmuille yhtä yhteistä sekvenssiä, joka alustetaan nollaksi jokaisen uuden istunnon alussa. Tällä sekvenssinumerolla broadcast-lähetetään paketit kaikille istunnon seuraavan hypyn solmuille kerralla. Uuden liittyvän solmun tarvitsisi tietää hyppy hypyltä sekvenssin aloitusnumero, jotta se pystyisi liittymään istuntoon. Tämän vuoksi lähettävä solmu asettaa lähetettävään pakettiin ensimmäistä pakettia merkitsevän bitin (”first packet bit”), kun se huomaa reititystaulussa muutoksen. Kun uusi solmu havaitsee tämän bitin, se tietää, ettei paketteja ole kadonnut ennen kyseistä pakettia. Toisaalta jos bittiä ei ole asetettu, solmu olettaa joidenkin pakettien kadonneen ennen vastaanotettua pakettia.

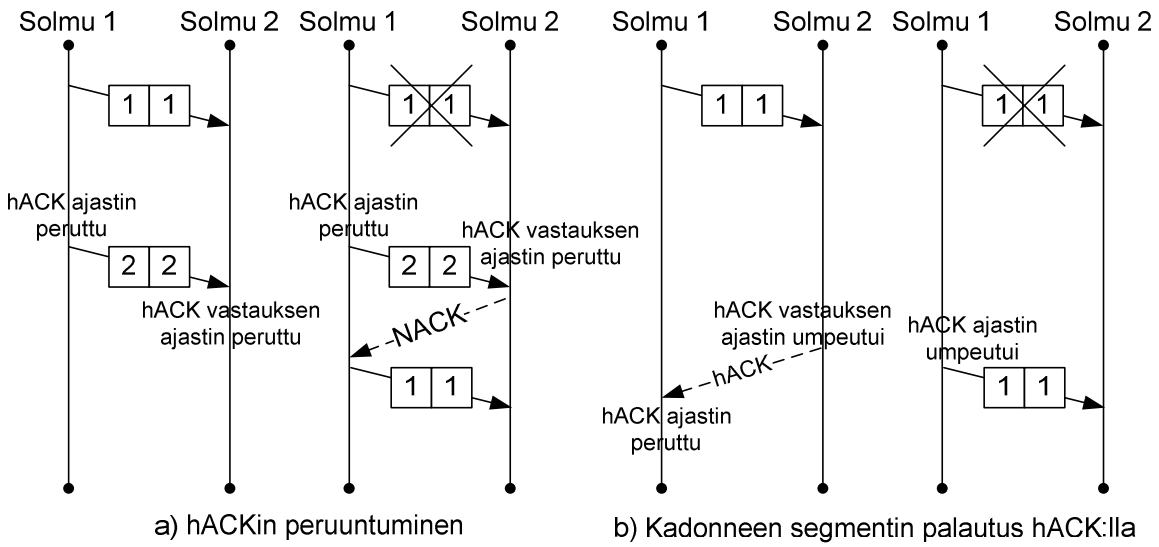


Kuva 6: HRS:n välitystoiminta reittimuutoksen tapahtuessa

Kuvassa 6 on esitetty kummankin toimintamuodon periaate. Laatikoissa vasemmanpuoleinen numero on päästä päähän sekvenssinumero ja oikeanpuoleinen hyppykohtainen sekvenssinumero.

Kokonaisten viestien ja viimeisten pakettien katoamista varten HRS:n NACK-pohjaista lähestymistapaa on täydennetty ACK-pohjaisella lähestymistavalla. Koska solmut eivät ylläpidä tietoa istunnoista, HRS käsittelee mitä tahansa pakettia viimeisenä pakettina, jos sitä ei seuraa uusi paketti järkevän ajan kuluessa. Tätä varten on toteutettu kaksi ajastinta.

Kun solmu lähettää paketin, se asettaa hACK-ajastimen (hop-by-hop ACK) ja vastaanottava solmu asettaa hACK-ajastimen vastaanottaessaan paketin. Jos paketin jälkeen ei saavu enää uusia paketteja vastaanottajan hACK-ajastimen kuluessa, vastaanottaja lähettää lähettäjälle hACK-viestin, jolloin lähettäjä tietää viimeisimmän paketin menneen perille. Jos vastaanottajalta ei kuulu hACK-viestiä lähettäjän ajastimen kuluessa, paketin oletetaan kadonneen ja se lähetetään uudelleen. hACK-ajastin peruuntuu myös uuden paketin saapuessa lähettäjän lähetysjonoon. Muiden kuin sekvenssin viimeisten pakettien kanssa käytetään normaalisti NACK-viestejä.



Kuva 7: Viivästetty hACK -lähestymistapa

Useita paketteja lähetettäessä suurin osa hACK-viesteistä jää lähettämättä uusien pakettien nollatessa ajastimet ja niiden katoamiset havaitaan sekvenssinumeroissa olevista aukoista. Viimeiset paketit tulevat silti lähetetyiksi perille hACK-viestien avulla.

Pidemmät ajat hACK-ajastimissa viivästävät virheenkorjausta, mutta aiheuttavat vähemmän hACK-viestejä. Lyhyemmät ajat taas nopeuttavat virheenkorjausta käyttämällä enemmän hACK-viestejä. Muuttamalla hACK-ajastimen arvoa sovellukset voivat tehdä kompromisseja hallintatiedon määrän ja pakettitoimituksen nopeuden välillä.

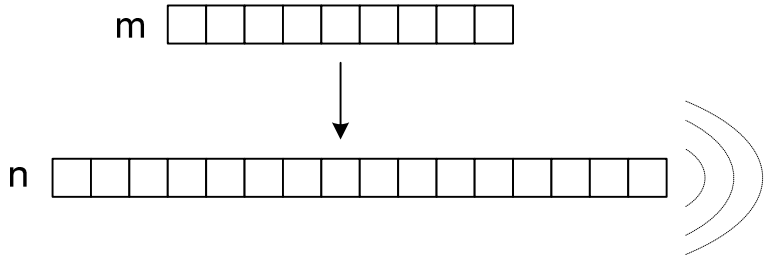
2.1.5. FBcast

FBcast[6] on protokolla, joka on suunniteltu langattomaan broadcast-lähetykseen langattomissa sensoriverkoissa. Se perustuu redundantteihin suihkulähdekoodeihin. Näistä kerrotaan tarkemmin edempänä. Protokollan tavoite on vähentää viestiliikenteen määrää broadcast-lähetyksessä sekä parantaa luotettavuutta.

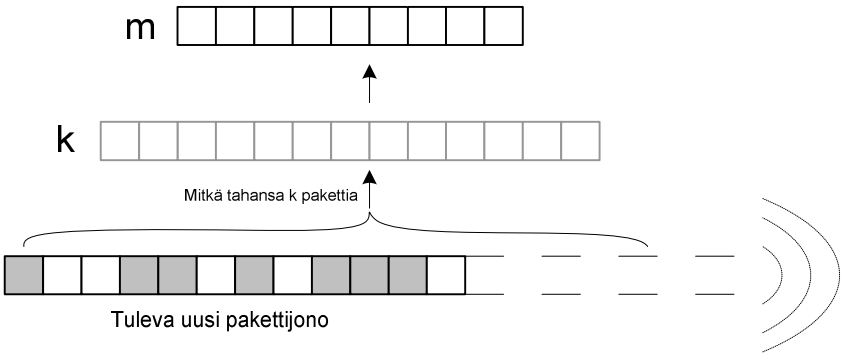
Normaalisti sensoriverkoissa jokainen solmu uudelleenlähettää kaiken saamansa datan, mikä johtaa usein tarpeettomiin lähetyksiin. Tämän vuoksi energiaa kuluu turhaan, minkä lisäksi lisääntynyt viestiliikenne lisää törmäysten määrää verkossa ja heikentää luotettavuutta. Broadcast-lähetystä käytetään usein tärkeänkin liikenteen lähettämiseen, kuten ohjelmistopäivityksiin. On selvää, että lähetyksen on oltava luotettavaa, jotta solmut saavat päivitykset ehjänä ja verkko pysyy toiminnassa ja yhtenäisessä tilassa.

FBcastin kehittäjien mukaan tulevaisuuden sensoreissa tulee olemaan entistä enemmän laskentatehoa, kun taas radioympäristö tulee pysymään lähes muuttumattomana taustahäiriön ja muiden tekijöiden vuoksi. Näin ollen onkin kannattavaa panostaa hieman laskentatehoa turhan viestinnän vähentämiseen. FBcast perustuu forward error correction (FEC) koodeihin.

Suihkulähdekoodi (fountain code[7]) on FEC-menetelmä, jolla tietoa voidaan siirtää luotettavasti. Lähetetään esimerkiksi viesti, jonka koko on *m* pakettia. Viesti ”räjäytetään” *n*:n (n > m) paketin mittaiseksi koodaamalla se jollain sovitulla menetelmällä. Kun vastaanottaja on saanut *k* pakettia (k ≥ m), se pystyy rakentamaan alkuperäisen informaation näiden pakettien avulla. Menetelmän analogian voi ajatella siten, että jos asetat kupin suihkulähteen alle, ei ole tärkeää mitkä yksittäiset pisarat tippuvat kuppiin, vaan että kuppi tulee täyteen *joistakin* satunnaisista pisaroista.



Kuva 8 FBcast lähteessä: m alkuperäistä pakettia koodataan n:ään pakettiin



Kuva 9: FBcast vastaanottajalla: k pakettia valitaan vastaanotetuista ja dekoodataan m:ksi

FBcastissa jokainen solmu, joka vastaanottaa uuden paketin lähettää sen eteenpäin todennäköisyydellä *p*. Siemenluku ja satunnaislukugeneraattori oletetaan olevan kaikkien solmujen tiedossa. FBcast tarjoaa seuraavia etuja: *n/m* (tunnetaan myös nimellä venytyskerroin) voidaan asettaa mielivaltaisen pitkäksi tai lyhyeksi joustavasti. Lisäksi pakettikokoa ei ole rajoitettu. Viimeisenä, koodauksen ja dekoodauksen pitäisi olla resurssimielessä halpaa. Kaikki suihkulähdekoodit eivät ole yhtä joustavia, ja kehittäjät ovatkin kiinnostuneita lähinnä LT-, Raptor- ja Online-koodeista.

Tiedon koodaamisesta on kolminkertainen etu: Parantunut luotettavuus, joka saavutetaan siten, että lisätään ylimääräistä tietoa paketteihin. Tämän vuoksi solmu pystyy generoimaan alkuperäisen paketin, vaikka se saisi esimerkiksi kohinan vuoksi vastaanotettua vain osan paketeista. Ylimääräisen lähetysdatan määrä vähenee, koska redundanssin vuoksi kaikkia paketteja ei tarvitse vastaanottaa tai lähettää eteenpäin, jolloin jaetusta kanavasta kilpaileminen vähentyy. Lisäksi koodaus tarjoaa datan luottamuksellisuuden. Vastaanottajan on tiedettävä käytetty satunnainen siemenluku, jotta lähetetyn viestin voi uudelleenrakentaa. Tämän vuoksi salakuuntelijoiden on vaikea purkaa lähetettyä informaatiota.

2.1.6. TCP-Jersey

TCP:n suorituskyvyn heikkeneminen langattomissa ja hybridiverkoissa johtuu pääasiassa siitä, ettei TCP osaa erottaa ruuhkautumisesta johtuvan ja radiolinkkien virheistä johtuvan pakettihukan välillä. TCP havaitsee vain kadonneen paketin, eikä katoamisen syystä ole olemassa mitään tietoa. TCP-Jersey[8] on TCP-variantti, joka osaa erottaa langattomilla linkeillä katoavat paketit ruuhkan takia katoavista paketeista ja toimia sen mukaisesti. Se koostuu kahdesta pääosa-alueesta: vapaan kaistanleveyden arvointi (available bandwidth esimation, ABE) ja ruuhkavaroitus (congestion warning, CW).

ABE on lähettäjäpuolen lisäys, joka tarkkailee jatkuvasti yhteydelle saatavilla olevaa kaistaa ja säätelee lähetysnopeutta kun ruuhkautuminen uhkaa. ABE:n toiminta perustuu siihen, että se tarkkailee ACKien saapumisnopeutta. Koska lähetetyn paketin koko on tiedossa, se pystyy viiveestä estimoimaan käytettävissä olevan kaistanleveyden.

TCP-Tahoe ja –Reno käyttävät AIMD (Additive increase, multiplicative decrease) algoritmia, jolloin kaistanleveyden arviointi on hyvin karkeaa ja enemmänkin reaktiivista kuin proaktiivista. TCP-Westwood[9] käyttää lähettäjäpäässä kaistanleveyden estimaattoria, joka perustuu palaavien kuittausten aikaväleihin. Vastaanotettaessa kuittaus ajanhetkellä t_k, kaistanleveydestä saadaan näyte kaavalla

$$b_k = \frac{d_k}{t_k - t_{k-1}} \tag{1}$$

missä d_k on kuitatun datan määrä ja t_{k-1} on edellisen vastaanotetun kuittauksen ajanhetki. Tätä kaistanleveysarviota vielä suodatetaan alipäästösuotimella, jotta suuret heilahtelut saadaan poistettua.

TCP-Jersey tarkkailee myös saapuvia kuittauksia. Estimaattori on johdettu ajassa liikkuvan ikkunan periaatteesta, jota on ehdottanut Clark ja Wang [10]. Heidän ehdotuksessaan reititin arvioi yksittäisen vuon käyttämää kaistanleveyttä seuraavasti:

$$R_n = \frac{T_w \times R_{n-1} + L_n}{(t_n - t_{n-1}) + T_w} \tag{2}$$

missä R_n on arvioitu kaistanleveys kun paketti n saapuu ajanhetkellä t_n, t_{n-1} on edellisen paketin saapumisaika, L_n on paketin koko ja T_w on vakio aikaikkuna.

TCP-Jersey arvioi vapaata kaistanleveyttä lähettäjällä. ABE tarkkailee kuittausten vastaanottotahtia ja arvioi vapaan kaistan TCP-yhteydelle, minkä jälkeen se laskee optimaalisen ruuhkaikkunan koon kerran RTT:n aikana. Kun ikkunan pienennystä tarvitaan CW:n ilmoittaman ruuhkautumisen takia, TCP-Jersey asettaa tarvittavat muuttujat (*cwnd* ja *ssthresh*) optimaaliseen ikkunakokoon. ABE estimaattori on seuraavanlainen:

$$R_n = \frac{RTT \times R_{n-1} + L_n}{(t_n - t_{n-1}) + RTT} \tag{3}$$

missä R_n on arvioitu kaistanleveys n:nnen kuittauksen saapuessa ajanhetkellä t_n, t_{n-1} on edellisen kuittauksen saapumisaika, L_n on kuitattavan datan koko ja *RTT* on TCP:n arvio päästä-päähän RTT-viiveestä ajanhetkellä t_n. Optimaalinen ruuhkaikkunan koko segmentteinä lasketaan kaavalla

$$ownd_n = \frac{RTT \times R_n}{seg\_size} \tag{4}$$

missä *seg_size* on segmenttikoko.

CW on reitittimeen tehtävä konfiguraatio, jossa se varoittaa päätepisteitä merkitsemällä kaikki paketit, kun viitteitä orastavasta ruuhkautumisesta ilmenee. Pakettien merkitseminen auttaa TCP-lähettäjää erottamaan ruuhkautumisesta johtuvan pakettihukan langattomilla linkeillä tapahtuvasta pakettihukasta.

TCP-Jerseyn kehittäjien tekemien NS2-simulaatioiden perusteella Jersey tarjoaa paremman suorituskyvyn kuin myös langattomiin verkkoihin kehitetty TCP-Westwood, ja selvästi paremman suorituskyvyn kuin TCP-Reno.

2.1.7. ENIC

ENIC[11] eli *explicit notification with enhanced inter-layer communication and control* on luotettavaa tiedonsiirtoa varten kehitetty menetelmä. Sen keskeisenä perustana on protokollakerrosten välinen vuorovaikutus. Tiedonsiirron luotettavuutta pyritään parantamaan MAC-, reititys- ja TCP-kerrosten yhteistyön avulla. Menetelmässä on toteutettu myös päästä päähän huolto- ja korjaustoiminnallisuus TCP:lle.

TCP on valittu toteutuspohjaksi sen takia, että se tarjoaa luotettavan päästä päähän tiedonsiirron, minkä lisäksi se on yhteensopiva nykyisen TCP-pohjaisen Internetin kanssa. TCP kärsii kuitenkin heikosta suorituskyvystä erityisesti ad hoc –verkoissa. Tavanomainen TCP olettaa pakettihukan johtuvan aina verkon ruuhkautumisesta, mutta langattomissa verkoissa sen voi aiheuttaa useampikin syy, esimerkiksi pakettien katoaminen huonon kuuluvuuden tai häiriöiden takia.

Virhealttiit langattomat linkit voivat aiheuttaa satunnaisia bittivirheitä tai virhepurskeita, joiden takia saatetaan hukata useampiakin paketteja yhden lähetysikkunan sisällä. Yksittäiset pakettihukat aiheuttavat nopean uudelleenlähetyksen (fast retransmit) ja nopean toipumisen (fast recovery), joka pienentää ruuhkaikkunan kokoa. Useamman paketin katoaminen sen sijaan johtaa uudelleenlähetyksen aikavalvonnan laukeamiseen (retransmission timeout), joka käynnistää slow start –mekanismin. Slow startissa lähetysikkunaa kasvatetaan yhdellä jokaista vastaanotettua kuittausta kohti, eli eksponentiaalisesti, kunnes kuittaus paketista jää saamatta tai saavutetaan ennalta määritetty kynnysarvo. Langattomissa verkoissa on olemassa myös piilevän päätteen ongelma (hidden terminal problem), joka saattaa aiheuttaa signaalin vääristymistä. Piilevän päätteen ongelmasta kerrotaan enemmän seuraavalla sivulla.

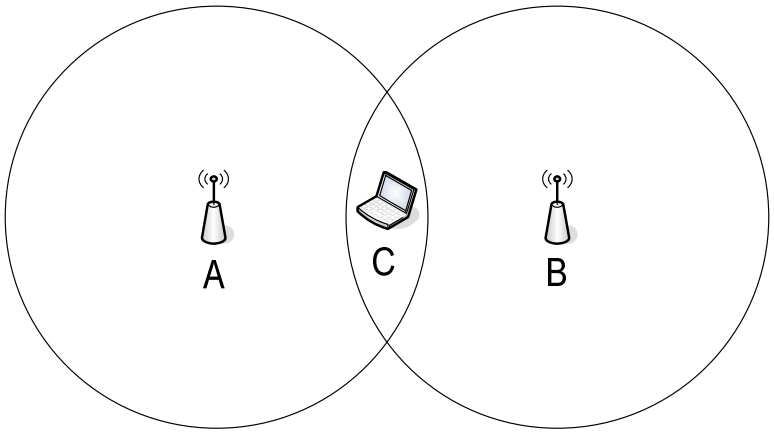
Reititysmuutoksista aiheutuu pakettihukkaa ja uudelleenjärjestymistä. Mobiilissa ympäristössä reittimuutoksia tapahtuu jatkuvasti ja reititysprotokollien on päivitettävä reittejään usein. Tämän vuoksi paketteja katoaa ja järjestyy uudelleen. Reitin katoaminen voi johtaa useiden pakettien katoamiseen tai uudelleenjärjestymiseen.

Ruuhkanhallintamekanismit ja uudelleenlähetyksen aikakatkaisu aiheuttavat suorituskyvyn heikkenemistä, koska tavanomainen TCP ei osaa erottaa pakettien katoamisten syitä.

Tavanomaiset protokollat langallisissa verkoissa käyttävät yleensä tiukkaa vaakasuuntaista kommunikointia protokollakerrosten välillä vierekkäisissä reitittimissä. Mobiiliverkkoihin tämä lähestymistapa ei sovellu sellaisenaan, koska TCP:n suorituskyvyn heikkenemiseen johtavat tapahtumat tapahtuvat useilla eri kerroksilla. Tehokkaan ja luotettavan päästä päähän tiedonsiirron

varmistamiseksi pitää ottaa huomioon eri kerrosten väliset vuorovaikutukset. Kerrosten välinen pystysuora kommunikointi saattaa auttaa ylempien kerrosten protokollia sitoutumaan lähemmin alemman kerroksen protokolliin. Tämän johdosta ylimääräisen hallintatiedon määrä vähentyy ja verkko kykenee reagoimaan nopeammin virhetilanteisiin ja reititysmuutoksiin. Tätä lähestymistapaa kirjoittajat kutsuvat nimellä *enhanced inter-layer control* (ENIC).

Perinteinen Carrier Sense Multiple Access (CSMA) menetelmä ei sovellu langattomiin verkkoihin piilevän päätteen ongelman takia, joten ENIC:n kehittäjät ehdottavat RTS-CTS –tyyppistä (Request-to-Send – Clear-to-Send) lähestymistapaa. Lähettävä pääte siis pyytää lähetyslupaa RTS-viestillä, johon vastaanottaja antaa lähetysluvan CTS-viestillä. CTS:n jälkeen lähetetään varsinainen data. Lopuksi vastaanottaja kuittaa lähetyksen ACK-viestillä. Samanlaista lähestymistapaa käytetään muun muassa WLAN:ssa.



Kuva 10 Piilevän päätteen ongelma

Piilevän päätteen ongelma on esitetty kuvassa 10. Ympyrät ovat solmujen lähetyssäteitä. C kuulee A:n ja B:n lähetyksen ja pystyy lähettämään niille, mutta A ei kuule B:tä eikä päinvastoin. Tästä aiheutuu se, että A saattaa lähettää samaan aikaan kuin B, jolloin signaalit sekoittuvat C:llä, jolloin kummankaan viestiä ei voida ottaa vastaan. RTS/CTS/DATA/ACK -toimintatavan avulla A tai B pystyy C:n lähettämistä viesteistä päättelemään lähettääkö joku muu sillä hetkellä. Esimerkiksi B lähettää ensin RTS-viestin. A ei kuule tätä, mutta se kuulee C:n lähettämän CTS-viestin, jolloin se tietää, että joku aikoo lähettää C:lle jotain. A ei myöskään kuule B:n lähettämää dataa, mutta se kuulee lopuksi C:n lähettämän ACK-viestin. ACKin jälkeen A voi pyytää omaa lähetyslupaa RTS-viestillä.

RTS/CTS/DATA/ACK –mekanismia käytetään myös reittivirheiden havaitsemiseen. Kun MAC-protokolla epäonnistuu lähettämään datakehyksen useista yrityksistä huolimatta, se ilmoittaa tästä

ylemmälle kerrokselle, joka aloittaa reitityksen korjaustoimenpiteet. Reititysvirheen tapahtuessa reititysprotokollat yrittävät ensin hätäkorjausta edelliseltä solmulta ylävirrasta. Jos tämä ei onnistu, reititysprotokolla ilmoittaa lähdesolmulle ja aloittaa päästä päähän reitinkorjaustoimenpiteet.

Jos virhe ei ole korjattavissa viereiseltä solmulta, tieto virhetilanteesta lähetetään kaikille lähde- ja kohdesolmuille. Tämän tiedon saadessaan solmut aloittavat TCP-moduulien operaatioiden koordinoinnin. Tässä tilanteessa TCP-prosessit ja niiden tilamuuttujat jäädytetään siksi aikaa, kunnes reitin korjaus on suoritettu. Korjauksen jälkeen TCP-prosesseja jatketaan lähettämällä kaikki kuittaamattomat paketit. RTO:n (retransmission timeout) arvo lasketaan uudestaan sen mukaan, muuttuuko reitin pituus. Jos käytettäisiin vanhaa RTO-arvoa, lyhyemmällä reitillä vanhan arvon käyttö aiheuttaisi turhaa tyhjäkäyntiä. Toisaalta pidemmällä reitillä vanhan arvon käyttäminen voi aiheuttaa turhia uudelleenlähetyksen aikakatkaisuja.

Tavanomaisen TCP:n positiivisen kumulatiivisen kuittauksen sijasta ENIC käyttää viivästettyä kuittausta (delayed ACK, DACK). Kohdesolmu lähettää kuittauksen vasta tietyn ajastimen umpeuduttua. Tämä mekanismi saattaa vähentää ACK-pakettien määrää ja helpottaa jaetun median kilpailutilannetta. Kadonneiden pakettien tapauksessa käytetään valinnaista kuittausta (selective ACK, SACK) kertomaan lähdesolmulle kadonneiden pakettien sekvenssinumerot tarkasti sen sijaan, että käytettäisiin tavanomaisia kumulatiivisia ACK-viestejä.

Menetelmää on testattu ns2-simulaatiolla liikkuvien solmujen välillä. Tuloksien perusteella menetelmällä saavutetaan paljon parempi tiedonsiirtonopeus kuin standardilla SACK:lla. Koska SACK TCP on tehokkaampi kuin tavanomainen TCP, voidaan ENIC:n päätellä tarjoavan parempaa suorituskykyä kuin mikään tavanomainen TCP-toteutus. Menetelmän suorituskyky heikkenee silti liikkuvuuden lisääntyessä. Tämä johtuu siitä, että satunnaiset bittivirheet langattomilla linkeillä aiheuttavat tarpeettomia ruuhkanhallintatoimenpiteitä. Kehittäjien mukaan ECN (Explicit Congestion Notification) vähentäisi selvästi suorituskyvyn huonontumista liikkuvuuden lisääntyessä.

2.1.8. ATP

ATP[12] (Ad hoc transport protocol) on kuljetustason protokolla, joka on suunniteltu korvaamaan TCP ad hoc –verkoissa. ATP:n kehittäjien mielestä useat TCP:n ominaisuudet ovat epäsopivia ad hoc –verkkoihin, minkä takia he esittelevät uuden protokollan.

ATP:ssa on useita ominaisuuksia, joista yksi tärkeimmistä on käyttää alempien protokollakerrosten informaatiota ja eksplisiittistä palautetta muilta solmuilta kuljetuskerroksen mekanismien apuna. Palautetta käytetään erityisesti yhteyden alussa verkon kapasiteetin arviointiin, ruuhkatilanteiden havaitsemiseen, välttämiseen ja hallintaan sekä polku- eli reititysvirheistä ilmoittamiseen. ATP ei tarvitse mitään vuokohtaisia tilatietoja välisolmuilla, joten se on skaalautuva.

ATP käyttää nopeuspohjaista lähetystä TCP:n ikkunapohjaisen lähetyksen sijasta. Tämä vähentää lähetysten purskeisuutta. Lisäksi lähetysten skedulointi tapahtuu lähettäjän ajastimen avulla, jolloin itsekellotus saapuvien ACKien perusteella eliminoituu.

TCP:ssä ruuhkanhallinta ja luotettavuusmekanismit on tiukasti yhdistetty riippumaan ACKien saapumisesta. ATP:ssa sitä vastoin nämä kaksi asiaa on erotettu toisistaan. Ruuhkanhallinta tapahtuu käyttämällä verkosta saatavaa palautetta. Luotettavuus on taattu karkeajakoisen vastaanottajapalautteen ja selektiivisten kuittausten avulla. Yhteyden välisolmut antavat ruuhkatietoa käytettävissä olevan nopeuden perusteella. Tämä tieto kuljetetaan hyötydatan mukana vastaanottajalle, joka kokoaa tämän tiedon perusteella yhteenvedon ja lähettää sen takaisin lähettäjälle. Luotettavuuden varmistamiseksi vastaanottaja lähettää selektiivisiä ACKeja, joilla se ilmoittaa lähettäjälle havaituista puuttuvista paketeista. Normaalissa TCP:ssä SACK-tieto täydentää kumulatiivisia kuittausviestejä, kun taas ATP luottaa pelkästään SACK tietoon.

Reitin keskellä olevat solmut auttavat ruuhkanhallintaa ylläpitämällä viivetietoja. Solmut ylläpitävät tietoa jonon pituudesta sekä lähetysviiveestä kyseisellä solmulla ja merkkaavat nämä tiedot paketteihin. Vastaanottaja saa suoraan tietoa verkon viiveestä ja ilmoittaa tiedon palautteena lähettäjälle. Lähettäjä päättää tiedon perusteella lisääkö tai vähentääkö se lähetysnopeutta, vai pitääkö se nopeuden kenties ennallaan. ATP:n nopeuden säilyttäminen on merkittävä ero normaalin TCP:n mahdollisista tiloista, joita ovat siis nopeuden kasvattaminen ja vähentäminen. Lisäksi nopeuden kasvattaminen ja vähentäminen tapahtuu saadun palautteen takia tarkemmin kuin TCP:ssä.

ATP:tä on testattu simuloimalla ns2-simulaation avulla. Tulosten perusteella ATP tarjoaa huomattavasti paremman suorituskyvyn kuin standardi TCP, TCP-ELFN ja ATCP. Lisäksi ATP saavuttaa paremman globaalin oikeudenmukaisuuden verkossa.

2.2. Yhteenveto

Edellä esiteltiin useampikin luotettavan tiedonsiirron toteuttava menetelmä. Menetelmien ja protokollien pelkästä määrästä ja eroista voidaan huomata, että luotettavan tiedonsiirron ongelmaan

ei ole vain yhtä ratkaisua. Osa menetelmistä on kehitetty selkeästi tietynlaisille sovelluksille, kun taas osa yrittää löytää yleisemmän tason ratkaisua. Tämän perusteella voidaan jo päätellä, että ongelmaan ei välttämättä ole vain yhtä universaalia ratkaisua, vaan ratkaisumenetelmiä tulee ehkä miettiä enemmän sovelluskohtaisesti.

3. Suunnittelulähtökohdat

Tässä luvussa käsitellään menetelmän suunnittelun pohjana käyttäviä suunnittelulähtökohtia. Näitä ovat luotettavuus, verkon rakenteen vaikutus, kapasiteetin tarkkailu, prosessointi, sekä varsinainen tiedon välitys. Lisäksi pohditaan aikaisemmasta tutkimuksesta poimittujen lupaavimpien menetelmien ominaisuuksien soveltuvuutta.

Menetelmiä vertaillaan eri metriikoiden suhteen, jonka jälkeen pohditaan parhaiten soveltuvien menetelmien ominaisuuksia.

Luvun lopussa kerrotaan hieman välitysprosesseista.

3.1. Ominaisuudet

3.1.1. Luotettavuus

Keskeisenä lähtökohtana on taata tiedonsiirron luotettavuus. Luotettavuus tarkoittaa sitä, että lähetetty tieto saapuu perille vastaanottajalle. Ei riitä, että vastaanottaja saa otettua vastaan jotain, vaan tarkoitus on saada koko lähetetty viesti perille. Viestin eheyden on siis säilyttävä. Jos vastaanottaja ei saa vastaanotettua koko informaatiota, kaikki vastaanotetut paketit ovat käytännössä turhaa liikennettä.

Ylempänä esitellyissä olemassa olevissa menetelmissä luotettavuuden varmistamiseen on käytetty monia erilaisia ratkaisuja. Joissain menetelmissä on käytössä hyppy hypyltä kuittaukset, kun jotkin taas luottavat päästä päähän menetelmiin. Joukossa on myös hieman eksoottisempiin menetelmiin perustuvia ratkaisuja.

3.1.2. Verkon rakenne

Verkon rakenteesta ei tehdä mitään ennakko-oletuksia, vaan menetelmän on sovelluttava kaikille linkeille niiden tyypistä riippumatta. Oli kyseessä sitten nopea kuitulinkki tai epävarma radiolinkki, menetelmän on varmistettava tiedonsiirron luotettavuus kaikissa olosuhteissa.

Koska linkkien kapasiteetit ja viiveet saattavat vaihdella suurestikin, menetelmän olisi syytä sisältää ominaisuus, jonka avulla se selvittää linkkien tyypin ja parametrit. Varsinaisesti kyseiset tiedot saataisiin luultavasti reititysprotokollalta, mutta menetelmän tarkoitus on olla mahdollisimman erillään reititysprotokollista. Toisaalta aiemmin esitellyssä ENIC:ssä[11] kommunikaatiota tapahtuu paitsi kerrosten kesken protokollapinon suhteen vaakasuunnassa vierekkäisillä reitittimillä, myös pystysuunnassa saman reitittimen protokollapinossa. Ratkaisusta on se hyöty, että mahdollisista linkkien katkeamisista tai vastaavista tilanteista saadaan siirtoyhteystason protokollalta nopeasti tieto, jolloin voidaan aloittaa korjaustoimenpiteiden koordinointi. Ihannetilanteessa voitaisiin toimia eri protokollakerrosten kesken, mutta samalla olla riippumattomia käytettävistä reititys- ja siirtoyhteysprotokollista. Käytännössä lienee kuitenkin tarpeen toteuttaa jonkinlainen rajapinta, jota voi konfiguroida käytettävien protokollien mukaan.

HRS:ssä käytetään hyväksi hyppykohtaista sekvenssiä lähetyksen perillemenon varmistamiseksi. Verkon solmut ylläpitävät kaikille naapurisolmuilleen sekvenssinumeroa, joka on juokseva kokonaisluku. Tämä luku leimataan kaikkiin kyseiselle naapurille meneviin paketteihin, jolloin voidaan helposti huomata, jos välistä puuttuu paketti. Tämä lähestymistapa soveltuu sekä kiinteille linkeille että radiolinkeille. Välissä olevien solmujen ei tarvitse ylläpitää mitään tilatietoa välittämästään liikenteestä, vaan ainoa tarvittava tieto on sekvenssinumero. Solmulla on välittömiä naapureita vain suhteellisen rajallinen määrä, jolloin ylläpidettävien sekvenssien määrä ei kasva äärettömästi. Tämä lupaa skaalautuvuutta. Sekvenssinumeron leimaaminen pakettiin on suhteellisen helppo operaatio, joten sen ei pitäisi myöskään vaatia kohtuutonta prosessointitehoa verkon solmuilta.

3.1.3. Verkon kapasiteetti ja tarkkailu

Verkon kapasiteetin ei voida olettaa olevan vakio millään aikavälillä, joten jatkuva kapasiteetin mittaus on tärkeä osa tiedonsiirron varmistamista. Kapasiteettia mittaamalla saadaan verkon koko käytettävissä oleva kapasiteetti aina käyttöön. Toisaalta ruuhka- tai häiriötilanteessa rajoittunutta kapasiteettia ei ajeta yli liian suurella tiedonsiirtomäärällä.

ATP:ssa[12] välisolmut ylläpitävät viivetietoja, jotka ne raportoivat vastaanottavalle solmulle. Vastaanottaja aggregoi nämä tiedot yhteen ja informoi lähettäjää verkon viive- ja kapasiteettitilanteesta. Lähettäjä säätää lähetysnopeuttaan saatujen raporttien perusteella.

TCP-Jerseyssä[8] on myös mekanismi käytettävissä olevan kaistanleveyden määrittämiseen. Siinä lähettäjä pystyy saapuvien kuittausten viiveen ja lähetettyjen pakettien koon perusteella laskemaan tarkan arvion kaistanleveydestä tietyllä hetkellä. Tässä menetelmässä tarvitsee tarkastella myös viiveen vaihteluja, jotta arvio käytettävästä kaistanleveydestä pysyy tarpeeksi tarkkana.

3.1.4. Prosessointi

Menetelmän vaatimien operaatioiden suorittaminen voidaan hoitaa kaikissa laitteissa itse tai vaihtoehtoisesti voidaan käyttää keskitettyä ratkaisua. Jos verkon laitteissa on rajalliset laskentaominaisuudet tai laitteet ovat akkukäyttöisiä, voi olla kannattavaa minimoida laskentakuormaa. Keskitetty laskenta tosin lisää hallintaliikennettä huomattavasti, mutta kuten FBcastin[6] kehittäjät mainitsevat, laskenta vie kuitenkin huomattavasti vähemmän tehoa kuin radiolähetys. Aiemmin mainittu hyppykohtainen kuittaus lisää laskentakuormaa laitteilla ainakin vähän, mutta niin kauan kuin laitteet pystyvät hoitamaan lisätoimenpiteet tiedonsiirto-operaatioiden ohessa, ei ongelmia pitäisi syntyä.

3.1.5. Tiedonvälitys vs. pakettien välitys

Tiedonvälitys ja varsinainen pakettien välitys ovat kaksi eri asiaa, joiden toteuttamisessa voi olla eroa. Pakettien välitys tarkoittaa, että pyritään välittämään kaikki paketit, ja siten myös niiden mukana kuljettama tieto. Tiedonlähetys tulee varmistetuksi siis siten, että kaikki paketit välitetään perille. FBcastissa[6] kuitenkin esitellään menetelmä, jossa kaikkia yksittäisiä paketteja ei tarvitse siirtää perille, jotta varsinainen tieto saadaan siirrettyä. FBcast perustuu suihkulähdekoodeihin, joiden avulla alkuperäinen viesti voidaan rakentaa uudelleen kun koodatusta datasta on saatu kerättyä kasaan tarpeeksi. Painotusta tiedonvälityksen ja paketinvälityksen välillä olisi syytä voida muuttaa tilanteen mukaan. Esimerkiksi radioympäristön ollessa huono, voi suihkulähdekoodaus olla järkevä vaihtoehto. Toisaalta hyvän kuuluvuuden vallitessa kannattaa lähettää paketit sellaisenaan, jolloin suihkulähdekoodauksen tuottamaa ylimääräistä tietoa ei tarvitse lähettää.

Jos linkkien tila on epäluotettava, monitiereititys voi olla varteenotettava vaihtoehto luotettavan lähetyksen varmistamiseksi. Monitiereitityksessä kohdekoneille ylläpidetään useita reittejä ja paketit välitetään eri reittien välillä kuormituksen tasaamiseksi (load balancing). Yksi mahdollisuus

olisi lähettää data kokonaisuudessaan useampaa reittiä pitkin, jolloin ei tarvitse olla yhden reitin varassa. Vastaanottaja aggregoi saapuvan liikenteen yhdeksi kokonaisuudeksi, jolloin moninkertaiset paketit suodattuvat lopulta pois.

Radioympäristön ollessa todella huono voi välivarastointi tulla kysymykseen. Esimerkiksi PSFQ toimii korkean virhemäärän vallitessa juuri siten, että se varastoi paketteja niin kauan, kunnes ne saadaan lähetetyksi. Välivarastointi on tuttu myös sähköpostin välittämisessä toimivan SMTP:n toiminnasta. Kun SMTP-palvelin vastaanottaa sähköpostiviestin, se tallentaa sen muistiinsa. Sen jälkeen se säilyttää sitä muistissaan niin kauan, kunnes se saa seuraavalta SMTP-palvelimelta kuittauksen viestin onnistuneesta vastaanotosta.

3.2. Menetelmien vertailu

Seuraavassa eri menetelmiä on vertailtu tiettyjen verkon metriikoiden suhteen. Näiden perusteella saadaan parempi yleiskuva siitä, millaisiin tilanteisiin eri menetelmät soveltuvat. Tyhjät ruudut tarkoittavat sitä, että kyseisestä metriikasta on vaikea tehdä johtopäätöksiä ilman menetelmien laajempaa testausta, joka on tämän työn rajauksen ulkopuolella.

Vertailuun ei ole valittu kaikkia esiteltyjä menetelmiä, koska erityisesti langattomiin sensoriverkkoihin tarkoitettujen menetelmien hyvät puolet kulminoituvat HRS:ään.

Taulukossa 1 on riveillä eritelty eri metriikat ja sarakkeissa on eri menetelmät. Plussa tarkoittaa sitä, että menetelmä on kyseisen metriikan suhteen hyvä. Miinus taas tarkoittaa, että kyseisen metriikan suhteen menetelmässä olisi parantamisen varaa.

Taulukko 1 Menetelmien vertailua eri metriikoiden suhteen

Metriikka	HRS	ENIC	Fbcast	ATP	TCP-Jersey
Skaalautuvuus	++	+	-	+	+
Kapasiteetti		+	-	++	+
Luotettavuus	+		+		
Reaaliaikaisuus	-		-		
Prosessointivaativuus			---		+
Muistivaativuus	+	+	-	+	+
Dynaamisuus	+		+		
Itsekonfiguroituvuus	+		+		-
riippumattomuus (reitityksestä)		-	+	-	+
riippumattomuus (päätepisteistä)	+	--	-	--	--
riippumattomuus (välipisteistä)	--	-	-	--	+
hallintatiedon määrä	+	-	+	-	+

Skaalautuvuus tarkoittaa sitä, kuinka hyvin menetelmä toimii, kun toimintaympäristöä (eli käytettävää verkkoa) laajennetaan. Jos verkon laajentaminen ei aiheuta menetelmässä merkittävää toimenpiteiden lisääntymistä tai kasvata menetelmän muisti- tai prosessointivaatimuksia, voidaan sen sanoa olevan skaalautuva.

Kapasiteetti tarkoittaa tässä yhteydessä sitä, kuinka hyvin koko verkon käytettävissä oleva kapasiteetti hyödynnetään.

Luotettavuus kuvaa lähinnä menetelmän toimintavarmuutta. Kaikki menetelmät toteuttavat suunnitellussa ympäristössään luotettavan tiedonsiirron.

Reaaliaikaisuus tarkoittaa sitä, miten nopeasti lähetetty tieto on käytettävissä vastaanottajalla. Menetelmiä ei ole tarkoitettu varsinaiseen reaaliaikaiseen tiedonsiirtoon, mutta tavoite olisi, että tieto siirtyy kohtuullisessa ajassa vastaanottajalle.

Prosessointivaativuus kertoo, kuinka paljon menetelmä vaatii laskentatehoa verkon laitteilta. Muistivaativuus kertoo tarvittavan muistin määrän verkon laitteilla.

Dynaamisuus tarkoittaa menetelmän kykyä sopeutua nopeasti verkossa tapahtuviin muutoksiin. Muutoksia voivat olla esimerkiksi reititysmuutokset tai linkkien katkeamiset.

Itsekonfiguroituvuus kuvaa sitä, miten hyvin menetelmä säätelee itse itseään vallitsevien olosuhteiden mukaan. Toisin sanoen osaako se sopeutua verkon erilaisiin muutoksiin

automaattisesti muuttamalla tarvittavia parametreja. Itsekonfiguroituvuus ja dynaamisuus ovat hyvin lähellä toisiaan, mutta eivät ole täysin sama asia.

Riippumattomuus reitityksestä tarkoittaa sitä, että menetelmä ei tarvitse tietoa alla toimivasta reititysprotokollasta. Riippumattomuus päätepisteistä tarkoittaa sitä, että päätepisteiden ei tarvitse aktiivisesti lähettää tietoja lähetyksen tilasta, viiveistä tai kapasiteetista. Tietysti päätepisteet ovat aktiivisia toimijoita siinä mielessä, että varsinainen data kuitenkin liikkuu niiden välillä. Riippumattomuus välipisteistä tarkoittaa sitä, että päätepisteiden välillä oleville reitittimille tai solmuille ei tarvitse tehdä mitään erityisiä toimenpiteitä menetelmän toimimiseksi.

Hallintatiedon määrä kuvaa sitä, kuinka paljon tiedonsiirto aiheuttaa ylimääräistä liikennettä verkkoon. Esimerkiksi erilaiset viive- tai kapasiteettiraportit ovat hallintatietoa.

3.2.1. Johtopäätökset

Tarkastelemalla taulukkoa, HRS erottuu selkeästi edukseen muista menetelmistä. Se tosin on suunnittelulähtökohdiltaankin lähimpänä haluttua lopputulosta menetelmän suhteen. Myös FBcastilla on hyviä ominaisuuksia, joista voi olla apua menetelmän toteuttamisessa.

ENIC, ATP, ja TCP-Jersey ovat käytännössä päästä päähän -menetelmiä, jolloin tiedonsiirto riippuu liikaa päätepisteiden välisestä kommunikaatiosta, joka tuottaa viiveitä ja ylimääräistä hallintatietoa. Tavoitteena olisi saada menetelmä mahdollisimman riippumattomaksi päätepisteistä, jotta mahdollisesti häiriöllisessä ympäristössäkin voidaan toimia luotettavasti ilman jatkuvaa päätepisteiden välistä kommunikointia. Tämän vuoksi edellä mainittuja menetelmiä ei oteta lähtökohdiksi lopulliseen ratkaisuun.

3.3. Välitysprosessit

Eri laitteistoissa ja ohjelmistoissa on erilaiset välitysprosessit paketeille. Suunniteltavan ratkaisun tulee olla soveltuva erilaisiin ympäristöihin ja laitteisto- tai ohjelmistoratkaisuihin. Tässä osiossa tarkastellaan välitysprosesseja varsinaisissa reititinlaitteissa sekä Unix-käyttöjärjestelmässä. Tavoitteena on sovittaa ratkaisussa tarvittavat menetelmät kyseisiin prosesseihin mahdollisimman saumattomasti.

Koska menetelmän keskeiseksi ominaisuudeksi on valittu hyppykohtaiset kuittaukset, on pakettien hyppy hypyltä sekvenssinumeroita pystyttävä tarkastelemaan ja manipuloimaan pakettia välittävässä laitteessa. Reititysmuutoksilla on myös vaikutus sekvenssien luonteeseen siten, että

uuden naapurin ilmaantuessa reititystauluun sille on luotava uusi sekvenssi, jota käytetään kommunikoitaessa kyseisen naapurin kanssa. Toisaalta naapureiden kadotessa reititystaulusta, niiden sekvenssit voidaan nollata ja poistaa, jotta entisistä naapureista ei jää muistiin turhaa tietoa. Ratkaisun täytyy siis manipuloida välitysprosessissa olevia paketteja sekä samalla tarkkailla reititysprosessissa tapahtuvia muutoksia.

3.3.1. Välitysprosessi reitittimessä

Reitittimen välitysprosessi on tässä yritetty kuvailla mahdollisimman yleisessä tapauksessa. Eri valmistajilla on omat menetelmänsä välityksen toteuttamiseksi jopa eri reititinmallien välillä, joten kattava laitekohtainen tarkastelu jätetään tekemättä. Prosessin kuvauksen pohjana on käytetty Juniperin M5 ja M10 –reitittimien välitysprosessia.

Välitysprosessi alkaa siitä, kun paketti saapuu reitittimen PIC:lle (physical interface card). PIC:llä paketille tehdään layer 2 -prosessointi, jossa tarkastetaan paketin eheys. Tämän jälkeen PIC välittää paketin niin kutsutun midplanen läpi varsinaiselle prosessorikortille, jossa I/O-hallintapiiri paloittelee paketin ja puskurinhallintapiiri puskuroi paketin reitittimen muistiin. Paketin otsikko välitetään välitysprosessorille, joka etsii kohteen reititystaulusta ja tekee välityspäätöksen. Tämän jälkeen välitysprosessori ilmoittaa puskurinhallintapiirille välityspäätöksestä, ja sanoma välitetään lähettävälle PIC:lle. I/O-hallintapiiri kokoaa paketin muistista ja välittää sen lähettävälle PIC:lle, joka lähettää paketin seuraavalle laitteelle.

3.3.2. Välitysprosessi Unixissa

Unixin välitysprosessi on periaatetasolla käytännössä samanlainen kuin edellä kuvattu reitittimen prosessi, mutta koska se tehdään ohjelmistotasolla laitteistotason sijaan, on prosessin yksityiskohdat hieman erilaisia. Tässä on käsitelty FreeBSD 5.2:n välitysprosessia [13].

Välitysprosessi alkaa, kun paketti saapuu verkkorajapintaan ja se välitetään input-funktiolle ip_input(). Paketti voidaan käsitellä neljällä eri tavalla: i) paketti annetaan suoraan syötteenä ylemmän tason protokollalle, ii) paketti kohtaa virhetilan joka raportoidaan lähteelle, iii) paketti pudotetaan virheen johdosta, tai iv) se välitetään seuraavalle hypylle matkalla kohteeseensa. Tässä yhteydessä keskitymme viimeiseen vaihtoehtoon. Pääpiirteissään paketin käsittely sen tullessa sisään (kuvassa ip_input) on seuraavanlainen:

1. Tarkistetaan, että paketti on vähintään IP-otsikon pituinen ja varmistetaan, että otsikko on katkeamaton.

2. Lasketaan otsikon tarkistussumma (checksum) ja hylätään virheellinen paketti.

3. Tarkistetaan, että paketti on vähintään yhtä pitkä kuin otsikko ilmoittaa, muutoin tiputetaan paketti. Siistitään täyte (padding) paketin lopusta.

4. Suoritetaan mahdolliset suodatukset tai turvallisuusfunktiot, joita ipfw tai IPSec vaatii.

5. Käsitellään mahdolliset optiot otsikossa.

6. Tarkastetaan onko paketti tarkoitettu kyseiselle isäntäkoneelle. Jos on, jatketaan paketin käsittelyä. Jos ei ole, yritetään välittää paketti eteenpäin välitysprosessille, mikäli reititystoiminto on kytketty päälle. Muutoin paketti pudotetaan.

7. Jos paketti on paloiteltu (fragmented), se säilytetään kunnes kaikki palat on vastaanotettu ja uudelleenkoottu tai kunnes se on liian vanha säilytettäväksi.

8. Annetaan paketti seuraavan kerroksen protokollalle syötteenä.

Varsinainen välitysprosessi (kuvassa ip_forward) suorittaa seuraavat askeleet paketille, joka on menossa eri koneelle:

9. Tarkistetaan, että paketinvälitys on päällä. Pudotetaan paketti, jos ei ole.

10. Tarkistetaan, että kohdeosoite on sellainen, johon voidaan välittää paketteja. Esimerkiksi verkkoon 0, verkkoon 127 tai laittomaan osoitteeseen kohdistettuja paketteja ei välitetä.

11. Tallennetaan enintään 64 tavua vastaanotetusta viestistä, jotta voidaan luoda lähettäjälle virheviesti tarvittaessa.

12. Päätetään välitykseen käytettävä reitti.

13. Jos reitti ulos käyttää samaa rajapintaa kuin mistä paketti saapui ja jos lähettävä kone on samassa verkossa, lähetetään ICMP redirect –viesti lähettävälle koneelle.

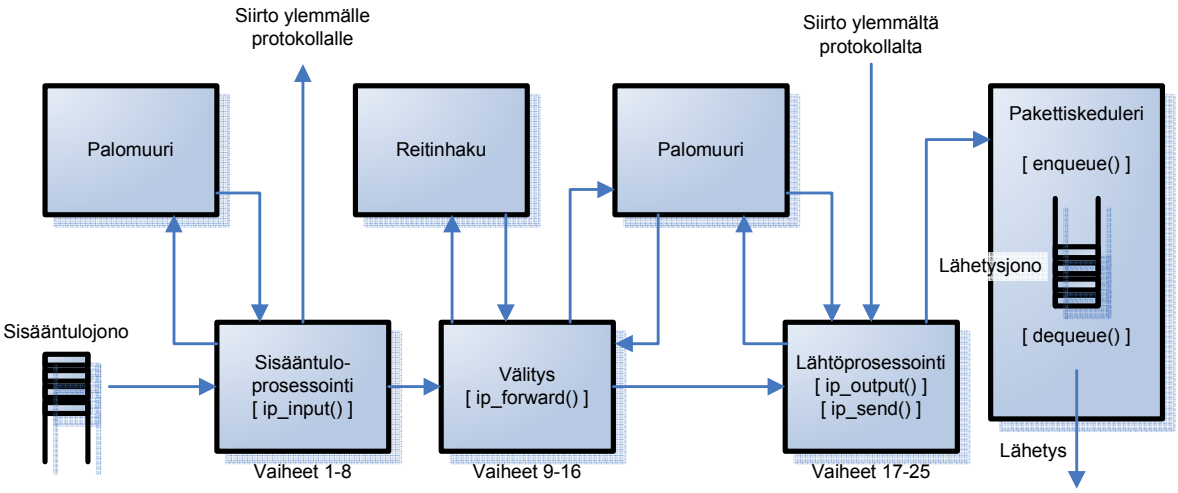
14. Käsitellään tarvittavat IPSec-päivitykset paketin otsikkoon.

15. Kutsutaan ip_output()-funktiota lähettämään paketti kohteeseensa tai seuraavaan yhdyskäytävään (gateway).

16. Jos havaitaan virhe, lähetetään ICMP-viesti lähdekoneelle.

Lähetettäessä pakettia (kuvassa ip_output) koneelta suoritetaan alla mainitut toimenpiteet. Välitettäessä tietoja kaikkia näitä vaiheita ei käydä läpi, vaan nämä ovat suurimmalta osin lähdekoneen suorittamia toimenpiteitä.

17. Lisätään mahdolliset IP optiot.
18. Täytetään puuttuvat otsikkokentät (IP versio, otsikon pituus, jne.)
19. Päätetään reitti (eli käytettävät rajapinta ja seuraavan hypyn osoite).
20. Tarkistetaan, onko kohde multicast-osoite. Jos on, määritetään lähtörajapinta ja hyppyjen määrä.
21. Tarkistetaan, onko kohde broadcast-osoite. Jos on, tarkistetaan onko broadcast sallittu.
22. Tehdään tarvittavat IPSec-käsittelyt paketille, kuten esimerkiksi kryptaus.
23. Tarkistetaan, muuttavatko suodatussäännöt pakettia tai estävät sen lähetyksen.
24. Jos paketti on pienempi kuin lähtörajapinnan maksimipakettikoko (MTU), lasketaan tarkistussumma ja kutsutaan rajapinnan lähetysfunktiota.
25. Jos paketti on suurempi kuin MTU, paketti hajotetaan paloiksi, jotka lähetetään yksitellen.



Kuva 11 Välitysprosessi unixissa

3.3.3. Johtopäätökset

Ratkaisun vaatimat toimenpiteet tulee sijoittaa yllä käsiteltyyn Unixin välitysprosessiin mahdollisimman saumattomasti. Loogisin vaihtoehto manipulointikohdaksi on sisään tulevan paketin käsittelyn vaihe 4, jossa suoritetaan paketille mahdolliset suodatukset. Tässä kohtaa otsikosta löytyvät sekvenssinumerot voidaan prosessoida ja suorittaa vaadittavat toimenpiteet (sekvenssin jatkuvuuden tarkistus, kuittausviestin luominen). Toisaalta vaiheessa 5 tarkastetaan otsikon optiot, joihin voidaan myös sisällyttää tietoja sekvensseistä tai toisista otsikoista. Kummassa vaiheessa mahdollinen käsittely tehdään, riippuu siis täysin valitusta toteutustavasta.

Paketin pitäisi mennä läpi välitysfunktioista ongelmitta, joten siihen ei välttämättä tarvitse kajota ollenkaan. Ratkaisun tavoitteena on olla mahdollisimman riippumaton reitityksestä ja näin ollen myös riippumattomuus välitysprosessista on toivottava ominaisuus.

Lähetysprosessissa voidaan lisätä otsikkoon optioita vaiheessa 17. Kuten edellä mainittiin, optioiden käyttö on yksi mahdollisuus toiminnallisuuden toteuttamiseksi. Toisaalta, jos halutaan käyttää pakettien suodattamisen avulla tapahtuvaa manipulointia, se voidaan suorittaa vaiheessa 23, jossa käsitellään suodatussääntöjä. Muutoin vaiheet voidaan käydä muuttumattomina läpi. Jos otsikkoon kajotaan, myös tarkistussumma muuttuu, joten manipulaatio on tehtävä ennen tarkistussumman laskemista.

TCP:n tarkistussumma lasketaan TCP-otsikon, hyötydatan sekä niin sanotun pseudo-otsikon yli, johon kuuluu IP-otsikosta lähde- ja kohdeosoitteet, protokolla sekä TCP-paketin pituus. Näihin tietoihin IP-otsikosta ei siis voi kajota, jos halutaan TCP:n tarkistussummien täsmäävän.

Mielenkiintoinen yksityiskohta on kuittausviestien käsittely. Ylemmillä protokollilla ei ole mitään tietoa käytetystä hyppykohtaisista kuittausmenettelyistä, joten ne eivät voi eikä niiden tarvitsekaan puuttua kuittaukseen. Toisaalta oletusarvoisesti tietylle koneelle tarkoitettu paketti ohjataan ylempien protokollien käsiteltäväksi. Kuittauspaketit on siis suodatettava ja ohjattava pois normaalista ketjusta, jotta ne voidaan käsitellä. Käsittelyn jälkeen seuraavan hypyn vastaanottamien pakettien käyttämä muisti voidaan vapauttaa.

4. Luotettava tiedonsiirtomenetelmä

Aiemmin käsiteltyjen vertailujen pohjalta ratkaisuksi on valittu HRS:n[5] kaltainen hyppykohtaisiin kuittauksiin perustuva menetelmä. Menetelmää ei toteuteta käytännön tasolla, vaan se mallinnetaan ja simuloidaan OPNET Modeler –ohjelmalla. Menetelmän nimeksi on valittu HBH (sanoista hop-by-hop).

Tässä osassa kerrotaan ensin Modelerista, jonka jälkeen käydään läpi ratkaisun implementointi ja simulointiin liittyvät yksityiskohdat. Lopuksi kerrotaan vielä eri simulointiskenaarioista.

4.1. Toteutus

Tässä osassa esitellään menetelmän toteutukseen liittyvät yksityiskohdat. Aluksi esitellään toteutukseen ja mallinnukseen käytetty työkalu, eli OPNET Modeler. Tämän jälkeen paneudutaan tarkemmin toteutuksen eri osa-alueisiin.

4.1.1. OPNET Modeler

Modeler on OPNETin kehittämä ohjelmisto, jolla tietoverkkojen mallintaminen ja simulointi onnistuvat suhteellisen tehokkaasti. Useat laitevalmistajat tukevat Modeleria, joten sen laitekirjastoista löytyy lukuisten valmistajien laitteiden malleja suoraan. Tämän työn puitteissa on kuitenkin tarkoitus käyttää mahdollisimman geneeristä eli yleispätevää laitemallia, joka ei riipu laitevalmistajista.

Ohjelma perustuu diskreettiin tapahtumasimulointiin. Tämä tarkoittaa sitä, että mallinnettavassa verkossa olevat laitteet tuottavat tapahtumia vain tietyillä ajanhetkillä. Näitä tapahtumia ovat esimerkiksi paketin luonti, paketin välitys, reittimuutos, paketin tuhoaminen, ja niin edelleen. Tapahtumat ja niiden luonne riippuvat mallinnettavasta laitteesta. Luotujen tapahtumien ketju mallintaa verkon toimintaa oikeassa maailmassa. Yhdistelemällä yksittäiseen objektiin (paketti,

laite) liittyviä tapahtumia, voidaan määrittää verkon kannalta oleellisia tunnuslukuja. Esimerkiksi linkkikohtaisesti voidaan tarkastella kuormitusta tai läpäisyä, päätelaitteilta tiedonsiirtonopeuksia ja reitittimistä jonotusviiveitä. Suureita voidaan tarkastella myös koko verkon kattavalla globaalilla tasolla.

Sen lisäksi, että ohjelmalla voi rakentaa ja simuloida virtuaalisia verkkoja valmiista laitteista, myös laitteiden prosesseja voidaan muokata jopa yksittäisten funktioiden tasolla. Tarvittaessa uusia laitteita voidaan luoda tyhjästä. Tämän työn tavoitteena on muokata olemassa olevaa mallia, toteuttaa HBH:n toiminnallisuus, sekä valita oleelliset tunnusluvut, joilla menetelmää analysoidaan.

4.1.2. Hyppykohtainen kuittaus

Menetelmän pohjaksi on valittu hyppykohtainen kuittaus (hop-by-hop acknowledgement). Yksinkertaistettuna verkon solmut ylläpitävät sekvenssejä naapurisolmuilleen, ja kuittaukset tapahtuvat jokaisella hypyllä tämän hyppykohtaisen sekvenssin perusteella.

Kuittauksia ei tehdä jokaiselle paketille erikseen, vaan kuittaaminen perustuu negatiivisiin kuittauksiin. Tämä tarkoittaa sitä, että niin kauan kun paketit virtaavat perille yhtenäisenä sekvenssinä, mitään ei tarvitse tehdä. Siinä vaiheessa kun sekvenssistä löytyy aukko, eli välistä puuttuu yksi tai useampi sekvenssinumero, lähetetään negatiivinen kuittaus (NACK, negative acknowledgement). NACKissa on tieto siitä, mikä paketti tai mitkä paketit välistä puuttuvat. Tämän tiedon perusteella lähde osaa lähettää oikeat paketit uudestaan.

NACK-kuittaus ei ole täysin luotettavaa, varsinkaan jos sekvenssin viimeinen paketti katoaa. Ilman seuraavan paketin saapumista ei voida tietää, onko sekvenssissä aukkoja. Tämän puutteen kiertämiseksi menetelmään on toteutettu hACK-toiminnallisuus (hop-by-hop ACK). Jos kohteeseen ei viimeisimmän paketin jälkeen ole tietyn ajan kuluessa tullut uusia paketteja, lähetetään lähettäjälle hACK-viesti. Viestin perusteella lähettäjä tietää, että viimeinenkin paketti on mennyt perille. Vastaavasti jos lähettäjä ei saa viimeiseen viestiin kuittausta kohteelta, lähetetään viimeinen paketti uudestaan niin kauan, kunnes se kuitataan.

Yksinkertaisuuden vuoksi mallissa ei oteta huomioon reititystä eikä reititysprotokollia. Käytännön toteutuksessa reititysmuutokset olisi otettava huomioon, koska niiden perusteella luodaan uusia sekvenssejä ja mahdollisesti myös poistetaan pitkän aikaa käyttämättömänä olleita sekvenssejä (solmu voi olla esimerkiksi kadonnut verkosta).

Kyseessä on niin sanottu Ad hoc –verkko, jossa jokaisen naapurin oletetaan olevan kuuluvuusalueella. Myöskään verkkokerroksen yläpuolisesta toiminnallisuudesta ei olla kiinnostuneita. Jos paketit välittyvät verkossa luotettavasti, ne todennäköisesti välittyvät myös asianomaisille sovelluksille luotettavasti.

Toinen yksinkertaistus simuloinnissa on tehty päästä päähän yhteyksien kanssa. Mitään varsinaisia päästä päähän sekvenssejä ei pidetä yllä, koska painopiste on nimenomaan hyppykohtaisen luotettavuuden varmistamisessa. Jos välissä olevat hypyt ovat luotettavia eli eivät aiheuta häviöitä, voidaan myös päästä päähän yhteyksien olettaa olevan häviöttömiä.

Hyppykohtaisen kuittauksen kanssa pohdittavaksi jää se, millä tasolla kuittaus tehdään. Kuittaus voidaan tehdä pakettivirran tasolla, eli ylläpidetään vain yhtä sekvenssiä kaikille linkin ylittäville paketeille. Tässä on ongelmana se, että jos yhdestä vuosta katoaa paketteja, joudutaan kaikkien voiden välitys jäädyttämään, kunnes sekvenssi on paikattu. Jos käytössä on useita voita, tästä aiheutuu huomattava haitta linkin läpäisykyvylle.

Toisaalta jos ylläpidetään vuokohtaisia sekvenssejä, ylimääräisen hallintatiedon määrä kasvaa, sillä yhden linkin yli voidaan välittää satoja voita. Näille kaikille voille olisi ylläpidettävä omaa sekvenssiään. Kompromissi täytyykin tehdä sen suhteen, että halutaanko yksinkertaisempi ja kapasiteettirajoitteisempi järjestelmä pienellä hallintatiedon määrällä vai onko mahdollista panostaa vuokohtaisiin sekvensseihin, jolloin pienellä hallintatiedon määrän lisäämisellä saadaan aikaan huomattava tehokkuuden kasvu.

Eräs vaihtoehto on jakaa vuot muutamaan ryhmään, esimerkiksi osoitteen alkuosan perusteella, jolloin ylläpidettävien sekvenssien määrä ei kasvaisi liian suureksi. Tässä tapauksessa eri ryhmässä olevien voiden pakettihukat eivät vaikuttaisi muiden ryhmien tiedonsiirtoihin millään tavalla, jolloin tehokkuus pysyisi hyvänä satunnaisista pakettihukista huolimatta.

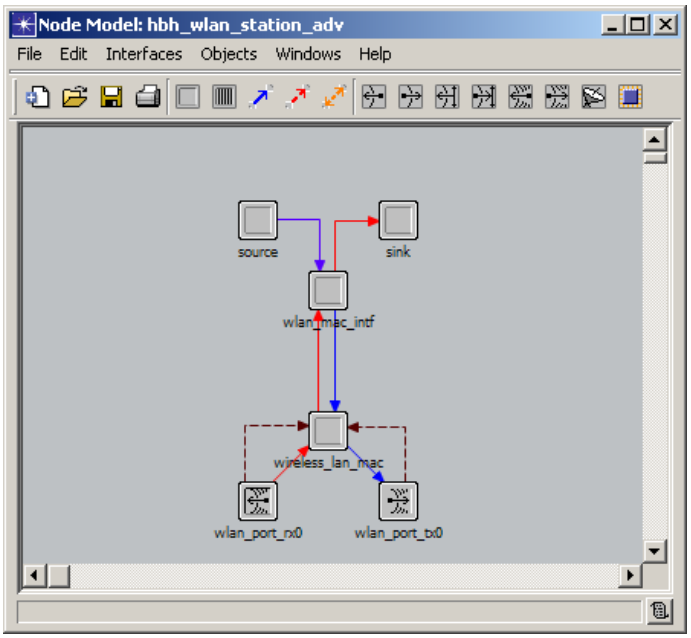
Tässä simulaatiossa tarkastellaan pientä pakettimäärää ja käytännössä siis yhtä vuota, joten painopiste on siis enemmänkin siinä, miten menetelmä käyttäytyy vuokohtaisesti yhden vuon kohdalla.

Eräs ongelmakohta liittyy siihen, missä kohtaa välitysprosessia sekvenssinumeroiden kuittaus ja leimaus tehdään. Tämän simuloinnin puitteissa sekvenssien leimaus tehdään MAC-tasolla, jolloin mistään ylemmän tason voista ei ole mitään käsitystä. Menetelmä on robusti, mutta vuotiedon puuttumisen takia törmätään yllä kuvattuun ongelmaan. Eli yhden vuon pakettien kadotessa myös muut vuot joutuvat odottamaan.

MAC-tason kuittauksella saadaan linkillä tapahtuvat pakettihukat havaittua ja korjattua, mutta itse pakettia välittävän laitteen sisällä tapahtuvat pakettihukat saattavat jäädä huomaamatta. Näitä pakettihukkia voi syntyä esimerkiksi palomuurissa tai suodatuksessa, jos paketti ei syystä tai toisesta selviä välitysprosessin läpi. Käytännön tasolla on siis tehtävä valinta vuotason ja linkkitason kuittauksen välillä. Lisäksi jollakin tavalla on huolehdittava siitä, että välittävässä laitteessa tapahtuvat pakettihukat tulevat huomatuiksi. Tämän mallinnuksen puitteissa kyseistä toiminnallisuutta ei ole toteutettu pääosin siksi, että valittu malli on melko yksinkertainen. Se ei tarjoa välitystoimintoja, vaan kaikki solmut ovat ainoastaan pakettien lähettäjiä ja vastaanottajia. Mallista on kerrottu seuraavassa kappaleessa.

4.1.3. WLAN-työasema –malli

Mallinnuksen pohjaksi löytyi Modelerista valmis yksinkertaisen WLAN-tukiaseman malli. Malli on kuvattu kuvassa 12.



Kuva 12 WLAN-työaseman toimintamalli

Malli toimii siten, että purskeinen lähde (bursty source, kuvassa ”source”) luo satunnaisin väliajoin paketteja. Lähteen tavoite on luoda lyhyitä usean paketin lähetyspurskeita. Lähde on ON/OFF –tyyppinen, eli lähde luo paketteja jonkin aikaa, ja jonka jälkeen on passiivinen jakso. Jaksojen pituudet arvotaan eksponentiaalijakaumasta, jonka keskiarvo ON-jaksoille on 10 sekuntia ja OFF-jaksoille 50 sekuntia. Pakettien koot ja saapumisvälit ovat myös eksponentiaalijakautuneita keskiarvoilla 1024 tavua paketin koolle ja 1,0 sekuntia saapumisvälille. Paketeille ei tässä vaiheessa

tehdä muuta kuin asetetaan satunnainen koko. Tässä on syytä huomata, että paketit ovat suhteellisen pieniä eikä niitä lähetetä kovin taajaan, sillä painopisteenä on menetelmän mekanismien tarkastelu eikä kapasiteettimittaus.

Luomisen jälkeen paketit välittyvät prosessorille (kuvassa ”wlan_mac_intf”). Välittäminen tapahtuu siten, että lähde lähettää keskeytyksen (interrupt), jonka prosessori poimii. Prosessorilla havaitaan keskeytyksen perusteella, että paketti tuli lähteestä, jolloin sille asetetaan kohdeosoite satunnaisesti. Kohdeosoitteen asettamisen jälkeen paketti välitetään MAC-prosessorille (”wireless_lan_mac”), joka puskuroi paketin ja lopulta lähettää sen lähtöporttiin (”wlan_port_tx0”).

Paketin saapuessa toimitaan siten, että vastaanottoporttiin (”wlan_port_rx0”) saapunut paketti välittyy MAC-kerroksen kautta prosessorille. Prosessori havaitsee jälleen keskeytyksen perusteella, että paketti on saapunut prosessoitavaksi. Koska paketti olisi saapunut prosessorille MAC:lta vain siinä tapauksessa, että se olisi kyseiselle solmulle tarkoitettu, se voidaan siirtää suoraan nieluun. Tämä toiminta johtuu siitä, että tässä mallissa ylempien protokollakerrosten toimintaa ei ole mallinnettu, eikä se toiminnallisuuden kannalta ole edes relevanttia.

Malli toimii siis siten, että luodut paketit lähetetään eteenpäin ja saapuneet siirretään suoraan nieluun tuhottavaksi.

Seuraavissa kappaleissa on kerrottu tarkemmin menetelmän toteutukseen liittyvistä asioista. Toteutus on muuttujien alustamista ja tarkempia tietorakenteisiin liittyviä yksityiskohtia lukuun ottamatta esitetty pseudokooditasolla liitteessä A.

4.1.4. Sekvenssit

Yllä kuvattua perusmallia on vaikea käyttää simulointiin sellaisenaan, koska se ei ylläpidä mitään sekvenssejä tai muuta tilatietoa yhteyksistä saati hypyistä. Ensimmäisenä malliin siis lisättiin kohdekohtaiset sekvenssilaskurit. Käytännössä paketin lähtiessä ulos, siihen leimataan sekvenssinumero. Ohjelman sisälle on luotu tietorakenne, joka sisältää kohdeosoitteen ja sekvenssinumeron. Paketin lähtiessä ulos haetaan listasta kohdeosoitteen perusteella oikea tietue, josta napataan sekvenssinumero leimattavaksi pakettiin.

Vastaanottopäässä tehdään vastaava operaatio, eli napataan paketista lähdeosoite ja sekvenssinumero ja päivitetään niitä vastaava tietue. Koska paketteja todennäköisesti katoaa välillä erinäisistä syistä (esimerkiksi kahden solmun samanaikainen lähetys), ei vastaanottopään sekvenssi

todennäköisesti ole simuloinnin aikana jatkuvasti sama kuin lähtöpään. Tästä lisää simulointituloksissa.

Pakettien otsikoissa oli perusmallissa vain kohdeosoite, mutta toiminnallisuuden laajentamisen vuoksi otsikkoon on lisätty myös paketin tyyppi (normaali, NACK, hACK), lähdeosoite ja sekvenssinumero. Sekvenssinumero on nimenomaan lähettäjä-vastaanottaja –kohtainen.

4.1.5. NACK-toiminnallisuus

NACK-toiminnallisuus on toteutettu siten, että vastaanotettaessa paketti sen sekvenssinumero tarkistetaan ja verrataan samasta osoitteesta tulleista paketeista ylläpidettyyn sekvenssinumeroon. Jos välistä puuttuu numeroita, luodaan NACK-viesti.

NACKin lähetykseen tarvitaan viimeisimmän sekvenssissä olevan vastaanotetun paketin sekvenssinumero sekä sekvenssistä poikkeavan paketin numero. Näiden perusteella saadaan helposti selville, montako pakettia ja ennen kaikkea mitkä paketit välistä puuttuvat. Tämän jälkeen jokaista puuttuvaa pakettia kohden lähetetään oma NACK-viestinsä lähettäjälle.

Lähettäjäpää vastaanottaa NACK-viestin, jossa olevien otsikkotietojen (lähde, kohde, sekvenssi) perusteella se osaa lähettää puuttuvan sekvenssinumeron paketin. Uudelleenlähetysten jälkeen sekvenssi palautuu jälleen eheäksi.

Yksinkertaisuuden vuoksi varsinaisesta puskuroinnista on mallinnuksen puitteissa luovuttu. Sen sijaan NACKeihin perustuen lähetetään vain aiemmin lähetettyjä imitoivia paketteja. Toisin sanoen luodaan uusi paketti, johon leimataan vain välistä puuttumaan jäänyt sekvenssinumero. Puskuroinnin pois jättäminen selittyy sillä, että sovellustason protokollia ei ole tässä mallissa mukana, jolloin pakettien oikealla sisällöllä ei ole varsinaista merkitystä. Jos ylempien tasojen protokollat olisivat yhtälössä mukana, täytyisi tietysti lähettää kadonnutta pakettia vastaava identtinen paketti, jotta ylemmän tason toiminnallisuus säilyisi.

4.1.6. hACK-toiminnallisuus

Kuten aiemmin on tullut ilmi, NACK-pohjainen toteutus ei riitä, jos halutaan varmistaa yksittäisten pakettien tai pakettivuon viimeisen paketin siirto. Yksittäisen paketin kadotessa tai vuon viimeisen paketin kadotessa ei NACKia tule ennen kuin seuraava vuo saapuu. Toiminnallisuutta ei kuitenkaan saa jättää sen varaan, että seuraava vuo tulee ”joskus”, koska se saattaa tulla 10 sekuntin tai vaikka

tunnin kuluttua. Yksittäisiä ja vuon viimeisiä paketteja varten on toteutettu hyppykohtainen hACK (hop-by-hop ACK) toiminnallisuus.

Lähetyspäässä jokaisen lähetetyn paketin jälkeen käynnistetään uudelleenlähetysajastin. Jos ajastin kuluu loppuun ennen kuin hACKia kohteesta saapuu, lähetetään viimeisin paketti uudestaan. Tämä ajastin nollataan ja käynnistetään uudestaan aina kun kohteeseen lähetetään uusi paketti.

Vastaanottopäässä vastaavasti viimeisimmän paketin saapuessa käynnistetään myös ajastin, mutta tässä tapauksessa ajastin hACK-paketin lähetystä varten. Uuden paketin saapuessa ajastin käynnistetään aina alusta, jotta hACKeja ei lähetetä ”turhaan” jokaisesta paketista. NACKit hoitavat välistä puuttuvat paketit.

Ajastimien pituuden suhteen pitää tehdä kompromissi viimeisen paketin kuittauksen odotuksen ja turhien hACKien lähetyksen välillä. Turha tarkoittaa tässä tapauksessa sitä, että lähetetty paketti kuittautuisi siedettävän ajan kuluessa sekvenssin seuraavalla paketilla. Siedettävä aika riippuu tietysti sovelluksesta, ja se on helposti räätälöitävissä haluttuun arvoon.

Ajastimet on asetettu siten, että uudelleenlähetystä odotetaan hivenen kauemmin kuin hACKin odotettavissa olevaa saapumista, koska kuittausviestin lähettäminen vie useimmissa tapauksissa vähemmän kaistaa kuin varsinaisen paketin uudelleenlähetys.

Ohjelmistotasolla toteutus on tehty siten, että lähetyspäässä on tietorakenteessa kohdeosoite ja sitä vastaavan skeduloidun uudelleenlähetystapahtuman kahva (handle). Uuden paketin lähtiessä tapahtuma peruutetaan ja skeduloidaan uusi tapahtuma, joka päivitetään tietorakenteeseen. Vastaanottopäässä on vastaava toteutus, mutta uudelleenlähetyksen sijaan on skeduloitu hACKin lähetys.

4.2. Simulointi ja skenaariot

Tässä osassa esitellään eri simulointitapaukset ja kuvaillaan eri skenaarioiden yksityiskohdat.

4.2.1. Vanilla toimintamalli

Vanilla tarkoittaa tässä yhteydessä täysin perustapausta ilman mitään erikoisuuksia. Verkon solmuihin on lisätty pakettilaskurit, mutta muuten toiminta on täysin sama kuin aivan perusmallilla, eli mitään NACK- tai hACK-toimintoja ei ole käytössä.

Tässä skenaariossa kaksi solmua yksinkertaisesti lähettävät toisilleen paketteja. Lähetettyjen ja vastaanotettujen pakettien määrästä kerätään statistiikkaa, jonka perusteella voidaan kuvaajista verrata lähetettyjen ja vastaanotettujen pakettien suhdetta.

4.2.2. HBH toimintamalli

Tässä skenaariossa on myös kaksi solmua jotka kommunikoivat keskenään. Nyt mukana on luotettavaan tiedonsiirtoon tarvittavat mekanismit, eli NACK- ja hACK-toiminnallisuus. Lähetettyjen ja vastaanotettujen pakettien lisäksi kerätään statistiikkaa myös lähetetyistä NACK- ja hACK-paketeista.

4.2.3. Vanilla toimintamalli häirittynä

Häirityissä skenaarioissa paketteja hävitetään tietyllä todennäköisyydellä. Eri tapaukset ovat 10%, 30%, 50% ja 80% pakettihukka. Tässä skenaariossa häirittyinä ovat vanilla-tyyppiset solmut. Kerätyn statistiikan perusteella voidaan tarkastella häirinnän vaikutusta tiedonsiirron luotettavuuteen. Paketteja katoaa toisistaan riippumatta, eli jokaisen paketin katoaminen määritellään erikseen. Tahallisia häviöpurskeita ei siis luoda, mutta kuten tuloksista selviää, häviöpurskeita syntyy tahattomastikin, joka on jopa toivottava piirre.

4.2.4. HBH toimintamalli häirittynä

Tämä skenaario on vastaava kuin yllä, mutta käytössä on NACK- ja hACK-toiminnallisuus. Pakettihukkana käytetään 5%, 10%, 20%, 30%, 50% ja 80% arvoja. Statistiikan perusteella voidaan tarkastella, miten toimintamalli käyttäytyy häirittynä. Avainkysymyksenä se, parantaako toimintamalli luotettavuutta merkittävästi verrattuna vanilla-malliin.

5. Tulokset

Tässä osassa esitetään edellisessä osassa esiteltyjen simulointitapausten tulokset. Perus- eli vanilla-tapauksista esitetään vähemmän tietoa, koska niissä ei erikoisominaisuuksia ole implementoitu. Tämän vuoksi monet HBH-menetelmän tunnusluvuista ovat joko epärelevantteja tai niitä ei ole saatavilla.

Ensin esitellään Vanilla-tapauksien tulokset eri pakettihukilla. Näiden jälkeen esitetään HBH-menetelmän tuloksia. HBH on kiinnostavampi, joten sen skenaarioista esitetään tarkemmat ja yksityiskohtaisemmat tiedot.

5.1. Skenaariot

Skenaarioista on esitetty aluksi taulukko, josta ilmenee simulaation aikana kerätty pakettistatistiikka. Tämän jälkeen esitetään kuvaajat, joista ilmenee lähettäjän ja vastaanottajan sekvenssit ajan funktiona. Näiden avulla voidaan seurata, miten lähetettyjen ja vastaanotettujen pakettien määrät kehittyvät simulaation ajon aikana.

5.1.1. Vanilla – ei pakettihukkaa

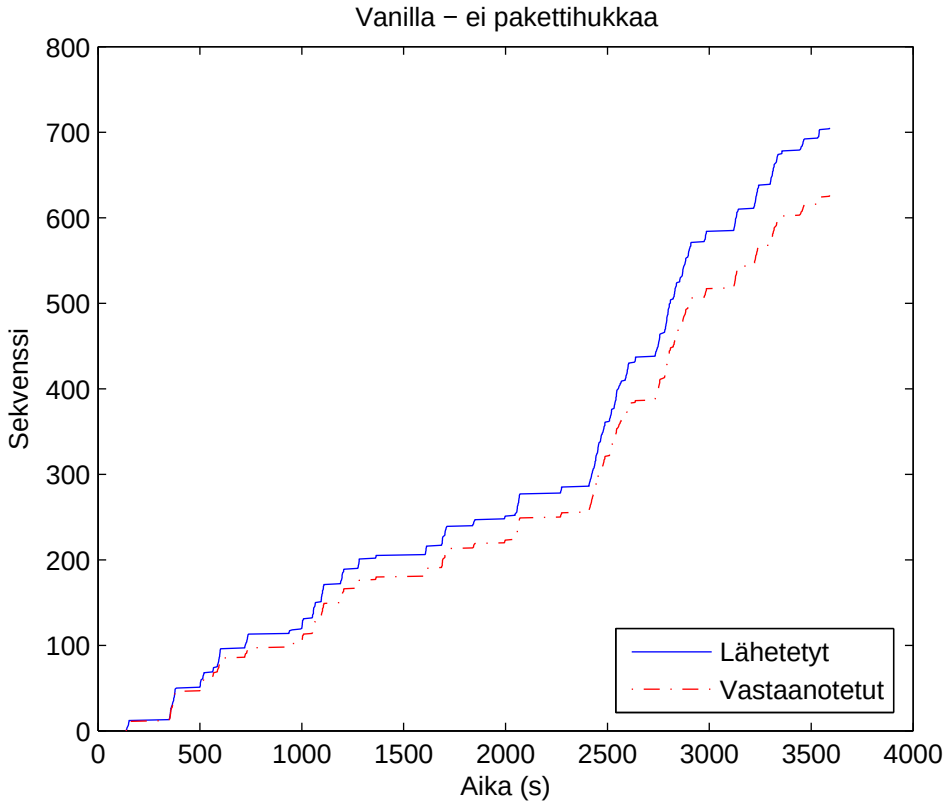
Ensimmäisessä tapauksessa simuloitiin Vanilla-tyyppisen solmun lähetys- ja vastaanottoprosessia.

Taulukko 2 Vanilla-solmun pakettistatistiikkaa

	Vanilla 1	Vanilla 2
Paketteja lähetty	706	512
Paketteja vastaanotettu	457	627

Taulukossa 2 on esitetty solmujen lähettämät ja vastaanottamat pakettimäärät. Vanilla 1 tarkoittaa ensimmäistä solmua ja Vanilla 2 toista. Kuvassa 13 pakettien sekvenssit ovat nähtävissä ajan

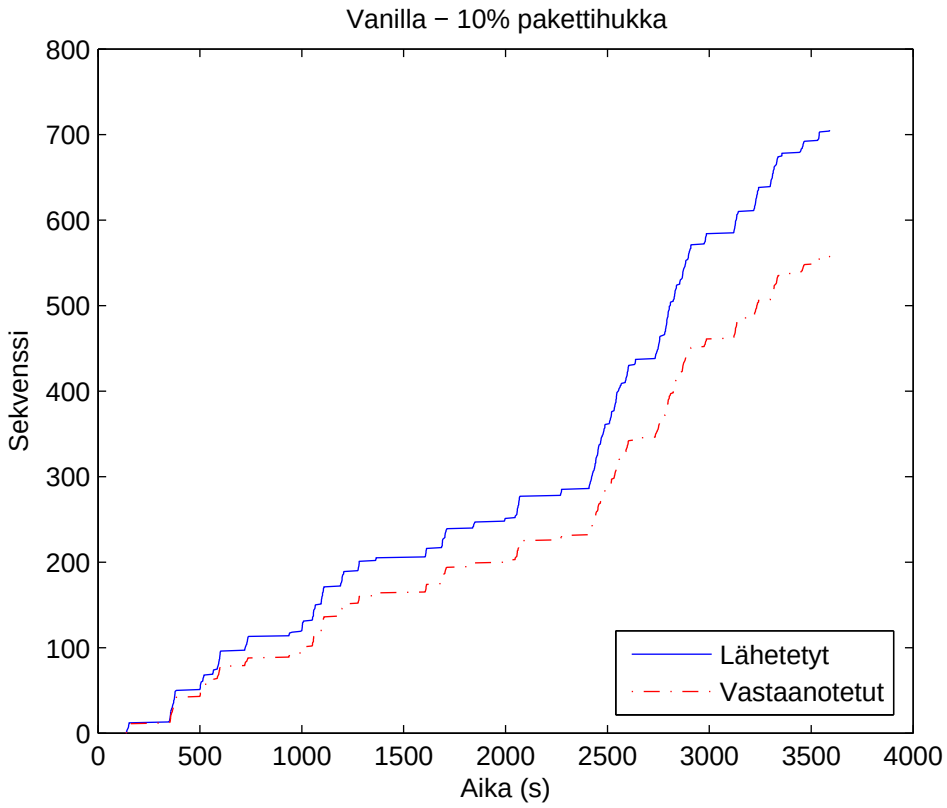
funktiona. Pakettihukka on melko huomattava jopa ilman erillistä lisättyä pakettihukkaa. Pakettihukan määrä on keskimäärin noin 11% laskettuna vastaanotettujen ja lähetettyjen pakettien perusteella ilman lisättyä pakettihukkaa.



Kuva 13 Vanilla-solmut ilman pakettihukkaa

5.1.2. Vanilla – 10% pakettihukka

Tässä skenaariossa käytössä oli samat solmut kuin edellisessä, mutta simulaatioon on lisätty vielä keinotekoinen 10% pakettihukka. Pakettihukka on toteutettu yksinkertaisesti siten, että jokaisen paketin kohdalla arvotaan satunnaisluku, joka määrää hylätäänkö paketti vai ei.



Kuva 14 Vanilla-solmut 10% pakettihukalla

Taulukko 3 Vanilla-solmun pakettistatistiikkaa 10% pakettihukalla

	Vanilla 1	Vanilla 2
Paketteja lähetty	706	512
Paketteja vastaanotettu	412	559

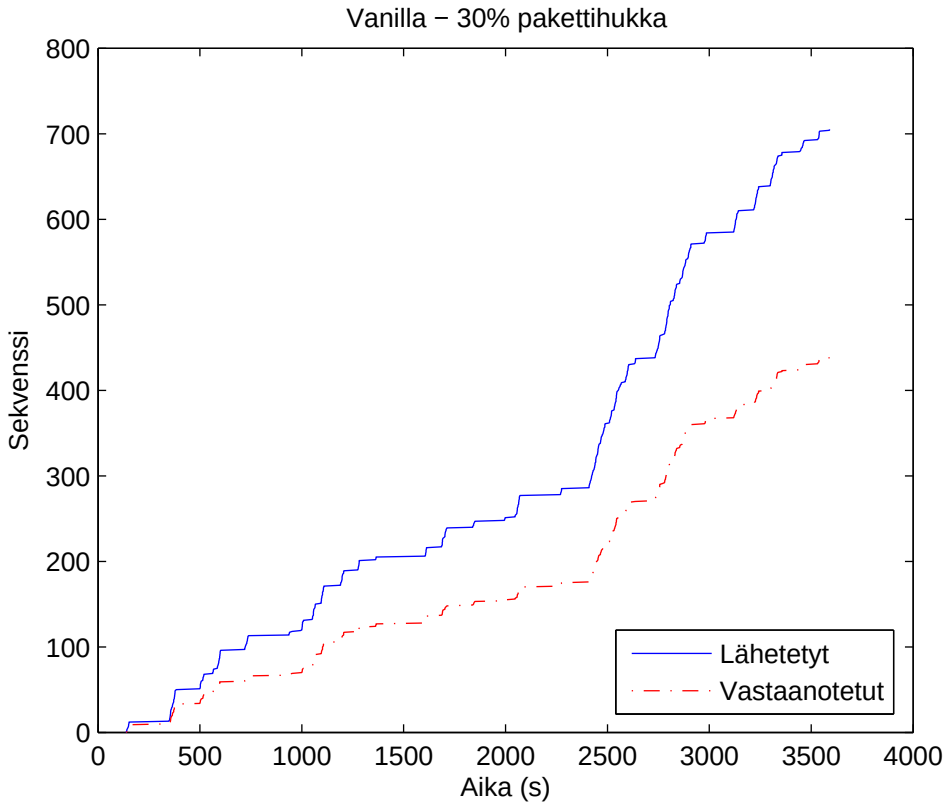
Kuvasta 14 ja taulukosta 3 havaitaan samankaltainen toiminta kuin ”pakettihukattomassa” tapauksessa, mutta lähetettyjen ja vastaanotettujen pakettien määrät vain eroavat toisistaan entistä enemmän.

Seuraavat pakettihukat eivät varsinaisesti tuo mitään lisätietoa käyttäytymisestä, mutta kuvista havaitaan selkeästi lähetettyjen ja vastaanotettujen pakettien määrien erojen kasvu pakettihukan kasvaessa.

5.1.3. Vanilla – 30% pakettihukka

Taulukko 4 Pakettistatistiikkaa (Vanilla 30% pak.huk.)

	Vanilla 1	Vanilla 2
Paketteja lähetetty	706	512
Paketteja vastaanotettu	323	439

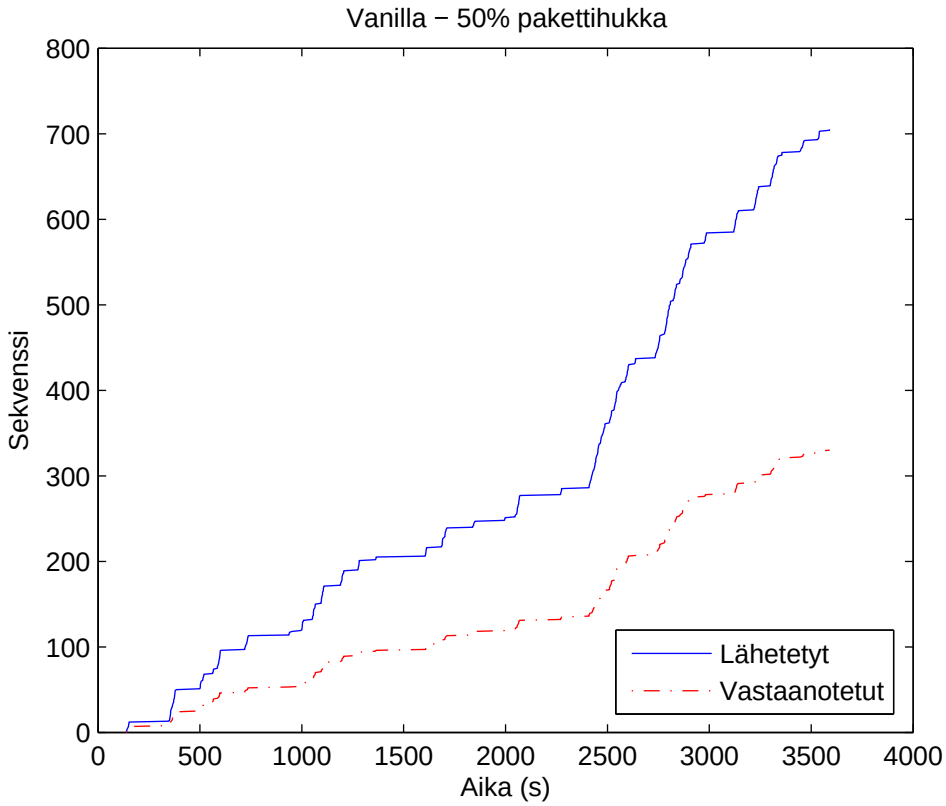


Kuva 15 Vanilla 30% pakettihukka

5.1.4. Vanilla – 50% pakettihukka

Taulukko 5 Pakettistatistiikkaa (Vanilla 50% pak.huk.)

	Vanilla 1	Vanilla 2
Paketteja lähetetty	706	512
Paketteja vastaanotettu	233	331

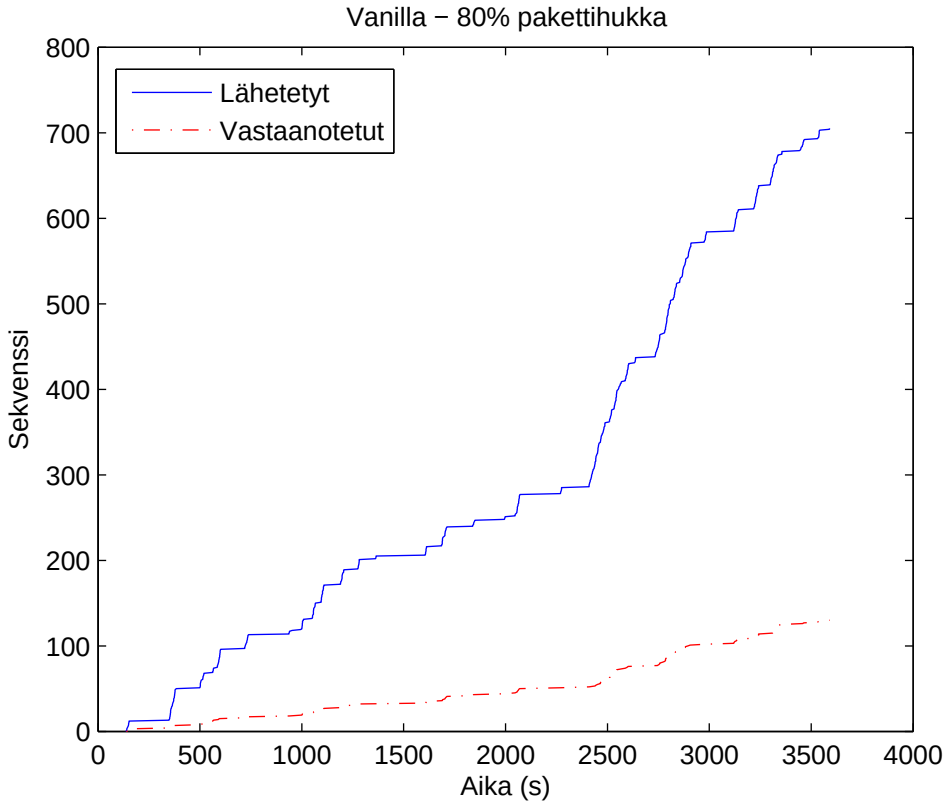


Kuva 16 Vanilla 50% pakettihukka

5.1.5. Vanilla – 80% pakettihukka

Taulukko 6 Pakettistatistiikkaa (Vanilla 80% pak.huk.)

	Vanilla 1	Vanilla 2
Paketteja lähetetty	706	512
Paketteja vastaanotettu	84	131



Kuva 17 Vanilla 80% pakettihukka

5.1.6. HBH – ei pakettihukkaa

Taulukko 7 Pakettistatistiikkaa (HBH ei pak.huk.)

	HBH 1	HBH 2
Paketteja lähetty	786	681
Paketteja otettu vastaan	623	708
NACKeja lähetetty	45	60
hACKeja lähetetty	80	104
Uudelleenlähetys	18	13
Paketteja sekvenssissä	707	622

Kuten vanilla-tapauksista kävi ilmi, pakettihukkaa ilmenee, vaikka sitä ei keinotekoisesti lisättäisi. Tämä voi johtua esimerkiksi siitä, että kummatkin solmut sattuvat lähettämään samaan aikaan. Yleensä jos solmu lähettää, se ei välttämättä voi kuunnella muiden lähetyksiä samanaikaisesti, joten lähetysten signaalit sotkeutuvat mennessään päällekkäin, tai ne yksinkertaisesti jäävät ”kuulematta” toisessa päässä joka lähettää samalla hetkellä.

Taukossa 7 on esitetty pakettihukattoman tapauksen pakettistatistiikkaa. Lähetettyjen pakettien määrä solmulta HBH 1 saadaan siten, että lisätään HBH 1:n sekvenssissä oleviin paketteihin uudelleenlähetykset sekä solmun HBH 2 lähettämien NACKien määrä. HBH 2:n lähettämät NACKit laukaisevat HBH 1:ssä näiden pakettien uudelleenlähetyksen, joka näkyy lähetetyissä paketeissa. Uudelleenlähetys-rivi kertoo, montako pakettia on lähetetty uudelleen sen takia, että hACKia ei ole saapunut tai hACKissa on ollut väärä sekvenssinumero. Kokonaishallintatiedon (overhead) määrä saadaan laskemalla sekvenssissä olevien pakettien lisäksi lähetettyjen pakettien määrä.

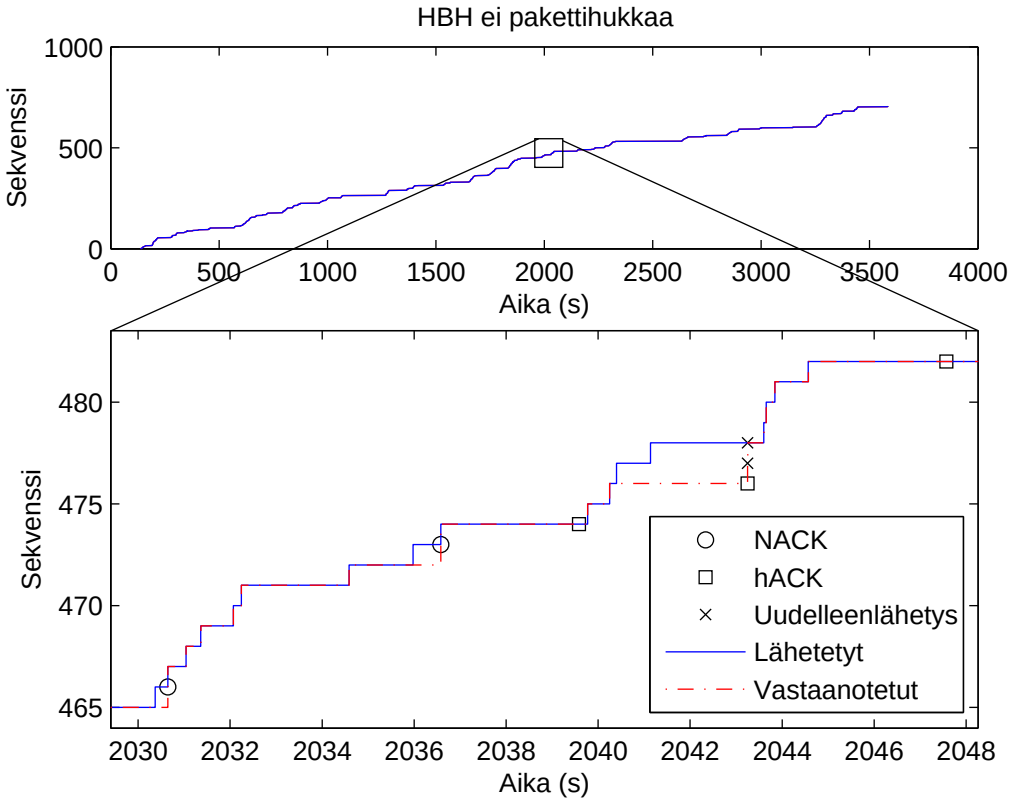
$$Overhead = \frac{Kaikki\ lähetetyt - Paketteja\ sekvenssissä}{Paketteja\ sekvenssissä} \times 100\,\% \qquad (5)$$

Kaikki lähetetyt sisältää kyseisen solmun lähettämien pakettien lisäksi myös vastapäisen solmun lähettämät NACKit ja hACKit, koska nämä ovat myös ylimääräistä hallintaliikennettä.

$$Overhead_{HBH1} = \frac{786 + 60 + 104 - 707}{707} \times 100\% = 34\%$$

$$Overhead_{HBH2} = \frac{681 + 45 + 80 - 622}{622} \times 100\% = 30\%$$

Ylimääräistä hallintatietoa syntyy siis noin 30% ilman lisättyä pakettihukkaa.

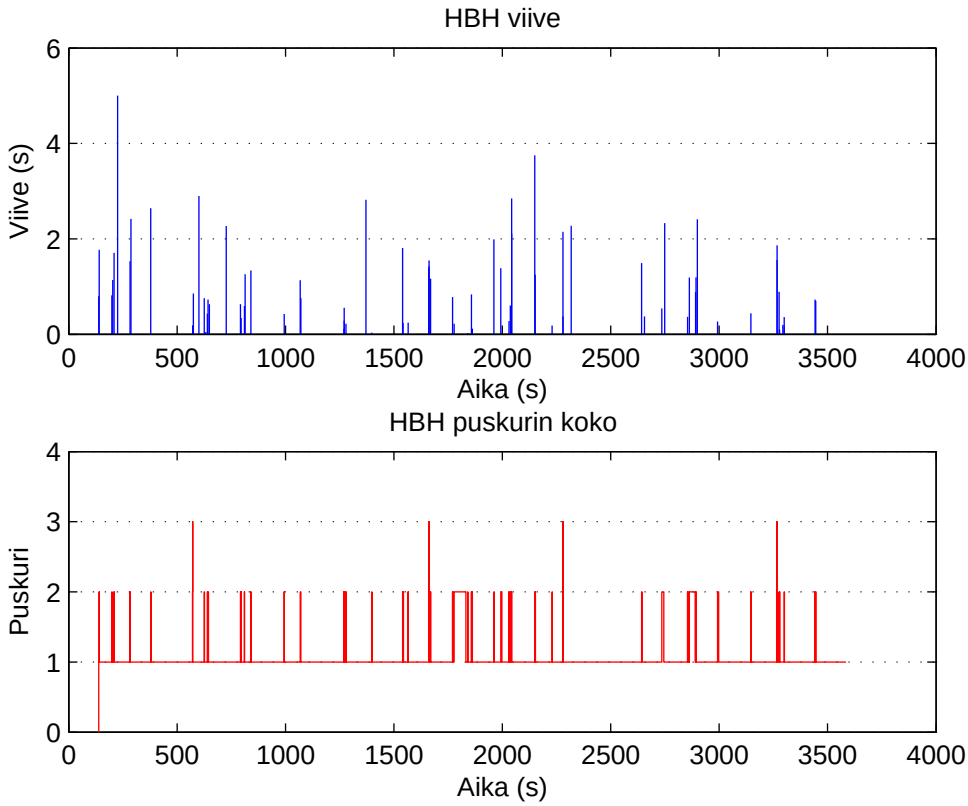


Kuva 18 HBH ei pakettihukkaa

Kuvassa 18 on esitetty kuvaaja lähetettyjen ja vastaanotettujen pakettien määrästä sekä tarkennettu yksityiskohta. Yksityiskohdasta nähdään, että alkupuolella (noin ajanhetkellä 2030s) on jäänyt paketti välistä, ja puute korjataan nopeasti NACKilla. Myös ajanhetkellä 2036s on samankaltainen tilanne, jossa jälleen paikataan puute NACKilla. Hetkellä 2040s saapuu hACK, joka pysäyttää uudelleenlähetyksen ajastimen. Ajanhetken 2041s tienoilla saapuu kaksi pakettia, jotka kummatkin katoavat. Vastaanottaja ei tiedä näistä paketeista mitään, ja lähettää hACKin viimeisimmän vastaanotetun paketin sekvenssillä. Lähettäjä huomaa, että kahta viimeistä pakettia ei ole otettu vastaan, ja ne lähetetään uudestaan ajanhetkellä 2044s. Tämän jälkeen saapuu vielä joitakin paketteja, jotka kuitataan odotetusti ajanhetkellä 2048s.

Mainittakoon, että hACKien lähetyksen ja uudelleenlähetyksen ajastimen ajoiksi on valittu 3 ja 5 sekuntia vastaavasti. Oikean sovelluksen tapauksessa (esimerkiksi reaaliaikaisen puheen) näin suuri viive ei käy laatuun. Kuitenkin suhteessa pakettien saapumisaikojen välin keskiarvoon, joka on yksi sekunti, nämä aika-arvot ovat sopivat. Suuret pakettien saapumisväliajat on valittu siksi, että menetelmän ominaisuudet tulevat paremmin esille verkkaisemman lähetystahdin kanssa. Sekunneista voi ottaa halutessaan vaikka yhden tai kaksikin nollaa pois, jolloin saadaan

nopeamman tiedonsiirtojärjestelmän lähetystiheys. Näissä tapauksissa kulkuaikaviiveet kasvavat suhteessa suuremmiksi, mutta sillä ei menetelmän puitteissa ole merkitystä.



Kuva 19 HBH viive ja puskurin koko

Kuvassa 19 on esitetty esiintyvät viiveet sekä tarvittava puskurin koko. Viive on laskettu suoraan erotuksena paketin lähetys- ja vastaanottoajasta. Kulkuaikaviiveet eivät näy kuvassa, koska ne ovat niin pieniä. Viiveet ovatkin tässä tapauksessa syntyneet siitä, että lähetetty paketti on kadonnut, ja se on jouduttu lähettämään uudestaan. Esitetty viive ei siis tosiasiassa ole varsinainen kulkuaikaviive, vaan se aika, joka on kulunut kun paketti on ensimmäisen kerran lähetetty ja kun se on viimein otettu vastaan. Kolmen sekunnin viive tarkoittaa sitä, että paketti on huomattu kadonneeksi hACKin vastaanoton yhteydessä ja viiden sekunnin viive tarkoittaa sitä, että hACKia ei ole saapunut ja paketti on lähetetty uudestaan. Muut viiveet tarkoittavat, että sekvenssi on paikattu NACKeilla.

Puskurin koko kertoo, kuinka monta pakettia kerkeää puskuroitua, ennen kuin ne saadaan otettua vastaan ja kuitattua. Puskurin koko on minimissään 1, ja tämä johtuu siitä, että puskurin koko on laskettu aina lähetyshetkellä, jolloin lähetetty paketti on aina vähintään yhtä suuremmalla

sekvenssinumerolla kuin viimeisin vastaanotettu paketti. Kuten Vanilla-tapauksen yhteydessä todettiin, systeemissä esiintyy noin 10% pakettihukka ilman keinotekoista pakettihukkaa. Tästä voidaan laskea, että todennäköisyys sille, että yli kolme pakettia katoaa ”putkeen” on luokkaa 0,10*0,10*0,10*100% = 0,1%. Tällä perusteella kolmen paketin puskurin voidaan suurella varmuudella olettaa riittävän tässä tapauksessa.

5.1.7. HBH – 5% pakettihukka

Taulukko 8 Pakettistatistiikkaa (HBH 5% pak.huk.)

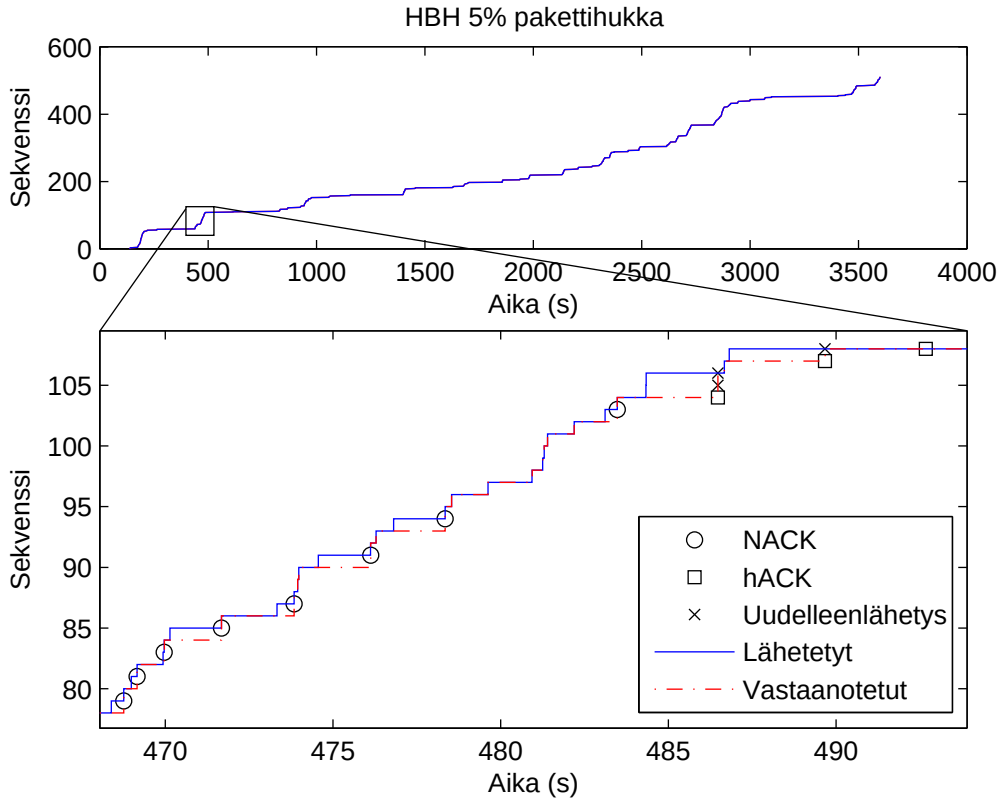
	HBH 1	HBH 2
Paketteja lähetty	609	630
Paketteja otettu vastaan	546	513
NACKeja lähetetty	67	84
hACKeja lähetetty	109	77
Uudelleenlähetys	17	23
Paketteja sekvenssissä	511	541

5%:n pakettihukka ei tuo vielä merkittävää kasvua hallintaliikenteen kannalta tai menetelmän toiminnan kannalta muutenkaan. Tilanne näyttää hyvin samankaltaiselta kuin edellisessä tapauksessa.

$$Overhead_{HBH1} = \frac{609 + 84 + 77 - 511}{511} \times 100\% = 51\%$$

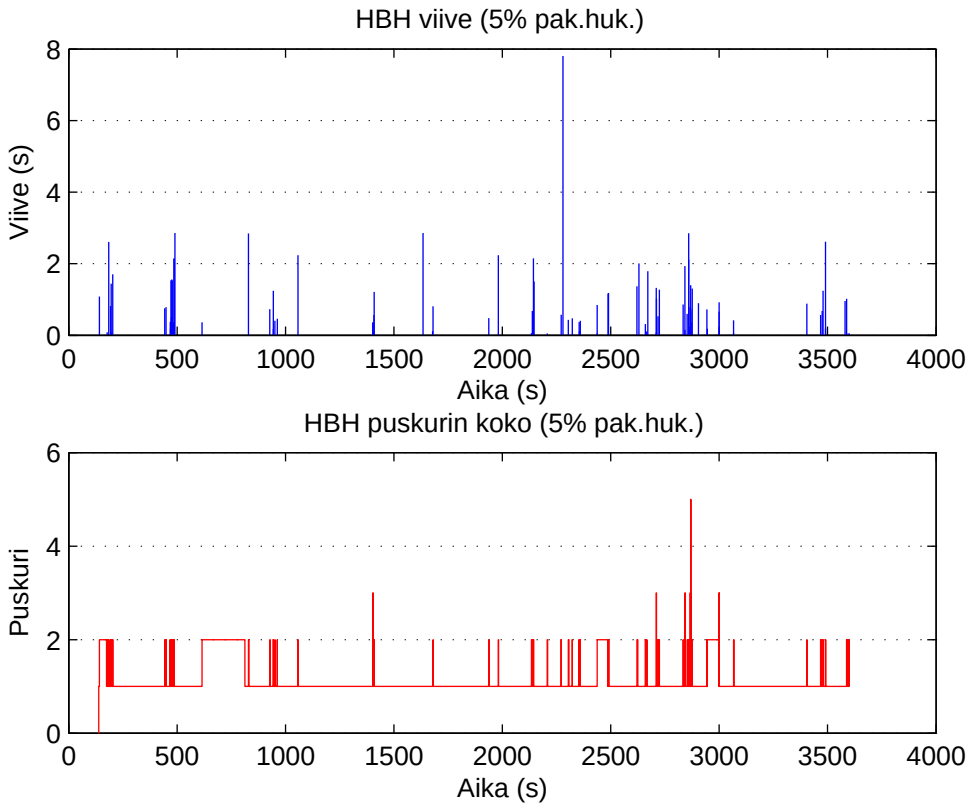
$$Overhead_{HBH2} = \frac{630 + 67 + 109 - 541}{541} \times 100\% = 49\%$$

Hallintatiedon määrässä tapahtuu pientä kasvua verrattuna pakettihukattomaan tapaukseen.



Kuva 20 HBH 5% pakettihukka

Kuvassa 20 esitetty kuvaaja vastaa hyvin paljon pakettihukattoman tapauksen kuvaajaa. 5%:n pakettihukka ei ole niin dramaattinen, että se aiheuttaisi vielä merkittävää suorituskyvyn heikkenemistä. Yksityiskohtaisen kuvaajan loppupäästä havainnollistuu selvästi, miten välistä jää paketti. Se lähetetään uudelleen hACKin perusteella, ja uudelleenlähetetystä paketista lähetetään uusi hACK.



Kuva 21 HBH viive ja puskurin koko (5% pak.huk.)

Kuvasta 21 havaitaan, että 5% pakettihukka ei aiheuta dramaattista muutosta viiveeseen tai puskurin kokoon. Yksi pidempi viive noin 2300 sekunnin kohdalla on ainoa muista erottuva tapaus. Tässä kohtaa on kadonnut muutama paketti epäsuotuisasti, jolloin viive on päässyt venähtämään.

Puskurin koko käy huippuarvossaan noin 2800 sekunnin kohdalla. Kuvasta sitä ei näe, mutta kyseisessä kohdassa katoaa neljä pakettia peräkkäin. Tämän tapahtumasarjan todennäköisyys tällä pakettihukalla on luokkaa 0,15*0,15*0,15*0,15*100% = 0,051%. Kyseessä on siis erittäin epäonnekas sattuma. Kuten aiemmassa tapauksessa, tässäkin on otettu huomioon 10%:n ominainen pakettihukka ennen lisättyä pakettihukkaa. Neljän paketin puskurin voidaan riittävällä varmuudella olettaa riittävän tällaisella pakettihukalla.

5.1.8. HBH – 10% pakettihukka

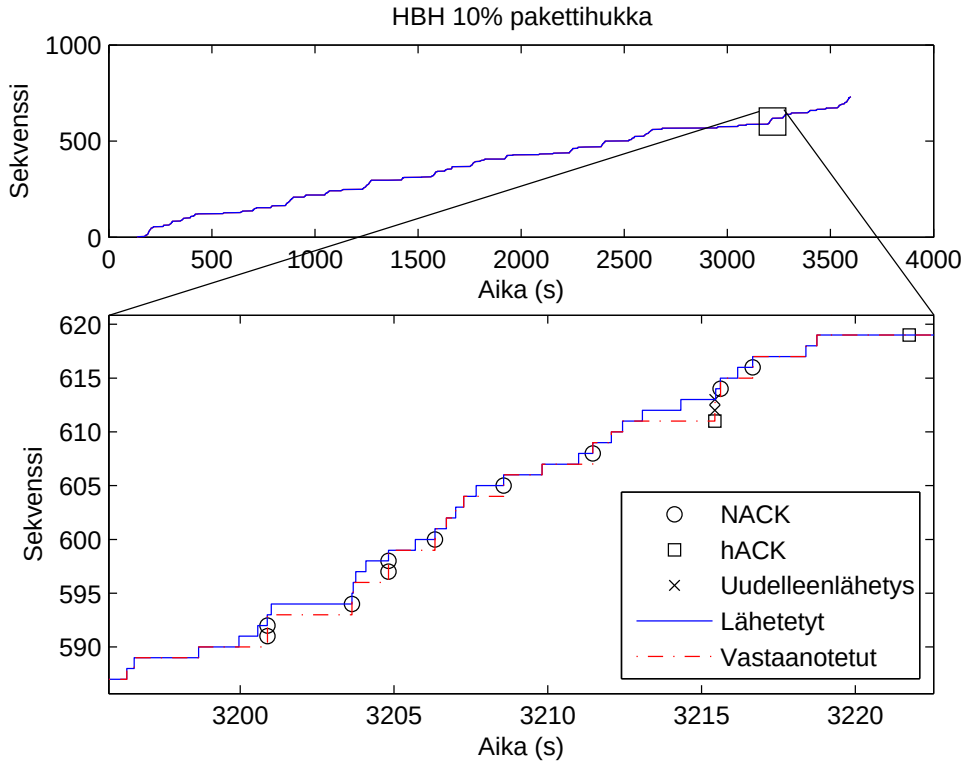
Taulukko 9 Pakettistatisiikkaa (HBH 10% pak.huk.)

	HBH 1	HBH 2
Paketteja lähetty	902	649
Paketteja otettu vastaan	543	739
NACKeja lähetetty	94	136
hACKeja lähetetty	93	109
Uudelleenlähetys	45	31
Paketteja sekvenssissä	731	537

10% pakettihukan tapauksessa havaitaan pientä hallintatiedon lisääntymistä, mutta toiminta pysyy suhteellisen stabiilina.

$$Overhead_{HBH1} = \frac{902 + 136 + 109 - 731}{731} \times 100\% = 57\%$$

$$Overhead_{HBH2} = \frac{649 + 94 + 93 - 537}{537} \times 100\% = 56\%$$

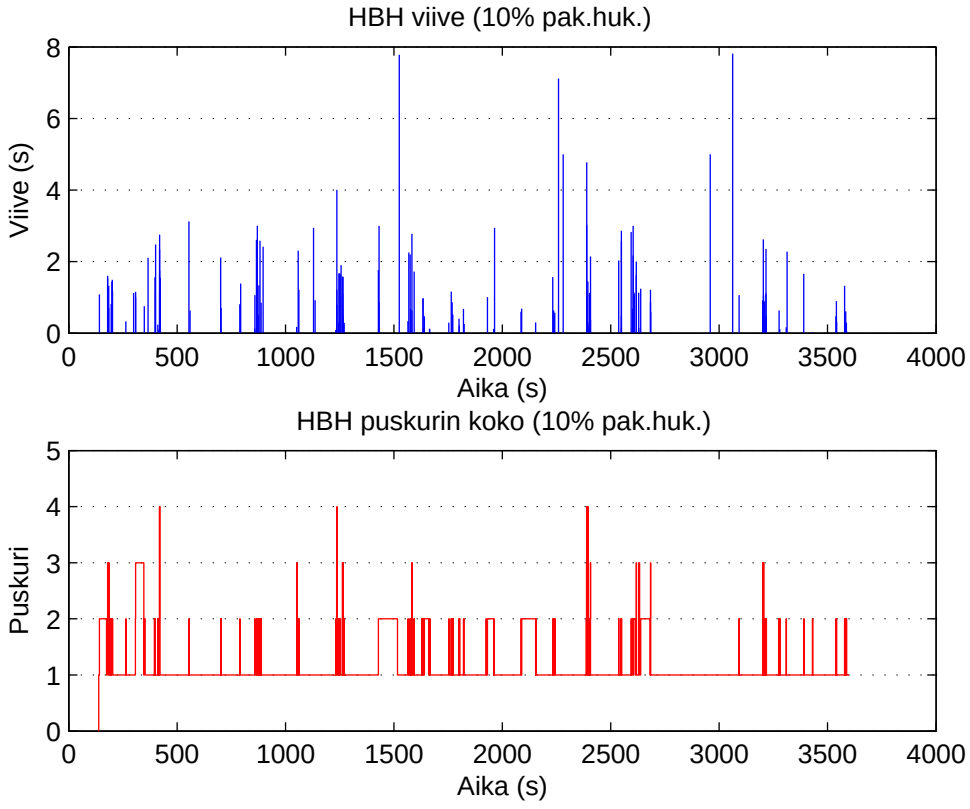


Kuva 22 HBH 10% pakettihukka

NACKien määrä lisääntyy jonkin verran, kuten kuvasta 22 havaitaan. Pakettihukista palaudutaan kuitenkin nopeasti.

Kuvasta 23 nähdään, että viivepiikkejä on useampia kuin 5% pakettihukan tapauksessa. Ero ei kuitenkaan ole merkittävä.

Puskurin koko ei ylitä arvoa neljä, mutta se käy useammin korkeammalla kuin 5% tapauksessa. Todennäköisyys, että neljä pakettia katoaa peräkkäin on 0,20*0,20*0,20*0,20*100% = 0,16%. Todennäköisyys on melko pieni, mutta kuten 5% tapauksessa kävi ilmi, ei tarvita kuin hieman epäonnea, jotta puskuri kasvaa suuremmaksi. Toisaalta neljän paketin puskurin voi olettaa tässä tapauksessa riittävän, koska viidennen paketin katoaminen samaan putkeen tapahtuu todennäköisyydellä 0,032%.



Kuva 23 HBH viive ja puskurin koko (10% pak.huk.)

5.1.9. HBH – 20% pakettihukka

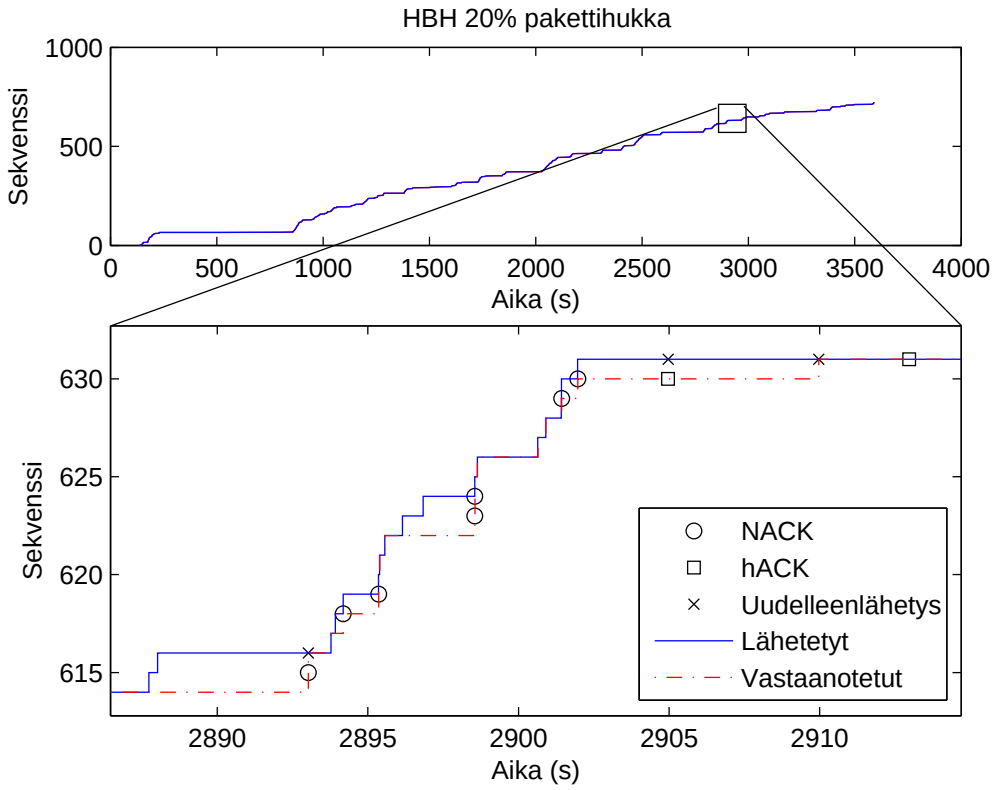
Taulukko 10 Pakettistatistiikkaa (HBH 20% pak.huk.)

	HBH 1	HBH 2
Paketteja lähetty	977	698
Paketteja otettu vastaan	521	734
NACKeja lähetetty	154	254
hACKeja lähetetty	121	116
Uudelleenlähetys	58	78
Paketteja sekvenssissä	723	506

$$Overhead_{HBH1} = \frac{977 + 254 + 116 - 723}{723} \times 100\% = 86\%$$

$$Overhead_{HBH2} = \frac{698 + 154 + 121 - 506}{506} \times 100\% = 92\%$$

20% pakettihukalla ylimääräistä hallintatietoa lähetetään jo lähes 100% hyötydatan määrästä, joten menetelmän tehokkuus alkaa hieman kärsiä. Toisaalta, kuten kuvasta 24 havaitaan, pakettihukista palaudutaan vielä todella hyvin.

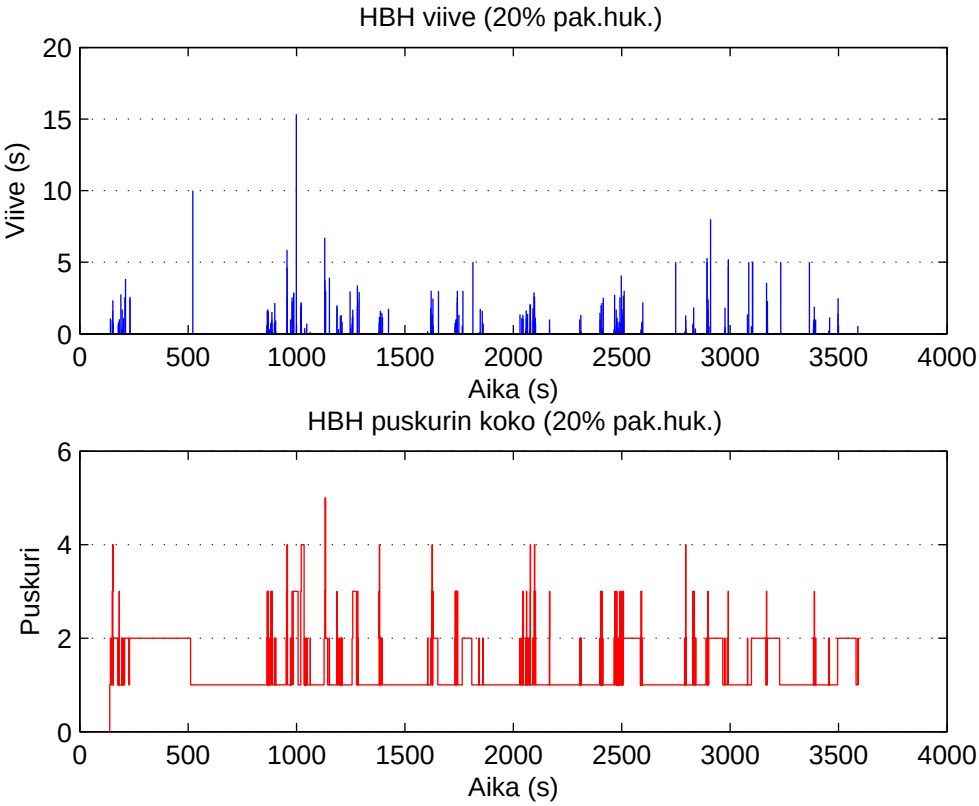


Kuva 24 HBH 20% pakettihukka

Kuvassa 25 esitetyt viiveet ovat hieman kasvaneet 10% pakettihukkaan verrattuna, minkä lisäksi muutaman pidemmän viiveen tapauksessa viiveen määrä on kasvanut. Uudelleenlähetyksestä johtuvia viiden sekunnin viiveitä on huomattavasti enemmän kuin edellisessä tapauksessa.

Puskurin koko käy useammin neljässä kuin aiemmin, ja keskimäärin se on kokoajan korkeampi kuin aiemmassa tapauksessa. Todennäköisyys viiden paketin katoamiselle peräkkäin on noin 0,30*0,30*0,30*0,30*0,30*100% = 0,243%. Lukema ei ole pieni, joten on onnekasta että neljäkin pakettia on kadonnut putkeen vain kerran simulaation aikana. Tässä tapauksessa puskurin kooksi olisi asetettava siis ainakin viisi pakettia.

Ajanhetkien 200 ja 500 sekuntia välillä puskurin koko pysyy pitkään arvossa kaksi. Tämä johtuu siitä, että tällä välillä ei lähetetä uusia paketteja, jolloin arvoa ei päivitetä. Kuvaa tarkastelemalla kuitenkin havaitaan, että sekvenssi on eheä myös tällä välillä.



Kuva 25 HBH viive ja puskurin koko (20% pak.huk.)

5.1.10. HBH – 30% pakettihukka

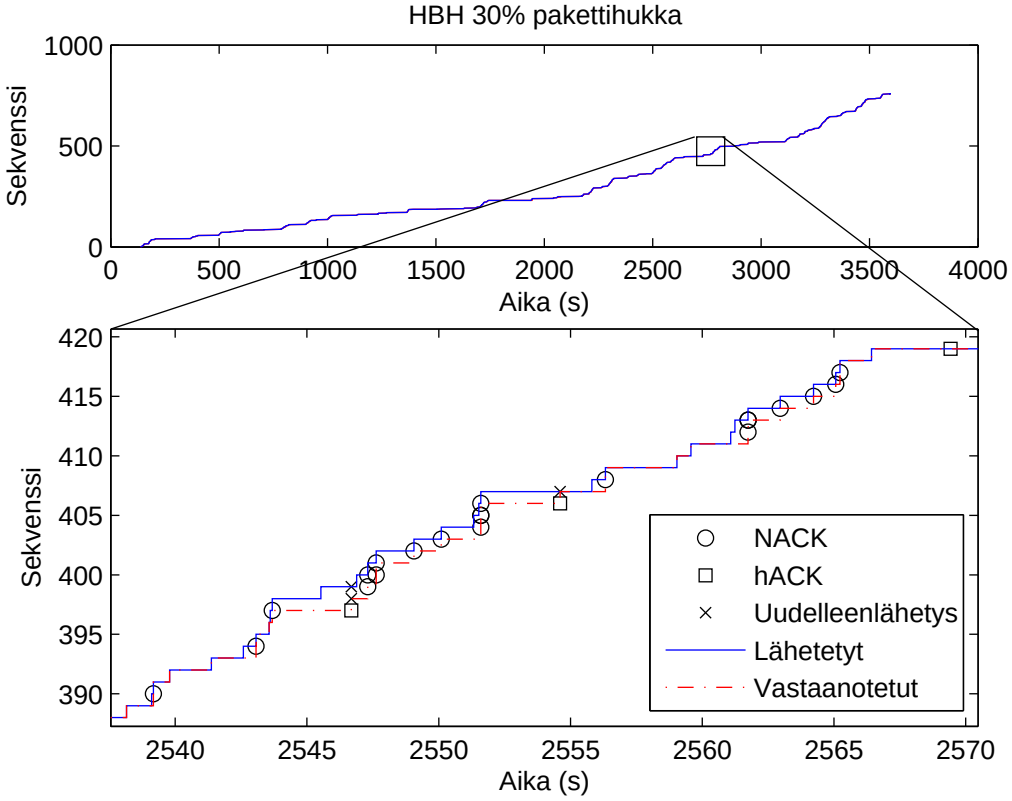
Taulukko 11 Pakettistatistiikkaa (HBH 30% pak.huk.)

	HBH 1	HBH 2
Paketteja lähetty	836	921
Paketteja otettu vastaan	588	562
NACKeja lähetetty	336	285
hACKeja lähetetty	140	144
Uudelleenlähetys	114	106
Paketteja sekvenssissä	527	571

$$Overhead_{HBH1} = \frac{836 + 285 + 144 - 527}{527} \times 100\% = 140\%$$

$$Overhead_{HBH2} = \frac{921 + 336 + 140 - 571}{571} \times 100\% = 145\%$$

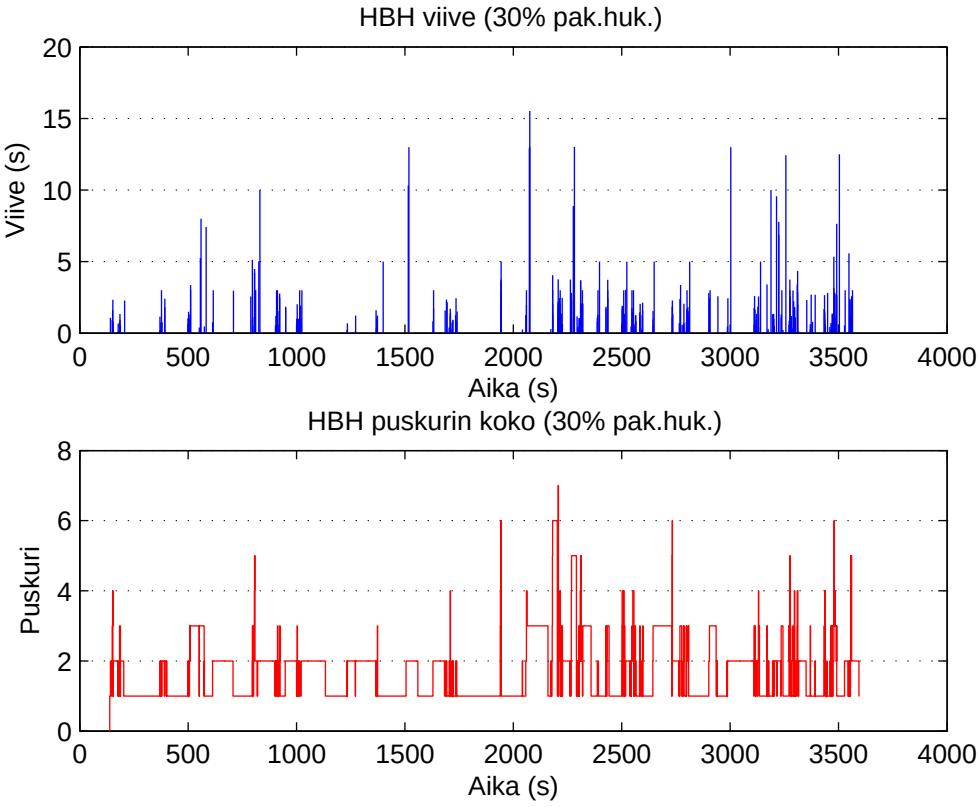
Hallintatiedon määrä alkaa olla 30% pakettihukalla jo melko huomattava, joten tehokkuus ei ole enää erityisen hyvä.



Kuva 26 HBH 30% pakettihukka

Kuten kuvasta 26 nähdään, menetelmä pysyy silti melko stabiilina vielä näinkin suurella pakettihukalla. NACKien määrä on selvästi lisääntynyt aiempiin tapauksiin verrattuna, mutta tämä on odotettavissakin.

Kuvassa 27 esitettyjen viivepiikkien määrä on lisääntynyt huomattavasti edelliseen tapaukseen verrattuna. Nyt korkeita viivepiikkejä on jo useampia. Lisäksi viiden sekunnin viivepiikkien määrä on lisääntynyt. Puskurin koko käy varsinkin simulaation loppupuoliskolla useasti melko korkealla. Huippuarvo on 7. Seitsemän paketin katoaminen peräkkäin tapahtuu noin todennäköisyydellä $0{,}40^7$*100% = 0,164%. Seitsemän pakettia riittänee tarpeeksi suurella varmuudella puskurin kooksi tällä pakettihukalla.



Kuva 27 HBH viive ja puskurin koko (30% pak.huk.)

5.1.11. HBH – 50% pakettihukka

Taulukko 12 Pakettistatistiikkaa (HBH 50% pak.huk.)

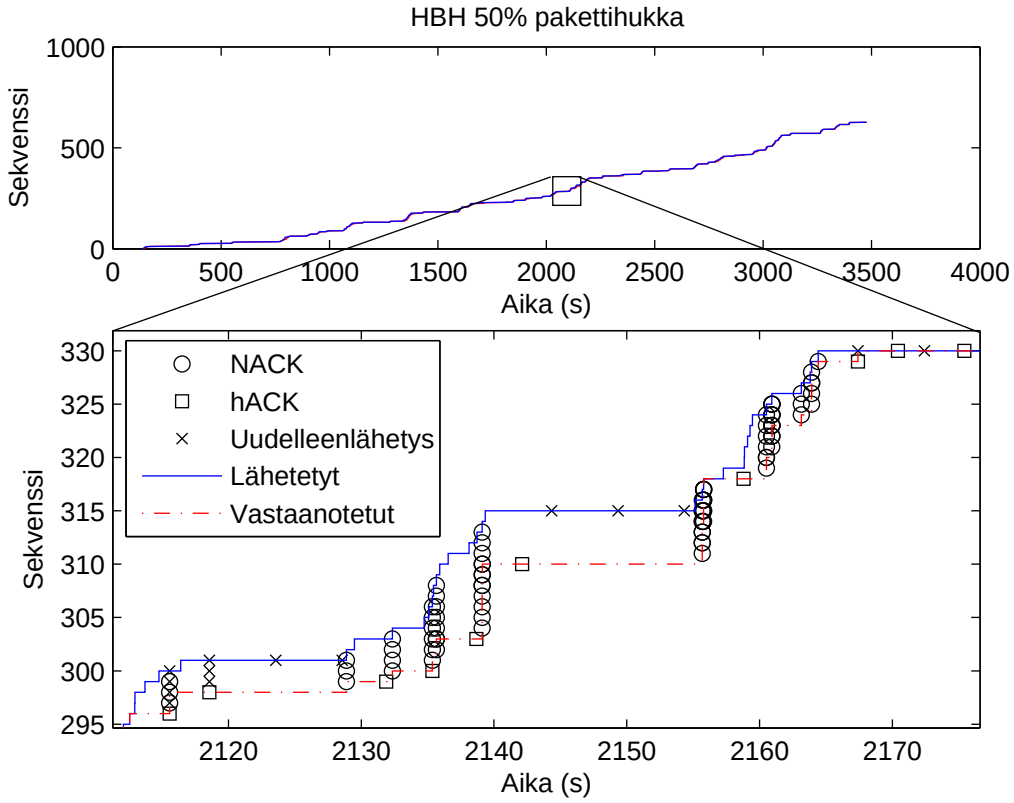
	HBH 1	HBH 2
Paketteja lähetty	1365	1445
Paketteja otettu vastaan	700	665
NACKeja lähetetty	859	893
hACKeja lähetetty	211	198
Uudelleenlähetys	279	340
Paketteja sekvenssissä	627	665

$$Overhead_{HBH1} = \frac{1365 + 893 + 198 - 627}{627} \times 100\% = 292\%$$

$$Overhead_{HBH2} = \frac{1445 + 859 + 211 - 665}{665} \times 100\% = 280\%$$

Luvuista käy ilmi, että 50% pakettihukalla menetelmän tehokkuus alkaa kärsiä toden teolla. Jos NACKeja ja vastauksia niihin lähetetään jo enemmän kuin sekvenssissä on paketteja, viiveet kasvavat väistämättä suuriksi.

Kuitenkin, kuten kuvassa 28 on esitetty, menetelmän avulla sekvenssit saavuttavat toisensa kuitenkin pikku hiljaa, joskin käytetty liikennemäärä on huikea jo tässä vaiheessa.

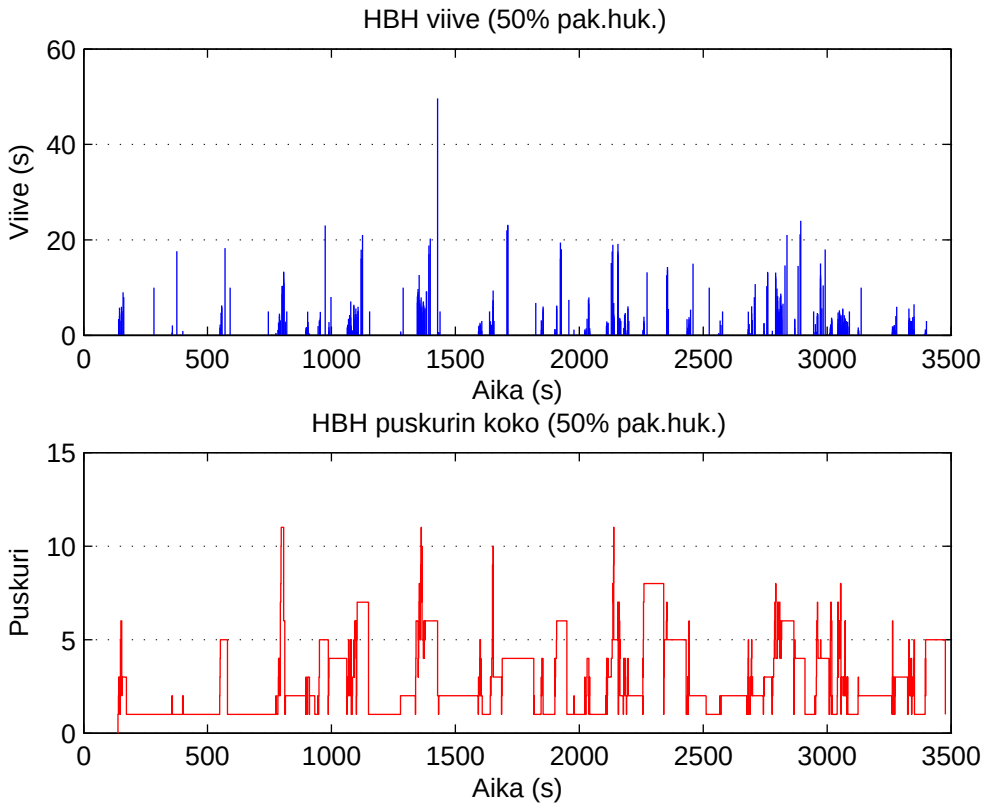


Kuva 28 HBH 50% pakettihukka

Viiveet kasvavat monessa kohtaa melko sietämättömiksi, pahimmillaan havaitaan jopa 50 sekunnin viivepiikki. Monessa kohtaa mennään myös 10 sekunnin huonommalle puolelle. Tätä selittää osittain myös pitkähkö uudelleenlähetysajastin.

Puskurin koko pyörii melko suurissa lukemissa myös, eikä se laske pieneksi kuin enää muutamassa kohtaa. Puskurin huippuarvo on 11, josta saadaan todennäköisyys 11:n paketin katoamiselle peräkkäin: $0{,}60^{11}$ *100% = 0,363%. Tällä pakettihukalla vaadittava puskurin koko liikkuu siis jossain 11:n ja 15:n välillä.

Kaiken kaikkiaan menetelmä alkaa olla 50% pakettihukalla aivan äärirajoilla. Seuraavasta tapauksesta nähdään, mitä tapahtuu kun liikennöintiolosuhteet käyvät täysin mahdottomiksi.



Kuva 29 HBH viive ja puskurin koko (50% pak.huk.)

5.1.12. HBH – 80% pakettihukka

Taulukko 13 Pakettistatistiikkaa (HBH 80% pak.huk.)

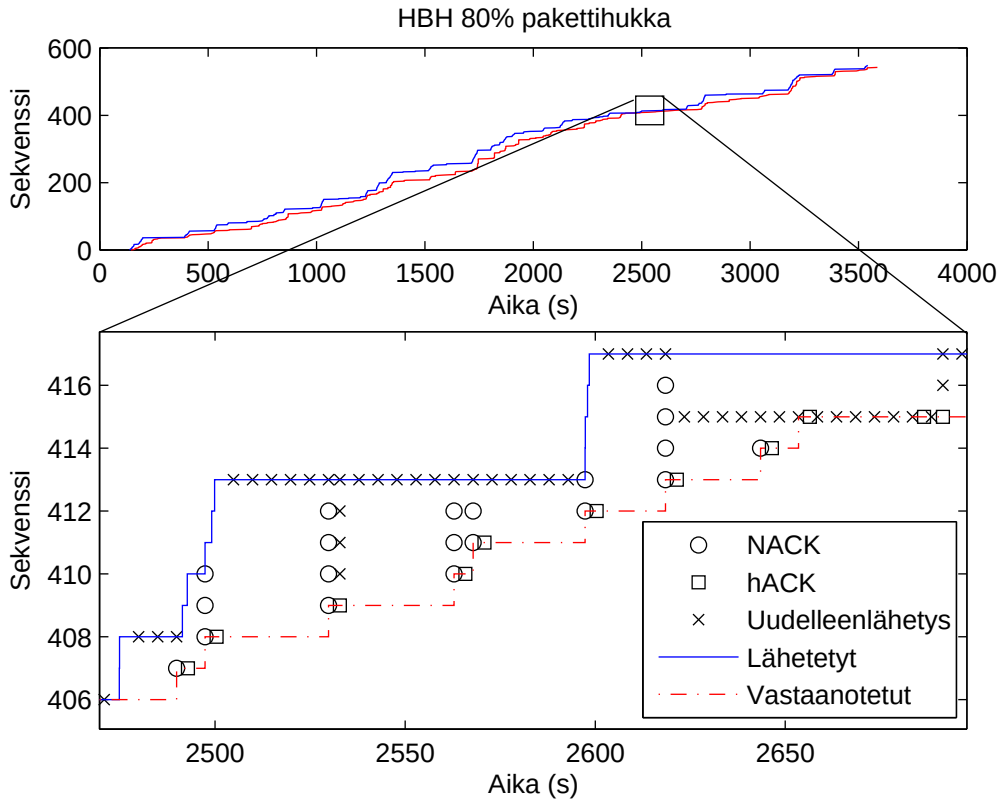
	HBH 1	HBH 2
Paketteja lähetty	2946	2794
Paketteja otettu vastaan	619	576
NACKeja lähetetty	5686	5729
hACKeja lähetetty	208	217
Uudelleenlähetys	1249	1082
Paketteja sekvenssissä	549	574

$$Overhead_{HBH1} = \frac{2946 + 5729 + 217 - 549}{549} \times 100\% = 1520\%$$

$$Overhead_{HBH2} = \frac{2794 + 5686 + 208 - 574}{574} \times 100\% = 1414\%$$

80% pakettihukalla menetelmän toimintakyky alkaa loppua. Syntyvän ylimääräisen hallintatiedon määrä on valtava. Jos hyötydatan lisäksi joudutaan lähettämään 15-kertainen määrä dataa eikä

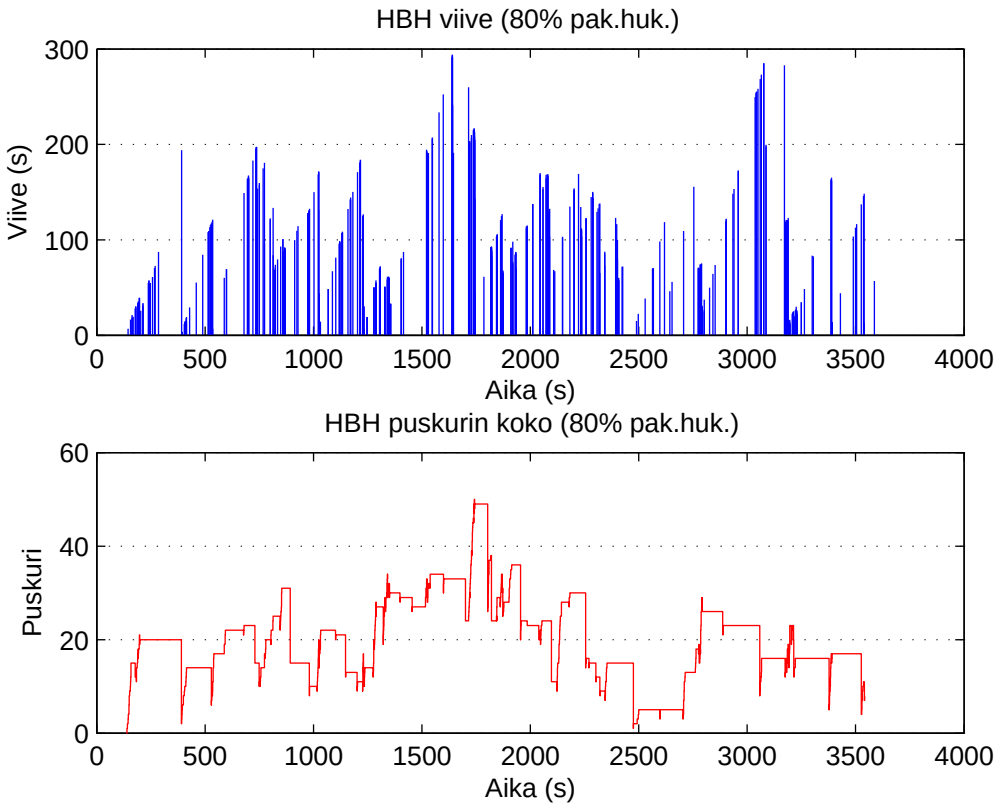
alkuperäinen data huonoimmassa tapauksessa päädy silti perille, voidaan turvallisesti todeta menetelmän hyödyllisyys ja stabiilius riittämättömiksi. Toisaalta 80% pakettihukka on jo sellainen tilanne, että oikeasti mikään liikenne ei menisi läpi. Tällaisessa tapauksessa häiriötä osuisi melko varmasti melkein jokaiseen pakettiin, joten mitään järkevää kommunikaatiota ei pystyisi tekemään.



Kuva 30 HBH 80% pakettihukka

Kuvassa 30 esiintyy 80% pakettihukan sekvenssikuvaaja. Tästä havaitaan, että pakettien läpimeno on hyvin rajallista, ja vastaanottopää on jatkuvasti huomattavasti jäljessä.

50% pakettihukalla viiveet olivat vielä siedettävyyden rajoissa, mutta kuten kuvasta 31 ilmenee, 80% pakettihukalla viiveet kasvavat mahdottomiksi. Myös puskurin koko liikkuu mahdottomissa lukemissa jatkuvasti.



Kuva 31 HBH viive ja puskurin koko (80% pak.huk.)

5.2. Yhteenveto

Taulukko 14 Yhteenveto eri pakettihukista

Pakettihukka	Hallintatieto	Viive(avg)	Viive(max)	Puskuri
0%	30%	120ms	2-5s	3
5%	50%	185ms	2-8s	4
10%	60%	290ms	3-8s	4
20%	90%	530ms	5-15s	5
30%	140%	1000ms	5-15s	7
50%	280%	3700ms	20-50s	15
80%	1500%	96s	N/A	N/A

Yhteenveto tuloksista on esitetty taulukossa 14. Hallintatieto kertoo ylimääräisen hallintatiedon määrän prosentteina hyötydatasta, viive(avg) kertoo keskimääräisen viiveen (laskettu), viive(max) kertoo tyypillisen viivehuipun ja puskuri ilmaisee, montako pakettia kyseisellä pakettihukalla olisi puskuroitava luotettavan toiminnan varmistamiseksi. 80% kohdalla N/A tarkoittaa sitä, että järkeviä arvoja ei ole käytettävissä. Maksimiviiveet olivat sietämättömiä ja puskurin koko liikkui myös täysin mahdottomissa arvoissa.

Tulosten perusteella voidaan havaita, että ilman mitään varmistusmenetelmää kasvava pakettihukka tuhoaa todella nopeasti realistiset mahdollisuudet minkäänlaiseen kommunikointiin. Vanilla-tapauksen tuloksista nähdään, että vastaanotettujen ja lähetettyjen pakettien käyrät eroavat toisistaan jo ilman lisättyä pakettihukkaa, ja lisätyn pakettihukan myötä ne eroavat vain entistä nopeammin.

HBH:n käyttö parantaa toimintaedellytyksiä huomattavasti. Suhteellisen pienillä pakettihukilla (0-30%) toiminta on erittäin stabiilia eikä ylimääräisen hallintatiedon määrä kasva dramaattisesti. Pakettihukasta palaudutaan nopeasti ja sekvenssit seuraavat toisiaan hyvin. 50% pakettihukalla menetelmä alkaa lähestyä äärirajojaan, vaikka kommunikointi vielä onnistuukin. Hallintatiedon määrä on tällä pakettihukalla kuitenkin jo melko merkittävä, joten kommunikointinopeuden ei voida olettaa olevan erityisen suuri. Toisaalta kuten jo työssä aiemmin mainittiin, prioriteettina on tiedonsiirron varmistaminen mahdollisesti palvelunlaadun kustannuksella.

80% pakettihukan tapauksessa todennäköisesti hyvin harva menetelmä pystyy kommunikoimaan täysin luotettavasti. Myös HBH on käytännössä toimintakyvytön tällaisella pakettihukalla sekä suuren hallintatiedon määrän että sietämättömän viiveen takia.

Kaiken kaikkiaan menetelmä toteuttaa asetetut vaatimukset tiedonsiirron luotettavuuden suhteen kiitettävästi pienemmillä pakettihukilla. Suuremmilla pakettihukilla tehokkuus kärsii huomattavasti samalla kun toiminta degeneroituu ja hallintatiedon määrä tukkii lopulta koko linkin. Se, voiko kyseisillä pakettihukilla (50-80% tai enemmän) toimia luotettavasti edes teoreettisesti, on asia erikseen.

5.2.1. Vaikutus TCP:n suorituskykyyn

Menetelmästä voidaan tulosten perusteella päätellä, että se parantaa luotettavuutta merkittävästi. Toinen näkökulma onkin se, miten menetelmä vaikuttaa ylemmän tason protokollien, esimerkiksi TCP:n, toimintaan. Tässä osassa tehdään kvalitatiivinen analyysi simuloinnista saatujen avainlukujen perusteella. Se tarkoittaa sitä, että toimintaa tarkastellaan teoreettisella tasolla tiettyjen perusolettamusten pohjalta.

TCP:n maksimisiirtonopeus saadaan kaavasta[15]

$$BW = \frac{MSS * \left[((1-p)/p) + w(p) + Q\{p, w\{p\}\}/(1-p)\right]}{RTT * \left[(w\{p\}+1)\right] + \frac{(Q\{p, w\{p\}\} * G\{p\} * T_0)}{1-p}} \quad (6)$$

missä BW on kaistanleveys, MSS on maksimisegmenttikoko, RTT on TCP:n näkemä edestakainen kulkuaikaviive, p on pakettihukka ja T_0 on uudelleenlähetyksen aikakatkaisu alussa (tyypillisesti 3 sekuntia, kuten ehdotettu RFC:issä 793 ja 1123). Lisäksi

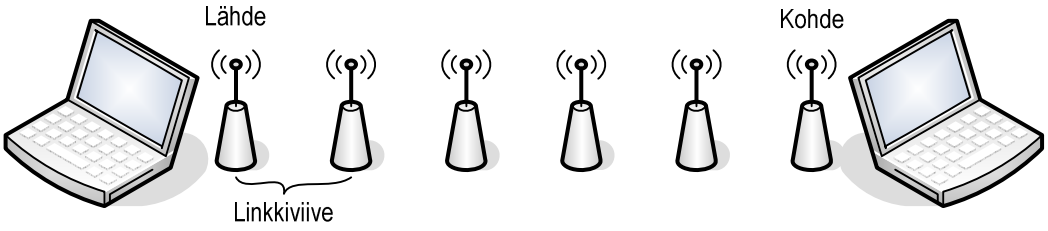
$$w\{p\} = (2/3)(1+sqrt(3*((1-p)/p)+1)) \quad (7)$$

$$Q\{p,w\} = min\{1,[(1-(1-p)^3)*(1+(1-p)^3)*(1-(1-p)^{(w-3)})]/[1-(1-p)^w]\} \quad (8)$$

$$G\{p\} = 1+p+2*p^2+4*p^3+8*p^4+16*p^5+32*p^6 \quad (9)$$

Kaavassa (6) oleva RTT ja sen vaihtelut vaikuttavat kaistanleveyteen. Mitä pienempi pakettihukka on, sitä suurempi vaikutus RTT:llä on maksimikaistaan. Esimerkiksi 0,1% pakettihukalla RTT:n tuplaantuminen karkeasti puolittaa kaistanleveyden. Näin pienellä pakettihukalla RTT:n suhde kaistanleveyteen on myös melko lineaarinen.

10% pakettihukalla sen sijaan RTT:n tuplaantuminen aiheuttaa paljon pienemmän pudotuksen kaistanleveyteen. Lisäksi havaitaan, että mitä suurempi RTT on, sitä enemmän kaistanleveys pienenee RTT:n tuplaantuessa. Esimerkiksi tuplaantuminen 100ms RTT:stä 200ms:iin aiheuttaa suhteessa pienemmän pudotuksen kaistanleveyteen kuin tuplaantuminen 800ms:stä 1600ms:iin. Toisaalta, vaikka suhteessa RTT kasvaa kummassakin tapauksessa yhtä paljon, absoluuttinen kasvu on paljon suurempi jälkimmäisessä tapauksessa.



Kuva 32 "Testiverkko"

Kuvassa 32 on esitetty testiasetelma. Tiedonsiirto tapahtuu siis kahden pisteen välillä, ja välissä on viiden hypyn verkko. RTT on linkkiviive kertaa 10, sillä RTT on edestakainen viive. HBH:n testauksessa on käytetty hidasta lähetysnopeutta ja liioitellun suuria hACK-ajastimia, koska siinä testattiin menetelmän toiminnallisuutta. Nostamalla tiedonsiirtonopeus kymmenkertaiseksi ja pienentämällä hACK-ajastimen aikaa kymmenesosaan saatiin keskimääräinen viivekin pudotettua karkeasti sanottuna noin kymmenesosaan. Tämä käyttäytyminen todennettiin simuloimalla pakettihukaton ja 10% pakettihukallinen skenaario uusilla arvoilla.

Laskennalliset tulokset TCP:n kaistanleveyden maksimille ilman HBH:n käyttöä ja sen kanssa eri nopeuksisille yhteyksille on esitetty taulukoissa 15-20. Maksimisegmenttikooksi on valittu tyypillinen 1460 tavua. Käytettävät linkkityypit ovat kiinteät linkit (1ms viive per hyppy), langattomat linkit (10ms viive per hyppy) sekä satelliittilinkit (300ms viive per hyppy). Satelliitin tapauksessa linkkejä oletetaan olevan vain kaksi. HBH:n kanssa teoreettiseksi TCP:n näkemäksi pakettihukaksi on valittu kaikissa tapauksissa 0,1%, sillä 0% pakettihukka tuottaisi kaavaan (6) nollalla jakamisen. Pakettihukka kertautuu useammalla hypyllä, jolloin kokonaispakettihukka saadaan kaavalla $1-(1-p)^n$, missä n on hyppyjen määrä ja p on linkkikohtainen pakettihukka.

Taulukko 15 TCP:n kaistanleveys kiinteillä linkeillä (1ms) ilman HBH:ta

Pakettihukka	Kokonaispakettihukka	Kaistanleveys
0.10%	0.50%	409184
5%	22.60%	2023
10%	41.00%	591
20%	67.20%	79
30%	83.20%	24
50%	96.90%	9
80%	100.00%	7

Kokonaiskulkuaikaviive 10ms

Taulukko 16 TCP:n kaistanleveys kiinteillä linkeillä (1ms) HBH:n kanssa

Pakettihukka	Kokonaisviive	Kaistanleveys
1%	0.13	289996
5%	0.19	201529
10%	0.3	129245
20%	0.54	72505
30%	1.01	38987
50%	3.71	10665
80%	96.01	412

TCP:n näkemä päästä päähän pakettihukka 0,1%

Taulukoista 15 ja 16 käy ilmi, että kiinteillä linkeillä ja pienellä pakettihukalla TCP toimii teoriassa tehokkaammin ilman HBH:ta. Tämä johtuu siitä, että HBH:n kanssa linkkikohtainen viive kasvaa käytännössä kymmenkertaiseksi (1ms vs 10ms). Erittäin pienellä pakettihukalla TCP toimii paljon tehokkaammin ilman HBH:tä, mutta jo hieman pakettihukkaa lisäämällä HBH tuo TCP:n toimintaan erittäin merkittävän nopeudenlisäyksen. Jopa 50% linkkikohtaisilla pakettihukilla voidaan vielä toimia, kun taas ilman HBH:ta TCP:n toiminta käytännössä loppuu linkkikohtaisen pakettihukan kasvaessa 5%:iin.

Taulukko 17 TCP:n kaistanleveys langattomilla linkeillä (10ms) ilman HBH:ta

Pakettihukka	Kokonaispakettihukka	Kaistanleveys
0.10%	0.50%	132131
5%	22.60%	1901
10%	41.00%	578
20%	67.20%	79
30%	83.20%	24
50%	96.90%	9
80%	100.00%	7

Kokonaiskulkuaikaviive 100ms.

Taulukko 18 TCP:n kaistanleveys langattomilla linkeillä (10ms) HBH:n kanssa

Pakettihukka	Kokonaisviive	Kaistanleveys
1%	0.22	174858
5%	0.28	138262
10%	0.39	99922
20%	0.63	62256
30%	1.1	35816
50%	3.8	10412
80%	96.1	412

TCP:n näkemä päästä päähän pakettihukka 0,1%.

Taulukoissa 17 ja 18 esitellyssä langattomien linkkien tapauksessa HBH:sta saadaan hyvin merkittävästi hyötyä jo aivan pienestä pakettihukasta lähtien. Kuten edellisessä tapauksessa, jo 5% linkkikohtainen pakettihukka hyydyttää TCP:n toiminnan täysin. HBH:n kanssa TCP toimii teoriassa tyydyttävästi vielä 20-30% pakettihukalla. Vasta 50%:ssa tapahtuu melko merkittävä suorituskyvyn lasku. 80%:lla toimintaa ei käytännössä ole, mutta tämä oli tulosten perusteella odotettavissakin.

Taulukko 19 TCP:n kaistanleveys satelliittilinkeillä (300ms) ilman HBH:ta

Pakettihukka	Kokonaispakettihukka	Kaistanleveys
0.10%	0.20%	45549
5%	9.75%	3398
10%	19.00%	1739
20%	36.00%	697
30%	51.00%	263
50%	75.00%	43
80%	96.00%	10

Kokonaiskulkuaikaviive 600ms.

Taulukko 20 TCP:n kaistanleveys satelliittilinkeillä (300ms) HBH:n kanssa

Pakettihukka	Kokonaisviive	Kaistanleveys
1%	0.720	64124
5%	0.780	59191
10%	0.890	51876
20%	1.130	40858
30%	1.600	28856
50%	4.300	10737
80%	96.600	478

TCP:n näkemä päästä päähän pakettihukka 0,1%.

Kuten edellisillä linkkityypeillä, myös satelliittilinkeillä saadaan merkittävä teoreettinen suorituskyvyn lisäys. HBH:n kanssa pystytään toimimaan jopa 50% pakettihukkaan asti, kun taas ilman varmistusta TCP:n toiminta heikkenee radikaalisti jo 5% pakettihukalla.

Yhteenveto

Kvalitatiivisen analyysin perusteella voidaan sanoa, että HBH parantaa TCP:n toimintaa kaiken tyyppisillä linkeillä huomattavasti, erityisesti kun pakettihukkaa alkaa esiintyä.

Nopeilla linkeillä ja vähäisellä pakettihukalla TCP toimii paremmin ilman HBH:ta, sillä HBH:n aiheuttama linkkikohtainen viive on kuitenkin huomattavasti suurempi kuin linkin oma siirtoviive. Toisaalta mainittakoon, että kiinteillä linkeillä harvemmin esiintyy kovin suuria pakettihukkia muualla kuin puskureissa.

Pakettihukat ovat olennainen osa TCP:n omaa lähetysnopeuden säätelyä viiveen ohella. Tämän vuoksi pakettihukan täydellinen poistuminen saattaa aiheuttaa sen, että useat TCP-vuot tukkivat koko verkon, koska lähetysnopeutta rajoittavia pakettihukkia ei tapahdu. Tässä vaiheessa HBH:n aiheuttama lisääntynyt hallintatiedon määrä ja viive kuitenkin rajoittanevat ”villiintyneitä” TCP-yhteyksiä ja liikenne palautuu lopulta normaalitilanteeseen. Tätä toimintaa estäisi mahdollinen eksplisiittinen ruuhkavaroitus, jolloin TCP osaisi hieman rajoittaa lähetysnopeuttaan, kun jonot alkavat täyttyä äärimmilleen. Toisaalta taas HBH yhdessä jonkin kapasiteettia mittaavan TCP-toteutuksen (esim. TCP-Jersey[8]) kanssa voisi tuottaa optimaalisen siirtotien käytön varmistamalla samalla linkkikohtaisen luotettavuuden.

Kaiken kaikkiaan menetelmä on erittäin lupaava keino TCP:n suorituskyvyn parantamiseen kun vähäistäkin pakettihukkaa esiintyy.

6. Johtopäätökset

Tämän diplomityön tavoitteena oli kehittää luotettava tiedonsiirtomenetelmä hybridiverkkoihin. Tämän tavoitteen pohjalta mallinnettiin hyppykohtaisiin kuittauksiin perustuva menetelmä nimeltään HBH (hop-by-hop). Mallinnus tehtiin OPNET Modeler ohjelmalla. Menetelmä perustuu hyppykohtaisiin sekvenssinumeroihin ja negatiivisiin kuittauksiin (NACK), joita täydentävät ajastimet kuittausta ja uudelleenlähetystä varten sekvenssien viimeisille paketeille. Menetelmän toteutus on riittävän yleispätevä, jotta se soveltuu sekä langattomille että langallisille linkeille.

6.1. Testaus

Menetelmää testattiin useilla eri pakettihukilla. Vertailun vuoksi testattiin myös perustapaus ilman menetelmän käyttöä. Tuloksissa verrattiin lähettyjen ja vastaanotettujen pakettien määrää ajan funktiona. Jos vastaanotettujen pakettien määrä seuraa lähetettyjen pakettien määrää, menetelmää voi pitää stabiilina. Tulosten perusteella menetelmä toimii erittäin hyvin, kun pakettihukka on vähäinen (noin 0-20%). Hieman suuremmalla pakettihukalla (20%-50%) menetelmä alkaa aiheuttaa jo huomattavan paljon ylimääräistä liikennettä kasvavien NACK-määrien muodossa, mutta toimii silti siedettävästi. Pakettihukan kasvaessa 50%:iin, saavutetaan piste, jossa menetelmä alkaa olla äärirajoillaan. Stabiilius saavutetaan, mutta ylimääräisen hallintatiedon määrä alkaa olla huomattava. 80% pakettihukalla menetelmä alkaa tuottaa jo niin paljon ylimääräistä hallintatietoa, että toiminta loppuu käytännössä täysin eikä menetelmä juuri enää stabiloidu.

Kvalitatiivisesta analyysista TCP:n toiminnasta HBH:n kanssa saatiin erittäin lupaavia tuloksia menetelmän tehokkuudesta. Erityisesti langattomilla linkeillä ja ylipäätään suuremmilla viiveillä toimivilla linkeillä HBH tehostaa TCP:n toimintaa merkittävästi. Ainoa tapaus, jossa ilman HBH:ta päästiin parempiin tuloksiin, oli pieniviiveinen (1ms) linkki pienellä pakettihukalla (<0,5%). Kaikissa muissa tapauksissa saatiin kohtalaisesta huomattavaan oleva (teoreettinen) kapasiteetin kasvu.

6.2. Tavoitteeseen pääsy ja parannuskohteet

Työn tavoitteeseen päästään siinä mielessä, että kohtuullisella pakettihukalla menetelmä parantaa tiedonsiirron luotettavuutta merkittävästi tuottamatta kuitenkaan tavatonta määrää ylimääräistä hallintaliikennettä. Parantamisen varaakin on, esimerkiksi suurempia pakettihukkia varten NACK-kuittaukset voisi koota yhteen pakettiin. Nyt korkeilla pakettihukilla todennäköisesti lähetetään samoista paketeista kuittauksia useampaan kertaan, mikä kasvattaa ylimääräisen tiedon määrää pahimmassa tapauksessa lähes eksponentiaalisesti. Esimerkiksi jos 10 pakettia katoaa peräkkäin, niistä lähetetään omat yksittäiset kuittaukset. Jos näiden avulla saadaan takaisin 2 pakettia, joudutaan uuden paketin saapuessa lähettämään taas 8 NACKia. On selvää, että kyseinen ratkaisu ei ole optimaalinen.

Toinen ominaisuus voisi olla kaistan ja/tai pakettihukan mittaus. Jos kohteessa esimerkiksi havaitaan, että välistä on jäänyt todella paljon paketteja, voitaisiin toimintamuotoa muuttaa lähetysnopeuden kustannuksella pakettikohtaiseen kuittaukseen. Käytännössä siis lähetettäisiin samaa pakettia, kunnes saadaan kuittaus, että se on saatu perille. Suuren pakettihukan tapauksessa tämä todennäköisesti rajoittaisi tiedonsiirtonopeuden murto-osaan linkin kapasiteetista. Kompromissi onkin tehtävä tiedonsiirtonopeuden ja luotettavuuden välillä.

Ratkaisussa ei myöskään oteta huomioon useamman hypyn päästä päähän toimintaa. Useamman hypyn toiminnassa eri lähteistä tulevat vuot olisi koostettava jatkuvaksi kokonaisuudeksi, joka välitettäisiin jälleen seuraavalle hypylle. Välistä ei saa puuttua paketteja, vaan puuttuvien pakettien pitää olla aina paikattavissa edelliseltä hypyltä. Tämän vuoksi olisi lähetyspuskurin lisäksi oltava olemassa vastaanottopuskuri, jossa sekvenssi kootaan jatkuvaksi ennen edelleenlähetystä.

6.3. Tulevaisuudennäkymät ja kehitysmahdollisuudet

Menetelmän toteuttamiseksi oikeilla laitteilla on jo mahdollisuudet, sillä IP-paketin tunnistekenttä (identification) on käytössä ainoastaan sirpaloitujen pakettien tunnistamisessa[14]. Jos sirpalointitoiminto kytketään pois päältä, kyseiseen kenttään voidaan leimata hyppykohtaisia sekvenssejä.

Yllä on mainittu joitakin puutteita, joita malli ei toistaiseksi toteuta. Näiden puutteiden korjaaminen olisi looginen seuraava askel. Puutteita olivat siis NACKien yhteen kokoamisen puute, kapasiteetin mittauksen puute ja usean hypyn toiminnan toteutuksen puuttuminen. Puutteiden korjaamisen jälkeen olisi edessä menetelmän toteuttaminen käytännön tasolla protokollana tai toimintapolitiikkana oikeissa laitteissa.

7. Lähdeluettelo

[1] C.Wan, A.T.Campbell, L.Krishnamurthy: Pump-Slowly, Fetch-Quickly (PSFQ): A Reliable Transport Protocol for Sensor Networks, IEEE Journal on Selected Areas in Communications, April 2005.

[2] F.Stann, J.Heidemann: RMST: Reliable Data Transport in Sensor Networks, IEEE Int. Workshop on Sensor Network Protocols and Applications 2003.

[3] Chalermek Intanagonwiwat et al: Directed Diffusion for Wireless Sensor Networking, IEEE/ACM Transactions on Networking, Vol. 11, Feb. 2003.

[4] Ö.B.Akan, I.F.Akyildiz: Event-to-Sink Reliable Transport in Wireless Sensor Networks, IEEE/ACM Transactions on Networking, Oct. 2005.

[5] H.Lee, Y.Ko, D.Lee: A Hop-by-hop Reliability support Scheme for Wireless Sensor Networks, IEEE Pervasive Computing and Communications Workshops, 2006 (PERCOMW'06).

[6] R.Kumar, A.Paul, U.Ramachandran, D.Kotz: On Improving Wireless Broadcast Reliability of Sensor Networks Using Erasure Codes, INFOCOM 2005, Georgia Institute of Technology.

[7] D.J.C.MacKay: Fountain Codes, IEE Proceedings Communications, Vol. 152, Dec. 2005.

[8] K.Xu, Y.Tian, N.Ansari: TCP-Jersey for Wireless IP Communications, IEEE Journal on Selected Areas in Commuications, May 2004.

[9] Claudio Casetti et al: TCP Westwood: End-to-End Congestion Control for Wired/Wireless Networks, Wireless Networks 8, 2002.

[10] D.D.Clark, W.Fang: Explicit allocation of best effort packet delivery service, IEEE/ACM Trans. Networking, Vol. 6, Aug. 1998.

[11] D.Sun, H.Man: ENIC – An Improved Reliable Transport Scheme for Mobile Ad Hoc Networks, IEEE Global Telecommunications Conference, 2001.

[12] K.Sundaresan, V.Anantharaman, H.Hsieh, R.Sivakumar: ATP: A Reliable Transport Protocol for Ad Hoc Networks, IEEE Transactions on Mobile Computing, Nov.-Dec. 2005.

[13] M.K. McKusick, G.V. Neville-Neil: The Design and Implementation of the FreeBSD Operating System, 2005, Addison-Wesley.

[14] RFC 791, 1981. Verkossa: http://www.faqs.org/rfcs/rfc791.html

[15] Mathis, Semke, Mahdavi, Ott: The macroscopic behavior of the TCP congestion avoidance algorithm, Computer Communication Review, 27(3), July 1997.

[15] J. Padhye, V. Firoiu, D. Townsley, J. Kurose: Modelling TCP throughput: A simple model and its empirical validation, Proc. SIGCOMM Symp. Communications Architectures and Protocols, August 1998.

8. Liite A Pseudokoodi

```
/*************************************/
process outbound packet  (source_addr, dest_addr, packet*)
        {
        iterate the destination sequences () {

                if (destination address entry found) {
                        increase sequence;
                }
        }
        if (correct sequence not found) {
                create new entry();
                initialize new entry();
        }

        Set packet attributes (type, source address, destination address, sequence);

        start resend timer (packet);

        increase packets sent counter;
        write the counter value to statistics();

        get sequence number from neighbor();

        calculate sequence difference and add to statistics();

        SEND PACKET (out stream to MAC);
        }

/*************************************/
process inbound packet (Packet* pkptr)
        {

        generate random number();
        if (random number < loss_percent) {
                send packet to sink );
                end function;
        }
        get packet type, source address, destination address, sequence ();

        if (type equals hack) {
                stop resend timer();
                check sequence from hack();
                send packet to sink ();
                end function;
        }
        if (type equals nack) {
                process nack ();
                send packet to sink ();
                end function;
        }
        iterate the source sequences() {
```

```
                    if (source address entry found) {
                            if (entry sequence < sequence) {
                                    increase entry sequence;
                                    update statistics();
                            }
                            if (entry sequence (after increase) < sequence) {
                                    send nack (to source);
                            }
                    }
            if (entry not found) {
                    create new entry();
                    initialize new entry();
                    }

            start timer for hack ();

            SEND PACKET (sink);
            }

/*****************************/
send nack (source, sequence)
            {
            calculate how many packets missing();
            for (go through loop until the right amount of nacks sent) {
                    create the packet and set attributes (type, source address, destination address, sequence);
                    SEND PACKET (outstream to MAC);
                    update statistics(nacks sent);
                    }
            }

/***********************************/
process nack ()
            {
            create a new packet with the same attributes as the missing packet();
            SEND PACKET (outstream to MAC);
            update statistics(packets sent);
            }

/*********************************/
start resend timer ()
{
            cancel pending resend ();
            schedule new resend();
            save a pointer to the resend event ();
}

/*******************************/
stop resend timer ()
{
            stop the pending resend ();
}

/*******************************/
start timer for hack ()
{
            cancel pending hack ();
            schedule new hack ();
            save a pointer to the hack event ();
}

/*********************************/
resend ()
{
            create a packet ();
            set attributes to match the missing packet ();
            SEND PACKET (outstream to MAC);
            update statistics();
}

/*******************************/
```

```
send_hack ()
{
	create a hack packet ();
	set attributes();
	SEND PACKET (outstream to MAC);
}

/*********************************/
check sequence from hack ()
{
	iterate destination list {
		if (entry found) {
			if (hacked sequence < entry sequence) {
				resend missing packets ();
			}
		}
	}
}

/************************************/
resend missing packets ()
	{
	calculate how many packets missing();
	for (go through loop until the right amount of packets sent) {
		create the packet and set attributes (type, source address, destination address, sequence);
		SEND PACKET (outstream to MAC);
		update statistics();
	}
}
```