Antti Ruohisto

Luotettavan UDP-pohjaisen protokollan toteutus

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

11.5.2016

Metropolia

Tiivistelmä

Tekijä(t) Otsikko Sivumäärä Aika	Antti Ruohisto Luotettavan UDP-pohjaisen protokollan toteutus 30 sivua 11.5.2016
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Miikka Mäki-Uuro

Tämän insinöörityön tavoitteena oli luoda pelikäyttöön soveltuva verkkoliikenneprotokolla. Protokollan tulisi pystyä tarvittaessa varmistamaan pakettien perillemeno, järjestys ja tarjoamaan helppo tapa luoda nopea yhteys koneiden välille. Protokollaa tullaan myöhemmin hyödyntämään peliprojekteissa.

Insinöörityössä valittiin sopiva protokolla oman protokollan pohjaksi. Tekstissä käydään läpi IP-, TCP- ja UDP-protokollien tuomat ominaisuudet sekä niiden heikkoudet ja vahvuudet. Lisäksi tutkitaan TCP- ja UDP-protokollien käyttöä rinnakkain. Lopulta päädytään käyttämään oman protokollan pohjana UDP-protokollaa.

UDP-protokollan päälle suunniteltiin seuraavat ominaisuudet: saapuvien pakettien tunnistaminen, kadonneen paketin tunnistaminen, pakettien pitäminen järjestyksessä, kadonneiden pakettien uudelleenlähetys, duplikaattipakettien tunnistaminen, virtuaaliyhteyden luominen ja ylläpitäminen sekä ruuhkanhallinta. Suunnitelmien pohjalta toteutettiin tarvittavat ominaisuudet. Työssä hyödynnettiin myös Boost.Asio-kirjastoa UDP-pakettien lähettämiseen ja vastaanottamiseen.

Insinöörityön lopputuloksena syntyi toimiva prototyyppi luotettavasta UDP-pohjaisesta protokollasta. Lisäksi työssä myös esitellään, kuinka protokollan avulla voidaan luoda asiakaspalvelin-mallin mukainen rakenne.

Avainsanat	Luotettava UDP, TCP, verkkokommunikointi

Metropolia

Author(s) Title Number of Pages Date	Antti Ruohisto Implementation of Reliable UDP Based Protocol 30 pages 11 May 2016
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Miikka Mäki-Uuro, Senior Lecturer

The goal of this thesis was to create a suitable network protocol for games. The protocol should be able to, if necessary, ensure the successful delivery of packets, keep the order of sent packets and offer a simple way to create a fast connection between two computers. This protocol will later be used in game development projects.

In this thesis an appropriate protocol was chosen to be the basis of the protocol. The text goes though the properties of IP, TCP and UDP as well as their weaknesses and strengths. Also the possibility of using TCP and UDP in parallel is explored. In the end UDP was chosen as the base of the protocol.

The following properties were designed to be implemented on top of the UDP: recognizing incoming packets, identifying lost packets, maintaining the proper sequence of the packets, resending lost packets, detecting duplicate packets, creating and maintaining virtual connection and flow control. After the design process, all of the necessary features were implemented. The thesis also utilized Boost.Asio library to send and receive UDP packets.

The result of the project was a functioning prototype of a reliable UDP based protocol. In addition, the thesis also presents how to create a client-server model using the protocol.

Keywords	Reliable UDP, TCP, network communication

Metropolia

Sisällys

Lyhenteet

1 Johdanto 1

2 Protokollan valitseminen 2

2.1 IP 2
2.2 TCP 3
2.3 UDP 5
2.4 TCP- ja UDP-protokollan yhdistäminen 6

3 Oman protokollan suunnittelu 6

3.1 Saapuvien pakettien tunnistaminen 7
3.2 Pakettien luotettavuus ja järjestys 7
3.2.1 Pakettien kuittaus 7
3.2.2 Kadonneen paketin tunnistaminen ja uudelleenlähettäminen 8
3.2.3 Duplikaattipaketit 9
3.2.4 Pakettien järjestys 9
3.3 Virtuaaliyhteys 9
3.3.1 Virtuaaliyhteyden muodostaminen 10
3.3.2 Yhteyden ylläpitäminen ja sulkeminen 12
3.3.3 Protokollan otsikkotiedot 12
3.3.4 Ruuhkanhallinta 13

4 Protokollan toteutus 14

4.1 Pakettien lähettäminen ja vastaanottaminen 14
4.1.1 Ohjelmointikieli ja Boost.Asio-kirjasto 14
4.1.2 UDP-soketin avaaminen 15
4.1.3 UDP-pakettien lähettäminen ja vastaanottaminen 15
4.1.4 UDP-paketin lähettäminen 15
4.1.5 UDP-paketin vastaanottaminen 16
4.2 Yhteyden tila 17
4.3 Sekvenssinumerot 18
4.4 Pakettien kuittaus 19
4.5 Paketin kuittauksen käsittely 20
4.6 Paketin uudelleenlähettäminen 20
4.7 Ruuhkanhallinta 21

4.8 Datan lähetys Connection-luokan avulla 23
4.9 Datan vastaanotto Connection-luokan avulla 23

5 Asiakas-palvelin-malli 23

5.1 Asiakas-palvelin-mallin rakenne 24
5.2 Client (asiakas) 25
5.2.1 Yhteyden luonti 25
5.2.2 Pakettien lähettäminen ja vastaanottaminen 26
5.2.3 Yhteyden sulkeminen. 26
5.3 Server (palvelin) 26
5.3.1 Virtuaaliyhteydet 26
5.3.2 Pakettien vastaanottaminen 27
5.3.3 Pakettien lähettäminen 27

6 Yhteenveto 28

Lähteet 30

Lyhenteet

IP	Internet Protocol. Verkkokerroksen protokolla, jonka avulla verkon koneet voivat kommunikoida keskenään.
TCP	Transmission Control Protocol. Yhteydellinen protokolla, jonka avulla tietokoneet voivat lähettää toisilleen tavujonoja luotettavasti ja oikeassa järjestyksessä.
UDP	User Datagram Protocol. Yhteydetön protokolla, joka mahdollistaa tiedonsiirron koneiden välillä.
HTTP	Hypertext Transfer Protocol. Sovelluskerroksen protokolla, jota selaimet ja WWW-palvelimet käyttävät tiedonsiirtoon.
DNS	Domain Name System. Nimipalvelujärjestelmä, joka muuntaa verkkotunnuksia IP-osoitteiksi.
ACK	Acknowledgement. Kuittaus, joka lähetetään aina, kun vastaanotetaan paketti.
RTT	Round-trip time. Aika joka kuluu paketin lähetyksestä kuittaukseen.

1 Johdanto

Tämän insinöörityön tarkoituksena oli luoda erityisesti pelikäyttöön soveltuva verkkoliikenneprotokolla. Tavoitteena oli luoda protokolla, joka tarjoaa helpon tavan luoda koneiden välille mahdollisimman nopea ja matalaviiveinen yhteys. Protokollaa tullaan myöhemmin hyödyntämään peliprojekteissa.

Insinöörityö on jaettu neljään pääosaan: ensimmäisenä tutkitaan olemassa olevia protokollia, joiden ominaisuuksia voidaan hyödyntää, toisena käydään läpi oman protokollan suunnitteluun vaikuttaneita tekijöitä, kolmantena käydään läpi protokollan toteuttaminen ja lopuksi selitetään lyhyesti mikä on asiakas-palvelin-malli, sekä käydään läpi luokat, jotka toteuttavat sen vaatimat ominaisuudet.

Luvussa 2 keskitytään valitsemaan sopivin olemassa oleva protokolla, jota tullaan hyödyntämään oman protokollan toteuttamisessa. Kappaleessa käydään läpi IP-protokollan tuomat ominaisuudet sekä perehdytään TCP- ja UDP-protokollien toimintaan hieman tarkemmin. Lopuksi vielä tutkitaan mahdollisuutta käyttää TCP- ja UDP-protokollia samanaikaisesti.

Oman protokollan suunnitteluosuudessa kuvataan tarvittavat ominaisuudet omaa protokollaa varten. Kappaleessa paneudutaan erityisesti pakettien luotettavuuteen ja järjestyksessä pitämiseen, virtuaaliyhteyden luomiseen ja ylläpitämiseen sekä ruuhkanhallintaan.

Protokollan toteutusluvussa keskitytään suunnitteluvaiheessa kuvattujen ominaisuuksien varsinaiseen toteutukseen. Luvussa myös valitaan käytetty ohjelmointikieli sekä hyödynnetyt kolmannen osapuolen kirjastot. Toteutuksen kuvauksessa käytetään apuna useita koodiesimerkkejä protokollasta ja ominaisuuksien toteuttamisesta pyritään kuvaamaan pääpiirteet.

Asiakas-palvelin-malli osiossa selitetään lyhyesti, mikä asiakas-palvelin-malli on. Tämän jälkeen esitellään tässä insinöörityössä toteutetut luokat, joiden avulla protokollan käyttäjä voi helposti luoda asiakas-palvelin-mallin mukaisen verkkoviestintärakenteen.

2 Protokollan valitseminen

Tarkoituksena oli tehdä helppokäyttöinen protokolla, jonka avulla käyttäjä voi helposti toteuttaa moninpelin vaatiman kommunikoinnin palvelimen ja asiakasohjelman välille. Protokollan avulla täytyi pystyä luomaan virtuaaliyhteys palvelimen ja asiakasohjelman välille. Lisäksi protokollan täytyi pystyä lähettämään ja vastaanottamaan viestejä sekä takaamaan, että viesti saapuu perille, jos käyttäjä niin haluaa.

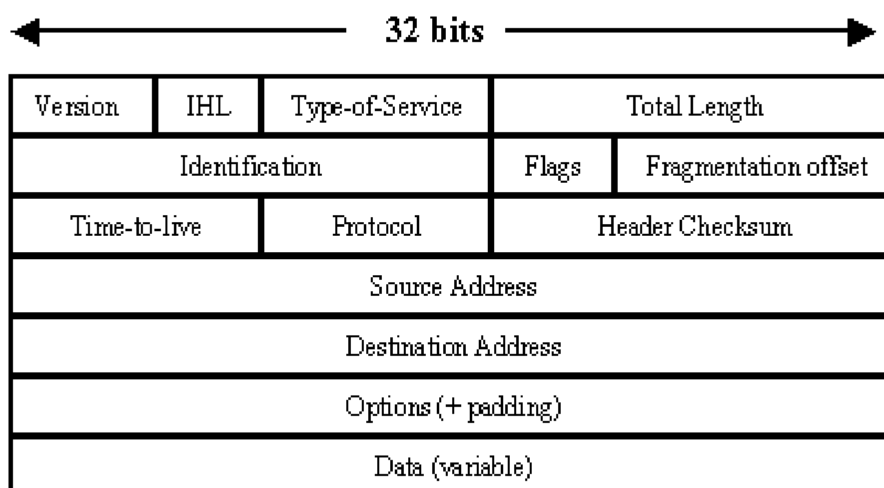
Verkkokerroksen protokollana käytettiin IP-protokollaa, mutta kuljetuskerroksen protokollana oli kaksi hyvää vaihtoehtoa: TCP (Transmission Control Protocol) ja UDP (User Datagram Protocol). Seuraavissa aliluvuissa käydään lyhyesti läpi, mitä ominaisuuksia internetprotokolla, TCP- ja UDP-protokolla tuovat mukanaan. Lisäksi käydään läpi myös TCP- ja UDP-protokollien vahvuudet ja heikkoudet, sekä tutkitaan mahdollisuutta käyttää molempia protokollia samanaikaisesti.

2.1 IP

Molemmat kuljetuskerroksen protokollat, joita harkittiin, toimivat lähes aina matalamman tason internetprotokollan päällä. IP eli Internet protocol tarjoaa epäluotettavan, yhteydettömän paketin toimituksen. Epäluotettavalla tarkoitetaan sitä, että IP-protokolla ei tee minkäänlaisia takuita siitä, että IP-paketit varmasti saapuvat perille. Yhteydetön tarkoittaa sitä, että jokainen paketti lähetetään ja käsitellään erillisinä paketteina. Peräkkäin lähetetyt paketit saatetaan reitittää eri reitittimien kautta ja saattavat myös saapua kohdekoneelle eri järjestyksessä kuin ne lähetettiin. [1; 3.]

IP-protokolla hoitaa myös pakettien reitityksen isäntäkoneelta kohdekoneelle. IP-paketissa on kerrottu kohdeosoite, jonka avulla yksilöidään verkossa olevat koneet. Tätä varten reitittimillä on reititystaulut, jotka sisältävät listan reiteistä eri kohteisiin tietoverkoissa. Nämä taulut siis sisältävät tiedon verkon topologiasta. Reititintaulun avulla reititin pystyy ohjaamaan IP-paketit oikeaan suuntaan. [7; 8.]

IP-paketin otsikkotiedot vievät 20 tavua ilman optioita. Kokonaisuudessaan otsikkotiedot sisältävät 12 pakollista kenttää (ks. kuva 1).

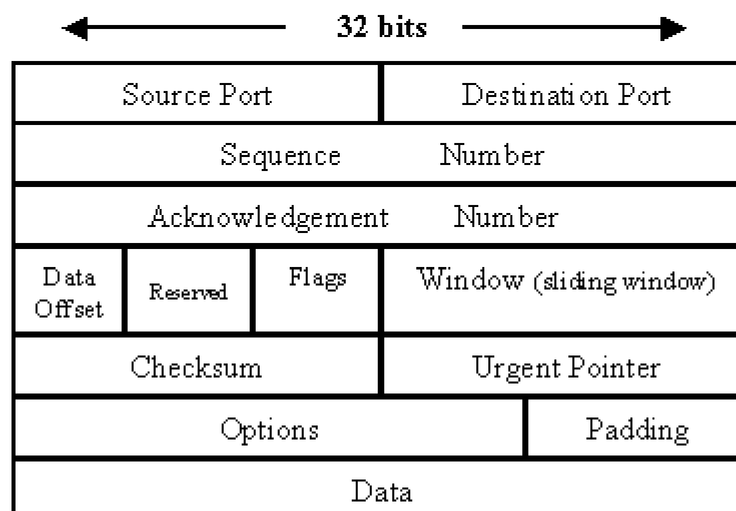


Kuva 1. IP-paketin otsikkotiedot [9.]

2.2 TCP

TCP eli transmimssion control protocol toimii IP-protokollan päällä ja on nykyisen internetin kulmakivi, sillä suurin osa internetin liikenteestä perustuu TCP-protokollaan. Esimerkiksi HTTP-yhteys muodostetaan TCP-protokollan avulla. TCP-protokollan avulla tietokoneet pystyvät lähettämään toisillensa tavujonoja luotettavasti, ja TCP-protokolla osaa myös jakaa suuret datamäärä sopivan kokoisiin paketteihin automaattisesti. [1.]

TCP muodostaa virtuaaliyhteyden koneiden välille ja pitää huolen, että kaikki paketit saapuvat perille oikeassa järjestyksessä. Tämän takia jokaisessa TCP-protokollan lähettämässä paketissa on sekvenssinumero, jonka avulla voidaan havaita puuttuvat paketit sekä myös duplikaatti paketit. Lisäksi TCP-paketti pitää sisällään 16-bittisen tarkistussumman, jonka avulla voidaan varmistua datan ehjyydestä. TCP-protokolla hoitaa myös ruuhkanhallinnan. Ruuhkalla useimmiten tarkoitetaan ruuhkaa internetin reitittimillä, joka syntyy siitä, kun reitittimelle saapuu enemmän paketteja, kuin se pystyy lähettämään, ja ylimääräiset paketit joudutaan hylkäämään.[1.]



Kuva 2. TCP-paketin otsikkotiedot [9.]

TCP vaikutti ominaisuuksiltaan varsin hyvältä ja helppokäyttöiseltä. Tarkempi tarkastelu kuitenkin paljasti merkittäviä ongelmia, jos vasteaika on tärkeä ohjelman toimivuuden ja käytettävyyden kannalta.

TCP on tietovirtapainotteinen protokolla, mikä tarkoittaa sitä, että TCP-protokollaa käyttäessä kirjoitetaan tietovirtaan ja TCP-protokolla hoitaa datan lähetyksen vastaanottajalle. Koska TCP-protokolla toimii IP-protokollan päällä, data täytyy pilkkoa paketteihin ennen lähetystä. Tästä voi seurata ongelmia, jos käyttäjä haluaa lähettää hyvin pieniä paketteja, sillä TCP-protokolla saattaa odottaa suurempaa määrää dataa, kunnes se lähettää paketit eteenpäin. Tämä voi aiheuttaa viivettä. Tämänkaltainen toiminta lisää kaistan tehokkuutta latenssin kustannuksella. Tämä saattaa vaikuttaa hyvin negatiivisesti esimerkiksi reaaliaikaisissa peleissä, sillä tarkoituksena on saada käyttäjän antama paketti vastaanottajalle mahdollisimman nopeasti ja useimmiten myös tasaisin väliajoin. [3.]

TCP-protokolla tarjoaa onneksi mahdollisuuden asettaa TCP_NODELAY valinnan, joka lähettää paketit heti, kun ne annetaan TCP-protokollan hoidettavaksi. Tätä myös kutsutaan Naglen algoritmin sammuttamiseksi. [1.]

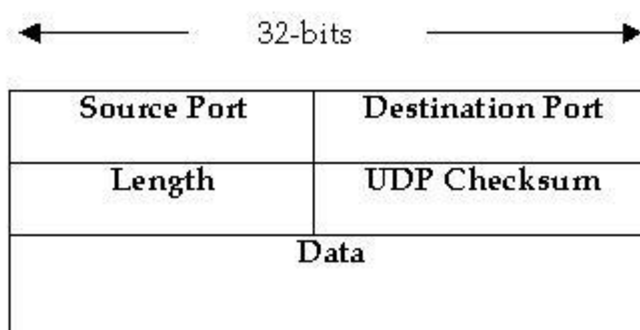
Todellinen ongelma, jonka TCP-protokolla aiheuttaa, on sen tapa toteuttaa luotettava ja järjestyksellinen paketin toimitus. TCP-protokolla muuttaa tavujonon sopivan kokoisiksi paketeiksi, lähettää ne IP-protokollan avulla vastaanottajalle, ja vastaanottaja uudelleenrakentaa vastaanotetuista paketeista tavujonon. Jos paketteja huomataan puuttuvan, joutuu TCP-protokolla odottamaan näitä paketteja jatkaakseen tavujonon rakentamista.

TCP-protokolla ei käsittele uudempia paketteja, vaikka sellaisia olisi tarjolla, jos aikaisempia paketteja puuttuu. Monissa sovelluksissa on tärkeää saada aina viimeisin tieto, esimerkiksi objektin tila nopeatempoisessa verkkopelissä tai reaaliaikaista ääntä tai kuvaa lähetettäessä. [3.]

2.3 UDP

UDP (User Datagram Protocol) on yhteydetön protokolla ja sisältää huomattavasti vähemmän ominaisuuksia kuin TCP-protokolla. Koska UDP-protokolla tarjoaa vähemmän ominaisuuksia, on se yleensä myös nopeampi ja sitä käytetäänkin usein reaaliaikaisen videon ja äänen välittämiseen sekä myös DNS-pyyntöjen lähettämiseen. TCP-protokollan tavoin, UDP-protokollakin toimii yleensä IP-protokollan päällä.[11.]

UDP-protokolla ei varmista paketin perillemenoa ja paketit myös saattavat saapua vastaanottajalle eri järjestyksessä kuin ne alun perin lähetettiin. UDP-paketti pitää kuitenkin sisällään vapaaehtoisen (IPv6-protokollassa pakollinen) tarkistussummakentän, jonka avulla voidaan varmistua datan ehjyydestä. UDP-protokolla on siis hyvin ohut kerros internetprotokollan päällä. Se sisältää ainoastaan tarkistussumman, datan koon sekä lähde- ja kohdeosoitteen portit, joista lähdekohteen portti ja tarkistussumma ovat vapaaehtoisia (ks. kuva 3) [9.]. Lähde- ja kohdeportit ovat sovelluksen kannalta tärkeimpiä asioita, joita UDP-protokolla tuo mukanaan, jotta paketit voidaan helposti osoittaa oikeille sovelluksille.



Kuva 3. UDP-paketin otsikkotiedot [9.]

UDP-protokolla ei siis suoraan tarjoa ominaisuuksia, joita tarvitaan protokollan toteuttamiseen. Se ei kuitenkaan ole huono asia, sillä UDP-protokollan päälle voidaan luoda

oma protokolla. Tällä tavoin pystyttiin luomaan tarvittavat ominaisuudet ilman TCP-protokollan tuomia ongelmia.

2.4 TCP- ja UDP-protokollan yhdistäminen

Eräänä vaihtoehtona voisi myös olla molempien protokollien käyttäminen rinnakkain. Tällä tavoin molemmista protokollista voitaisiin hyödyntää niiden edut. Data, jonka täytyy saapua luotettavasti ja järjestyksessä perille, käytetään TCP-protokollaa, ja data, joka täytyy mahdollisimman nopeasti toimittaa vastaanottajalle, voisi hyödyntää UDP-protokollaa.

On useita ohjelmia ja pelejä, jotka myös hyödyntävät tämän kaltaista tekniikkaa, mutta sillä on myös haittapuolensa. Kun TCP- ja UDP-protokollia käytetään samanaikaisesti ohjelmassa, siinä joudutaan luopumaan jonkin verran protokollan omasta hallinnasta. TCP-protokollasta ei voi esimerkiksi sammuttaa ominaisuuksia (muutamaa poikkeusta lukuun ottamatta), kun niitä ei tarvita, joten on järkevämpää toteuttaa tarvittavat ominaisuudet UDP-protokollan päälle. Toisena syynä olla sekoittamatta protokollia, on riski lisääntyneeseen paketin katoamiseen. TCP-protokollan vuonvalvonta- ja ruuhkanhallintamekanismit saattaa lisätä UDP-pakettien katoamista [10.].

3 Oman protokollan suunnittelu

Koska TCP-protokollan tapa toteuttaa luotettava ja järjestyksellinen paketin toimitus voi aiheuttaa hetkellisiä piikkejä vasteajassa eikä UDP-protokolla tee minkäänlaisia takauksia datan perillemenosta tai järjestyksestä, ei kumpaakaan protokollaan voitu suoraan käyttää sellaisenaan. Kuten aikaisemmin mainittiin, voidaan UDP-protokollan päälle rakentaa oma protokolla, joka toteuttaa vaadittavat toiminnot protokollaa varten. Protokollan tulisi toteuttaa seuraavat toiminnot: saapuvien pakettien tunnistaminen, kadonneiden pakettien uudelleen lähetys, pakettien pitäminen järjestyksessä, ruuhkanhallinta, duplikaatti pakettien tunnistaminen ja virtuaaliyhteyden luominen.

3.1 Saapuvien pakettien tunnistaminen

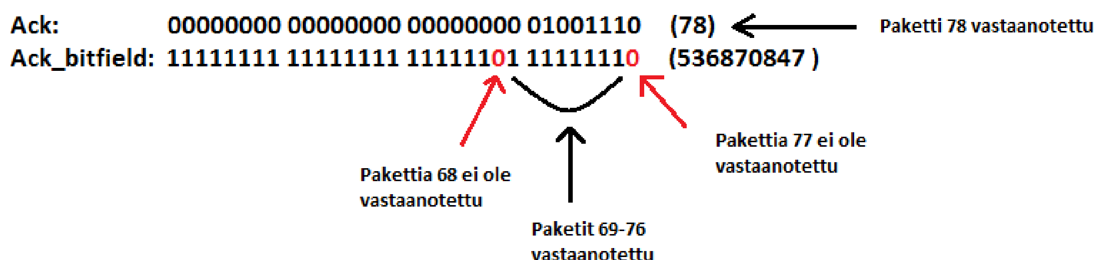
Protokollan tarvitsee myös tunnistaa, mitkä paketit on tarkoitettu sovellukselle, joka hyödyntää luotavaa protokollaa. Tätä varten protokollan otsikkotietoihin luodaan 32 bittinen protocol_id-kenttä. Tämä kenttä voidaan määrittää sovelluksessa, ja kaikki lähetettävät paketit merkitään annetulla 32-bittisellä arvolla ja paketteja vastaanotettaessa tarkistetaan paketin protocol_id. Jos paketin protocol_id vastaa aikaisemmin annettua arvoa, paketti päästetään läpi. Muussa tapauksessa paketti voidaan poistaa.

3.2 Pakettien luotettavuus ja järjestys

Tulevissa aliluvuissa esitellään paketin luotettavuuden ja järjestyksen kannalta olennaisia ominaisuuksia. Aluksi kerrotaan paketin kuittauksesta. Sen jälkeen kadonneen paketin tunnistamisesta ja uudelleenlähettämisestä, lopuksi vielä kerrotaan duplikaattipakettien tunnistamisesta ja pakettien järjestyksestä.

3.2.1 Pakettien kuittaus

Oman protokollan tulisi pystyä tarvittaessa takaamaan datan perillemeno. Tähän voidaan ottaa mallia TCP-protokollan tavasta lähettää jokaisen paketin mukana myös kuittaus saapuneista paketeista. Tätä varten oman protokollan otsikkotietoihin tarvitaan sekvenssinumero (sequence number) sekä kuittaus (acknowledgement) saapuneista paketeista. Molemmat kentät ovat 32-bittisiä etumerkittömiä kokonaislukuja. Tämä itsessään ei kuitenkaan riitä pakettien kuittaamiseksi, koska paketti, joka sisältää kuittauksen, saattaa myös kadota. Tätä varten lisätään vielä toinen 32-bittinen kenttä. Tämän kentän avulla voidaan välittää kuittaus viimeisimmästä 32 paketista. Tämä voidaan toteuttaa siten, että ensimmäinen bitti kertoo, onko kuittauskentän ilmoittaman paketin edellinen paketti vastaanotettu. Toinen bitti kertoo, onko sitä edellinen paketti vastaanotettu ja niin edelleen (ks. kuva 4).



Kuva 4. Ack ja Ack_bitfieldin toiminnan havainnollistaminen

Tämän kaltainen ratkaisu mahdollistaa 33 kuittausta jokaista vastaanotettua pakettia kohtaan. On hyvin epätodennäköistä, että toimivalla yhteydellä katoaisi näin suuri määrä peräkkäisiä paketteja.

Kuittausten lähettäminen muistuttaa hyvin paljon TCP-protokollan tapaa varmistaa pakettien luotettavuus. TCP-protokolla ylläpitää sliding window kenttää (ks. kuva 2) ja jos yksi paketti puuttuu välistä, joutuu lähettäjä pysähtymään, luomaan puuttuvan paketin uudestaan ja lähettämään sen samalla sekvenssinumerolla. Tämän kaltaista viivettä halutaan tässä protokollassa välttää ja jokaisella paketilla, myös uudelleenlähetyksellä, on oma uniikki sekvenssinumero. Tällä tavoin voidaan jatkaa pakettien lähettämistä ja samalla lähettää myös kadonneet paketit hidastelematta.

3.2.2 Kadonneen paketin tunnistaminen ja uudelleenlähettäminen

Kuittaukset ja sekvenssinumerot eivät kuitenkaan kerro vastaanottajalle, kun paketti on kadonnut, eikä vastaanottaja voisi edes tietää, onko kyseessä ollut kriittistä dataa, joka täytyy lähettää uudestaan. Tästä syystä lähettäjän vastuulla on seurata saatuja kuittauksia. Kadonneen paketin tunnistaa siitä, kun ennalta määrättyyn aikaan ei ole saapunut kuittausta lähetetystä paketista. Jos kadonnut paketti on merkitty tärkeäksi dataksi, jonka lähetys halutaan varmistaa, tulee se lähettää uudelleen. Tätä varten protokollan tulee pitää muistissa lähetettyjä paketteja, jotka on merkattu kriittiseksi.

Huomattavan erona TCP-protokollaan verrattuna on se, että uudelleenlähetyksessä paketille annetaan uusi sekvenssinumero, eikä samaa sekvenssinumeroa kuten TCP-protokollalla on tapana. Uudelleenlähetetyssä paketissa tulee olla merkattuna myös mistä paketista uudelleenlähetys on tehty, jotta vastaanottoja pystyy tunnistamaan mahdolliset duplikaatit ja tätä varten paketin otsikossa on varattu 32-bittinen kenttä ilmoittamaan sen

paketin sekvenssinumero josta uudelleenlähetys on tehty. Jos kyseessä ei ole uudelleenlähetys, kentän arvo on 0.

3.2.3 Duplikaattipaketit

Koska oma protokolla toimii UDP-protokollan päällä, joka taas toimii IP-protokollan päällä, on mahdollista, että osapuolet saattavat vastaanottaa duplikaattipaketteja. Duplikaatit on helppo suodattaa säilyttämällä sekvenssinumero viimeisimmistä vastaanotetuista paketeista ja tarkistamalla, onko kyseisellä sekvenssinumerolla jo vastaanotettu paketti. Jos paketin antamalla sekvenssinumerolla on jo vastaanotettu toinen paketti, se voidaan turvallisesti poistaa.

3.2.4 Pakettien järjestys

Paketit voidaan halutessa pitää järjestyksessä tarkistamalla saapuneiden pakettien sekvenssinumero. Jos paketin ja edellisen paketin sekvenssinumeron ero on suurempi kuin yksi, tiedetään, että yksi tai useampi paketti puuttuu välistä. Nämä paketit voidaan laittaa väliaikaisesti jonoon odottamaan, että välistä puuttuvat paketit saapuvat, ja sen jälkeen siirtää käsittelyyn.

3.3 Virtuaaliyhteys

Koska UDP-protokolla on yhteydetön protokolla, täytyy oman protokollan toteuttaa yhteyden luominen ja sen ylläpitäminen. Paketti joka lähetetään IP-protokollalla, joita molemmat UDP- ja TCP-protokollat käyttävät, kulkee useiden koneiden kautta, eikä koneiden välillä ole kiinteää "putkea", joita pitkin paketit liikkuvat. Tätä voi havainnollistaa käyttämällä tracert <osoite> -komentoa (ks. kuva 5) komentorivillä (traceroute unix järjestelmissä). Reitti koneiden välillä saattaa muuttua useinkin.[4.]

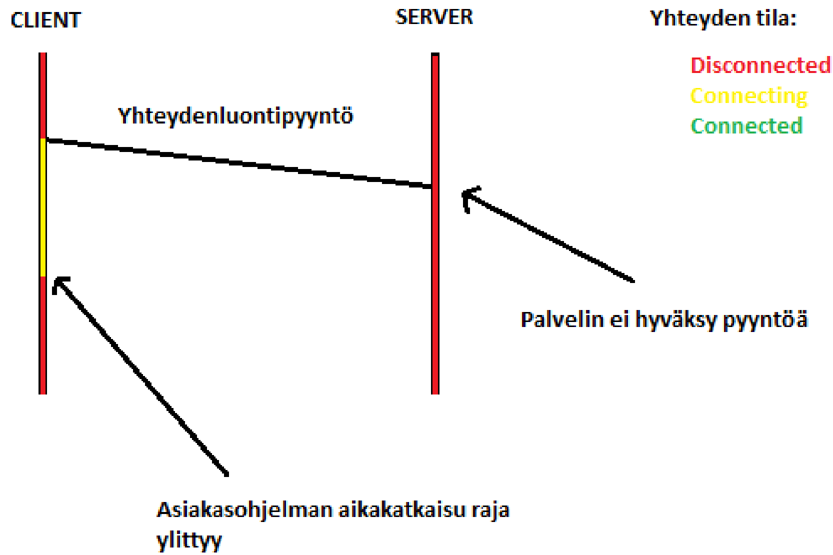
```
C:\Users\Antti>tracert -d metropolia.fi

Tracing route to metropolia.fi [195.148.144.10]
over a maximum of 30 hops:

  1    <1 ms    <1 ms    <1 ms  192.168.0.1
  2    12 ms    10 ms    10 ms  82.181.160.1
  3    11 ms     8 ms    11 ms  62.78.124.126
  4     9 ms    10 ms    10 ms  62.78.107.114
  5    13 ms    10 ms    12 ms  62.78.107.98
  6     9 ms     9 ms    10 ms  193.110.224.14
  7     9 ms    10 ms    10 ms  193.166.255.36
  8    13 ms    12 ms    10 ms  193.167.198.193
  9     9 ms    12 ms    12 ms  193.167.198.134
 10    11 ms     9 ms    10 ms  193.167.198.150
 11    11 ms    12 ms     9 ms  195.148.144.10

Trace complete.
```

Kuva 5. Paketin kulkema reitti lähetettäessä paketti metropolia.fi-osoitteeseen.

3.3.1 Virtuaaliyhteyden muodostaminen

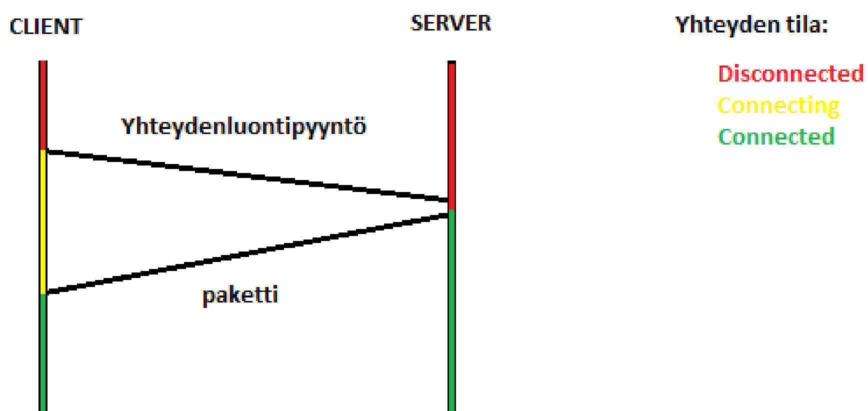
Virtuaaliyhteyden luomisessa harkittiin kahta eri tapaa. Yhteys voitaisiin luoda heti, kun vastaanotetaan paketti oikealla protocol_id:llä, jos lähettäjä ei ole aikaisemmin lähettänyt pakettia. Tässä protokollassa päätettiin kuitenkin tehdä 8 bitin kokoinen kenttä paketin otsikkotietoihin, joka ilmoittaa kyseisen paketin tyypin. Yhteyttä pyytävältä asiakasohjelmalta odotetaan CONNECTION_REQUEST (kentän arvo: 0) -tyyppistä pakettia, jotta yhteys voidaan luoda. Käyttäjä voi helposti lisätä paketin dataosioon tarpeellista tietoa liittyen yhteyden luomiseen kuten esimerkiksi nimimerkin tai muuta metatietoa. Paketin tyyppi kenttään mahtuu yhteensä 256 eri arvoa, joten eri pakettityyppejä voi olla suuri määrä, ja protokollan käyttäjä voikin määritellä omia tyyppejä ohjelmansa tarpeiden mukaan.

Kun asiakasohjelma lähettää palvelimelle yhteydenluontipyynnön, siirtyy asiakasohjelman yhteys ”disconnected”-tilasta ”connecting”-tilaan. Jos palvelin ei vastaa pyyntöön ennen aikakatkaisuaikaa, siirtyy asiakasohjelman yhteys takaisin ”disconnected”-tilaan (ks. kuva 6).



Kuva 6. Yhteydenluontipyyntöä ei hyväksytä

Mikäli palvelin vastaa asiakasohjelman pyyntöön, palvelin luo itselleen uuden yhteyden hashmappiin, jossa avaimena toimii asiakasohjelman päätepiste (Boost.Asio kirjaston udp endpoint), joka sisältää tiedon asiakasohjelman IP-osoitteesta ja portista ja arvona yhteys. Samalla asiakasohjelman yhteyden tila muutetaan connecting-tilasta connected-tilaan (ks. kuva 7).



Kuva 7. Yhteydenluontipyyntö hyväksytään

3.3.2 Yhteyden ylläpitäminen ja sulkeminen

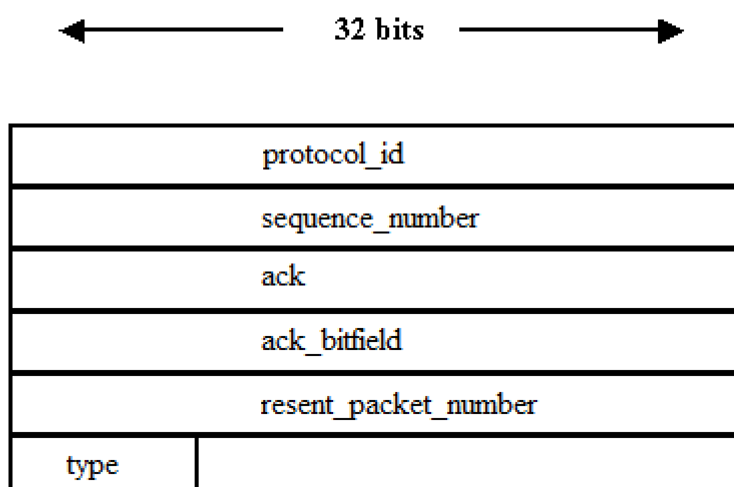
Yhteyden ylläpitämiseksi molempien osapuolien tulee lähettää paketteja, jotta yhteys pidetään päällä. Jos toinen osapuoli ei lähetä paketteja, toinen puoli päättelee, että toisella on internetyhteydessä ongelma, taikka on sulkenut ohjelman, joten se sulkee oman yhteytensä ja lopettaa pakettien lähettämisen. Protokollan käyttäjä voi itse määritellä yhteyden aikakatkaisuajan.

Yhteyden voi sulkea kumpi tahansa osapuoli lähettämällä DISCONNECT_NOTICE (kentän arvo: 3) -tyyppisen paketin. Tämä ilmoittaa toiselle osapuolelle, että aikoo lopettaa pakettien lähettämisen ja vastaanottamisen. Toinen tapa sulkea yhteys on vain lopettaa pakettien lähettäminen ja sulkea oma yhteys. Toisen osapuolen yhteys sulkeutuu, kun paketteja ei ole vastaanotettu aikakatkaisuajan määrittelemänä aikana.

3.3.3 Protokollan otsikkotiedot

Kokonaisuudessaan oman protokollan ylätunniste vie 168 bittiä. Tätä voitaisiin mahdollisesti pienentää lyhentämällä joidenkin kenttien bittimäärää kolmestakymmenestäkahdesta kuuteentoista kuten esimerkiksi protocol_id, sequence_number, ack ja resent_packet_id kenttiä. Tällä tavoin ylätunnisteen koko olisi 104 bittiä.

Protokollaan päätettiin kuitenkin käyttää 32-bittisiä kenttiä. Tätä voidaan harkita uudelleen, jos protokollan suorituskyky alkaa kärsimään. Kokonaisuudessaan paketin ylätunnisteen näkee kuvassa 8.



Kuva 8. Oman protokollan otsikkotiedot

3.3.4 Ruuhkanhallinta

Toisin kuin TCP-protokolla, UDP-protokolla ei tarjoa minkäänlaista ruuhkanhallintaa. Ruuhka tarkoittaa sitä, kun reitittimeen tulee enemmän paketteja kuin se pystyy lähettämään eteenpäin. Reitittimet pyrkivät kuitenkin parhaansa mukaan välittämään kaikki paketit eteenpäin ja tästä syystä ottavat paketit puskuriin. Jos puskuri alkaa täyttymään liikaa, voi se aiheuttaa suurta latenssia, ne saattavat tehdä esimerkiksi pelistä täysin pelaamattoman. Jos puskuri täyttyy, joudutaan ylimääräiset paketit hylkäämään. Tästä seuraa paketin katoamista, ja tämä onkin yleisin syy paketin katoamiselle. [5.]

Protokollan tulisi pystyä tunnistamaan mahdolliset ruuhkatilanteet, jotta voitaisiin välttyä kohtuuttomasta ruuhkasta ja sen aiheuttamista ongelmista. Ehkäpä helpoin tapa tunnistaa ruuhkautuminen on seurata pakettien RTT:tä (round trip time). Koska ruuhkatilanteissa latenssi nousee huomattavasti, voidaan korkeasta RTT-ajasta päätellä, että jokin reititin puskuroi suuren määrän paketteja.

Kun ruuhka on tunnistettu, tulisi tehdä jotain sen korjaamisiseksi. Helpoin tapa on vähentää lähetettävien pakettien määrää. Tällöin mahdollisesti ruuhkautunut reititin pystyy ehkä palautumaan. Kun RTT laskee takaisin alle annetun rajan, voidaan jatkaa normaalia lähettämistä. Tässä tulisi kuitenkin olla jokin minimi aikaraja, että ei jatkuvasti hypitä ruuhkanestotilasta normaaliin tilaan ja toisin päin. Ehtona normaaliin tilaan palaamiselle

voisi olla, että RTT pitää olla alle annetun rajan *t* sekuntia ennen kuin palataan takaisin normaaliin tilaan.

Tätä algoritmia voitaisiin parannella seuraamalla RTT:n lisäksi myös kadonneiden pakettien määrää, sekä vaihtelua RTT-arvossa. Tässä protokollassa päätettiin kuitenkin toteuttaa yksinkertaisin mahdollinen ruuhkantunnistus, sillä se toimii varsin hyvin.

4 Protokollan toteutus

Tässä luvussa esitellään, kuinka oma protokolla toteutettiin. Lisäksi käydään läpi hyödynnettävät kolmannen osapuolen kirjastot, ja esitellään omat ratkaisut koodi esimerkkeineen. Kyseessä ei ole kuitenkaan täysin kattava ohje oman UDP-pohjaisen protokollan luomiseen vaan yleiskuva toteutuksesta, joten toteutuksen kuvausta voidaan käyttää tukena oman UDP-pohjaisen protokollan luomiseen.

4.1 Pakettien lähettäminen ja vastaanottaminen

Tässä aliluvussa Boost.Asio-kirjaston käyttöön perehdytään hieman tarkemmin ja käydään läpi, kuinka Boost.Asion avulla voidaan avata UDP-soketti sekä kuinka UDP-sokettia hyödyntäen voidaan asynkronisesti lähettää ja vastaanottaa UDP-paketteja verkon yli.

4.1.1 Ohjelmointikieli ja Boost.Asio-kirjasto

Protokolla toteutettiin C++-ohjelmointikielellä, ja se hyödyntää Boost.Asio-kirjastoa. Boost.Asio on järjestelmäriippumaton C++-kirjasto matalan tason verkko- ja I/O-ohjelmointiin. Asio tarjoaa kehittäjille johdonmukaisen asynkronisen mallin käyttäen modernia C++-lähestymistapaa.

Syy, miksi projektissa päätettiin käyttää Boost.Asio-kirjastoa on, koska se on järjestelmäriippumaton ja suorituskyvyltään hyvä. Asio on myös varsin helppokäyttöinen ja säästää reilusti aikaa, kun ei tarvitse itse toteuttaa aivan matalimman tason asioita.

4.1.2 UDP-soketin avaaminen

Boost.Asio-kirjastolla UDP-pakettien lähettäminen onnistuu soketin avaamisen jälkeen. Soketin avaamiseksi täytyy luoda io_service-olio, joka toimii protokollan yhteytenä käyttöjärjestelmän I/O-palveluihin. Tämän lisäksi soketti vaatii tiedon, käyttääkö se IPv4- vai IPv6-protokollaa sekä mitä porttia tullaan käyttämään.

```
using boost::asio::ip::udp;

boost::asio::io_service io_service;
udp::socket socket(io_service, udp::v4(), 1234);
```

Kuva 9. UDP-soketin luominen ja avaaminen

4.1.3 UDP-pakettien lähettäminen ja vastaanottaminen

Boost.Asio tarjoaa synkronisen ja asynkronisen tavan lähettää ja vastaanottaa paketteja. Asynkronisten funktioiden nimissä on ”async_”-etuliite. Protokollan toteutuksessa käytettiin asynkronisia funktioita.

Asynkroniset funktiot vaativat callback-funktion, jota aikaisemmin luotu io_service-olio kutsuu operaation ollessa valmis. Asynkroniset lähetys- ja vastaanottofunktiot lisäksi vaativat, että lähetyksessä käytetyn datan muisti alue on validi, kunnes lähetys on suoritettu. Tämän lisäksi asynkroninen vastaanottofunktio vaatii, että sender_endpoint (eli lähettäjän osoite) -muuttujan muistialue on validi, kunnes callback-funktiota on kutsuttu. Lisäksi io_servicen ”run” -funktiota täytyy olla kutsuttu, jotta asynkroniset operaatiot toimivat.

4.1.4 UDP-paketin lähettäminen

Asynkroninen paketin lähetys tapahtuu Boost.Asio-kirjaston socket-olion avulla. Funktio, joka suorittaa lähetyksen, on nimeltään ”async_send_to” (ks. kuva 10).

Asynkroninen paketin lähetys vaatii parametreikseen Boost.Asio buffer -olion, joka sisältää lähetettävän datan, päätepisteen lähetykselle sekä callback-funktion, jolle annetaan parametreina lähetettävä data (jotta lähetettävän datan muisti alue säilyy eheänä), boost::system::error_code-muuttuja sekä lähetettyjen tavujen määrä.

```
socket_.async_send_to(boost::asio::buffer(data), remote_endpoint,
  boost::bind(&UdpSender::handleSend, this, tempData,
    boost::asio::placeholders::error,
    boost::asio::placeholders::bytes_transferred));
```

Kuva 10. Asynkroninen lähetys

Callback-funktiolle annettu shared_ptr pitää huolen, että lähetetyn datan muistialue on validi kunnes data on lähetetty. Lisäksi se myös automaattisesti poistaa datan, kun callback-funktioista poistutaan (ks. kuva 11).

```
void UdpSender::handleSend(std::shared_ptr<std::vector<unsigned char>> message,
  const boost::system::error_code& error,
  std::size_t bytes_transferred)
{
  if (error)
    std::cout << "Error sending message" << std::endl;
  else
    std::cout << "Successfully sent message" << std::endl;
}
```

Kuva 11. Asynkronisen lähetyksen callback-funktio

4.1.5 UDP-paketin vastaanottaminen

Asynkroninen paketin vastaanottaminen vaatii myös parametrikseen Boost.Asio buffer-olion, jonka sisällä on säiliö, mihin vastaanotetun datan voi tallettaa. Tämän lisäksi vaatii funktio myös udp endpoint-olion, johon tallennetaan lähettäjän päätepiste (IP-osoite ja portti). Funktio vaatii, että funktion kutsuja säilyttää muuttujien omistajuuden ja että olioiden muisti alue pysyy validina siihen asti, kunnes callback-funktioita kutsutaan.

UDP-paketin vastaanotossa päätettiin toteuttaa vaadittujen olioiden eheyden varmistaminen eri tavalla. Koska paketteja vastaanotetaan vain yksi kerrallaan, voi nämä oliot olla sen luokan muuttujia, jossa kutsutaan paketin vastaanottofunktiota (async_receive_from).

```
void UdpReceiver::startReceive()
{
  socket_.async_receive_from(
    boost::asio::buffer(recv_buffer_), remote_endpoint_,
    boost::bind(&UdpReceiver::handleReceive, this,
      boost::asio::placeholders::error,
      boost::asio::placeholders::bytes_transferred));
}
```

Kuva 12. UDP-paketin vastaanottofunktio

Callback-funktio saa parametrikseen error-koodin ja vastaanotettujen tavujen määrän. Data ja lähettäjän päätepiste saadaan luokan muuttujista recv_buffer_ ja remote_endpoint_. Callback-funktiossa myös käsitellään vastaanotettu data ja kutsutaan uudelleen paketin vastaanottamista. Tällä tavoin paketteja vastaanotetaan niin kauan kunnes, io_service-olio pysäytetään.

```
void UdpReceiver::handleReceive(const boost::system::error_code& error,
  std::size_t bytes_transferred)
{
  if (!error)
  {
    //handle data contained in recv_buffer_ and remote_endpoint_ objects
  }
  else
  {
    std::cout << "ERROR! ; UdpHandler handleReceive  " << error.message() << std::endl;
  }
  startReceive();
}
```

Kuva 13. UDP-paketin vastaanottofunktion callback funktio

4.2 Yhteyden tila

Protokollan ominaisuudet löytyvät connection-luokasta. Connection-luokan tehtävänä on seurata ja päivittää connection-luokan tilaa. Connection-luokalla on kolme tilaa: disconnected, connecting, connected. Connection-luokan on aloitettaessa disconnected-tilassa. Disconnected-tilasta voidaan siirtyä joko suoraan connected-tilaan, jos hyväksytään verkon yli tullut yhteydenluontipyyntö, tai jos itse pyydetään yhteyttä, siirrytään connecting-tilaan. Connecting-tilasta siirrytään connected-tilaan, jos vastaanotetaan hyväksyntä kohteelta. Jos kohde ei vastaa, siirrytään takaisin disconnected-tilaan (ks. kuvat 11 ja 12).

4.3 Sekvenssinumerot

Connection-luokka seuraa omaa ja yhteyden toisen osapuolen sekvenssinumeroa. Tätä tietoa pidetään muuttujissa local_sequence_number_ ja remote_sequence_number_. Jokaisen paketin lähetyksen jälkeen local_sequence_number-muuttujan arvoa nostetaan yhdellä ja remote_sequence_number-muuttujan arvoksi asetetaan aina viimeisimmän vastaanotetun paketin sekvenssinumero.

Koska sekvenssinumeron maksimiarvo on 2^32 - 1 (4 294 967 295), saattaa se jossain vaiheessa ylivuotaa, kun yhteys on ollut todella kauan päällä. Tämä on useimmissa tapauksissa äärimmäisen epätodennäköistä, mutta siihen on syytä varautua. Ylivuodossa sekvenssinnumero muuttuu maksimiarvosta nollaksi, kun sen arvoa nostetaan yhdellä. Protokollassa sekvennsinumeroa 0 ei käytetä, sillä 0 tarkoittaa resent_packet_id-kentässä, että kyseessä ei ole uudelleenlähetys. Tästä syystä sekvenssinumeroa korotettaessa täytyy tarkistaa, jos arvo on 0 ja lisätä siihen yksi jos on.

Suurempi ongelma on protokollan luotettavuussysteemin toiminta tällaisessa tilanteessa. Protokolla voi luulla pakettia sekvenssinumerolla 1 vanhemmaksi kuin paketti, jossa sekvenssinumero on 2^32 - 1. Lisäksi ylivuodon jälkeen ack_bitfieldin täyttö ei samasta syystä toimi normaalisti. Tätä varten on luotu funktio, joka kertoo, kumpi paketeista on uudempi (ks. kuva 14).

```
bool moreRecentPacket(uint32_t p1, uint32_t p2)
{
  return (p1 > p2) && (p1 - p2 <= max_sequence / 2) ||
         (p2 > p1) && (p2 - p1 > max_sequence / 2);
}
```

Kuva 14. Uudemman paketin tunnistaminen.

Funktiossa verrataan kahden paketin sekvenssinumeroa. Jos sekvenssinumeroiden välinen matka on vähemmän kuin puolet maksimisekvenssinumerosta (2^32 - 1), ovat sekvenssinumerot lähekkäin ja voidaan suoraan katsoa, kumpi sekvenssinumeroista on suurempi. Jos sekvenssinumeroiden välimatka on suurempi kuin puolet maksimisekvenssinumerosta, on välissä tapahtunut sekvenssinumeron ylivuoto. Tällöin voidaan olettaa, että pienempi sekvenssinumero on uudempi. Tätä funktiota tulee käyttää aina, kun verrataan sekvenssinumeroita.

4.4 Pakettien kuittaus

Osapuolet pitävät myös kirjaa viimeisimmistä vastaanotetuista paketeista. Tämän avulla voidaan luoda lähetettävään pakettiin ack, ja ack_bitfield-kenttien arvot. Ack-kenttään asetetaan remote_sequence_number_-muuttujan arvo. Ack_bitfield-kentän arvo riippuu siitä, kuinka moni paketti ennen remote_sequence_number_-arvoa on vastaanotettu (ks. kuva 9). Bitfield voidaan koota 32-bittiseen kokonaislukuun helposti käyttämällä bitshiftiä ja bitwise OR -operaattoreita ja iteroimalla läpi viimeisimmät saapuneet paketit (ks. kuva 15).

```
uint32_t createAckBitfield()
{
  uint32_t ack_bitfield = 0;
  std::lock_guard<std::mutex> lock(received_packets_lock);
  auto it = received_packets_.begin();
  while (it != received_packets_.end())
  {
    if (!moreRecentPacket(*it, remote_sequence_number_))
    {
      uint32_t distance;
      if (*it < remote_sequence_number_)
        distance = remote_sequence_number_ - *it;
      else
        distance = remote_sequence_number_ - *it + max_sequence;
      if (distance > 0 && distance < 33)
        ack_bitfield |= 1 << (distance - 1);
    }
    *it++;
  }
  return ack_bitfield;
}
```

Kuva 15. ack_bitfield-kentän luominen

Yllä näkyvässä funktiossa iteroidaan läpi säiliö, joka sisältää vastaanotetut paketit. Jokaisen paketin kohdalla varmistetaan, että kyseessä ei ole uudempi paketti kuin juuri vastaanotettu (paketit voivat tulla väärässä järjestyksessä) ja lasketaan oikea paikka ack_bitfieldissä. Seuraavaksi tähän kohtaan asetetaan bitti 1, jos kyseinen paketti on vastaanotettu. Tämä onnistuu siten, että siirretään 32-bittistä arvoa 1 vasemmalle etäisyys - 1 verran ja liitetään se yhteen ack_bitfieldiin OR-operaattorin avulla.

4.5 Paketin kuittauksen käsittely

Jokainen lähetetty paketti, joka on merkitty tärkeäksi, eli jonka toimitus pitää taata, tallennetaan muistiin, kunnes sitä vastaavan sekvenssinumeron kuittaus on vastaanotettu tai kun tietty aikaraja on ylittynyt ja paketti lähetetään uudestaan. Tätä varten jokaisen vastaanotetun paketin kohdalla käydään ack- ja ack_bitfield-kentät läpi ja poistetaan kuittausta vastaavat paketit väliaikaismuistista. Ack-kentän lukeminen on yksinkertaista, mutta ack_bitfield-kentän lukeminen vaatii jälleen bit shift-operaattoria sekä bitwise AND-operaattoria (ks. kuva 16).

```
void checkReceivedAcks(MessageHeader& header)
{
  std::lock_guard<std::mutex> lock(sent_messages_lock);
  sent_messages.erase(header.ack);
  uint32_t bitfield = header.ack_bitfield;
  for (int i = 1; i < 33; ++i)
  {
    if (bitfield & 0x01)
      sent_messages.erase(header.ack - i);
    bitfield = bitfield >> 1;
  }
}
```

Kuva 16. Pakettien poisto väliaikaissäiliöstä

Ensin poistetaan ack-kenttää vastaava paketti säiliöstä ja sen jälkeen iteroidaan ack_bitfield siten, että luetaan ensimmäinen bitti, ja jos se on 1, poistetaan ack - i vastaava paketti säiliöstä. Tämän jälkeen bittejä liikutetaan yksi oikealle ja toistetaan tätä, kunnes kaikki 32 bittiä on tarkastettu.

4.6 Paketin uudelleenlähettäminen

Mikäli tärkeäksi merkitystä paketista ei saada kuittausta ennalta määriteltyyn aikaan (oletuksena 1 sekunti), tulee se lähettää uudelleen. Tällöin lähetettyjen pakettien väliaikaissäiliöstä haetaan uudelleenlähetystä vaativa paketti, poistaen se säiliöstä, lähettämällä se ja tallentamalla se takaisin uudella sekvenssinumerolla.

4.7 Ruuhkanhallinta

Ruuhkan tunnistaminen toteutettiin seuraamalla RTT-arvoa ja mikäli RTT nousee yli annetun arvon, voidaan päätellä, että jokin reititin matkalla on ruuhkautunut. RTT on aika, joka kuluu paketin lähettämisestä sen kuittaamiseen. RTT voidaan siis laskea aina kuittauksia käsitellessä (ks. kuva 17).

```
void checkAck(uint32_t ack)
{
  auto packet = sent_packets_.find(ack);
  if (packet != sent_packets_.end())
  {
    long long packet_rtt = Timer::getElapsedTime(packet->second);
    rtt_ += (packet_rtt - rtt_) * 0.1f;
    sent_packets_.erase(ack);
  }
  updateCongestionAvoidance();
}
```

Kuva 17. RTT-arvon laskeminen

Funktiolle annetaan käsiteltäväksi kuittausnumero. Se etsitään sent_packets-säiliöstä, ja jos se löytyy, lasketaan kulunut aika lähetyksestä kuittaukseen. RTT-arvoa ei kuitenkaan korvata suoraan nykyisen paketin RTT-arvolla vaan RTT-arvoa muutetaan vain 10 % tämän paketin RTT-arvon ja aikaisemman RTT-arvon erotuksen verran. Kun RTT-arvo on laskettu, voidaan sent_packets-säiliöstä poistaa kyseinen paketti. Lopuksi kutsutaan funktiota, joka muuttaa yhteyden ruuhkatilaa tarvittaessa.

Yhteydellä on kaksi tilaa ruuhkan kannalta: joko ruuhkanvälttäminen on päällä tai ei ole. Käyttäjä pystyy kysymään Connection-luokalta sen nykyistä tilaa ja se jääkin käyttäjän vastuulle tehdä jotain tilanteelle, jos ruuhka on havaittu.

Algoritmi, jota käytetään ruuhkatilan vaihtamiseen, toimii seuraavalla tavalla: jos ruuhkatila ei ole päällä ja RTT nousee yli annetun rajan, siirrytään välittömästi ruuhkatilaan. Mikäli aikaisemmasta tilan vaihdosta on alle kymmenen sekuntia, nostetaan aikaa *t* (koodissa penalty_time) 15 sekuntia. Aika *t* kertoo, kuinka pitkä aika vaaditaan, että voidaan palata takaisin normaaliin tilaan. Vastaavasti, mitä kauemmin ruuhkatila on pois päältä, sen lyhempi aika *t* tulee olemaan. Tässä tapauksessa aikaa puolitetaan kymmenen sekunnin välein, jos RTT:n arvo on alle rajan. Minimiarvo ajalle *t* on yksi sekunti.(ks. kuva 18) Tällä tavoin vältytään tilanteesta, jossa jatkuvasti vaihdellaan tilojen välillä. Lisäksi jos yhteys on hyvä, yritetään nopeammin palauttaa ruuhkatila takaisin normaaliin tilaan.

```
long long now = Timer::getTimeMilliseconds();
if (!congestion_avoidance_)
{
  if (rtt_ > rtt_threshold_)
  {
    if (now - congestion_avoidance_toggle_time < 10000)
      penalty_time += 15000;
    congestion_avoidance_ = true;
    congestion_avoidance_toggle_time = now;
  }
  else
  {
    if (now - congestion_avoidance_toggle_time > 10000)
    {
      penalty_time /= 2;
      if (penalty_time < 1000)
        penalty_time = 1000;
      congestion_avoidance_toggle_time = now;
    }
  }
}
```

Kuva 18. Ruuhkatilan päivitys, kun ruuhkatila ei ole päällä.

Jos ruuhkatila on päällä, niin normaaliin tilaan voidaan palautua ainoastaan, jos RTT-arvo on alle annetun rajan yli *t* millisekuntia. Kun tila vaihdetaan takaisin normaalitilaan, asetetaan aika *t* kymmeneen sekuntiin (ks. kuva 19).

```
else if (congestion_avoidance_)
{
  if (rtt_ < rtt_threshold_)
  {
    if (now - congestion_avoidance_toggle_time > penalty_time)
    {
      congestion_avoidance_ = false;
      penalty_time = 10000;
      congestion_avoidance_toggle_time = now;
    }
  }
  else
  {
    congestion_avoidance_toggle_time = now;
  }
}
}
```

Kuva 19. Ruuhkatilan päivitys kun ruuhkatila on päällä.

4.8 Datan lähetys Connection-luokan avulla

Connection-luokkaa voidaan käyttää datan lähetykseen yhteyden läpi. Tämä tapahtuu sendMessage-funktion avulla. Funktio ottaa parametrikseen vektorin tavuja (uint8_t), MessageType enumin, millä voidaan kertoa, minkälaisesta paketista on kyse (esim. CONNECTION_REQUEST) sekä Reliability enumin, millä voidaan määritellä, pitääkö paketin toimitus varmistaa tai varmistaa ja pitää järjestyksessä. Connection-luokka huolehtii paketin otsikkotietojen luomisesta.

4.9 Datan vastaanotto Connection-luokan avulla

Connection-luokaan kuuluvat, vastaanotetut paketit, annetaan receiveMessage funktion avulla. Funktio ottaa vastaan UDPReceiver-luokan luoman ReceivedMessage structin, joka pitää sisällään lähettäjän päätepisteen, viestin koon sekä itse datan. receiveMessage-funktio kokoaa datasta myös otsikkotiedot. Tätä varten on luotu apuluokka MessageExtractor, joka luo ReceivedMessage tyyppisestä structista Message-tyyppisen structin, joka sisältää Header structin sekä itse datan. ReceiveMessage-Funktio tarkistaa otsikkotiedoissa olevan sekvenssinumeron ja poistaa paketin, mikäli kyseessä on duplikaattipaketti. Muussa tapauksessa paketti tallennetaan säiliöön, sekä päivitetään remote_sequence_number_ muuttujan arvo vastaamaan paketin sekvenssinumeroa.

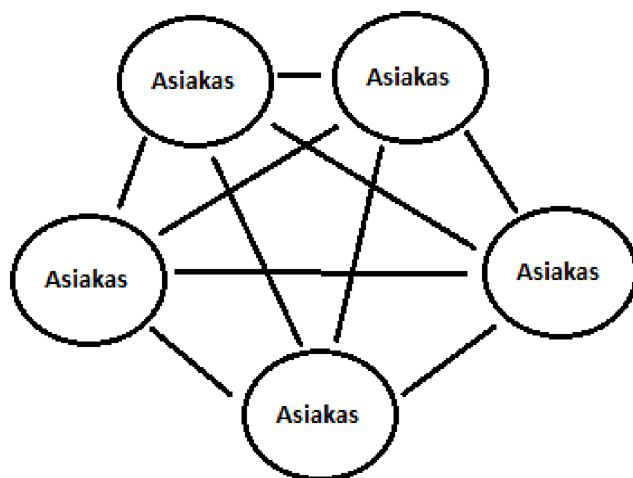
On käyttäjän vastuulla tarkistaa ja käsitellä Connection-luokan vastaanottamat paketit. Vastaanotetut viestit voidaan noutaa popReceivedMessages-funktion avulla, joka palauttaa kaikki vastaanotetut viestit ja tyhjentää received_messages-säiliön.

5 Asiakas-palvelin-malli

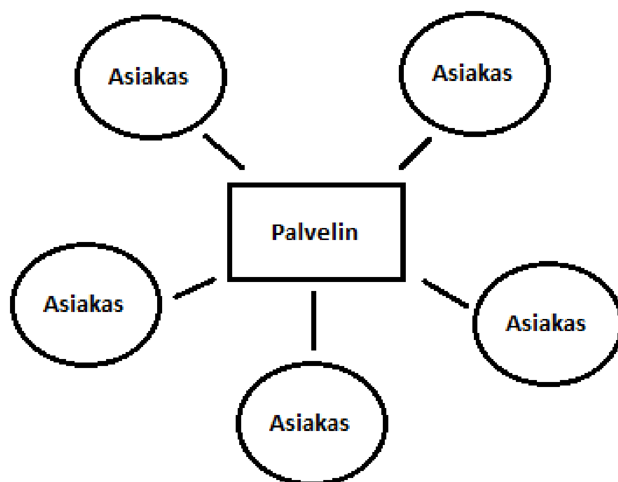
Tässä luvussa käydään läpi, kuinka protokollan avulla voidaan luoda asiakas-palvelin-mallin (Client-server model) mukainen rakenne. Ensimmäisenä selitetään lyhyesti, mikä asiakas-palvelin -malli on ja mihin sitä voidaan käyttää. Lopuksi näytetään, kuinka Client- ja Server-luokkien toiminnallisuus on toteutettu hyödyntäen aikaisemmin luotua protokollaa.

5.1 Asiakas-palvelin-mallin rakenne

Asiakas-palvelin-mallissa asiakaskoneet eivät keskustele suoraan keskenään, vaan kaikki kommunikointi tapahtuu palvelimen välityksellä. Asiakas-palvelin-mallissa ainoastaan asiakas voi pyytää yhteyttä. Toisin kun esimerkiksi vertaisverkoissa (peer-to-peer, ks. kuva 20) asiakaskoneet eivät myöskään ole suorassa yhteydessä toisiinsa, vaan kaikki kommunikaatio tapahtuu palvelinkoneen kautta (ks. kuva 21). [6; 12.]



Kuva 20. Vertaisverkko



Kuva 21. Asiakas-palvelin-malli

Esimerkkejä sovelluksista jotka käyttävät asiakas-palvelin-mallia on Sähköposti, HTTP, DNS-palvelin ja suurin osa moninpeleistä.

Asiakas-palvelin-mallin mukainen toiminnallisuus on jaettu kahteen eri luokkaan: Client luokkaan, joka vastaa asiakaskoneen toiminnallisuudesta, ja Server-luokkaan, joka vastaa palvelinkoneen toiminnallisuudesta. Molemmat luokat käyttävät kommunikoimiseen aikaisemmin luotua protokollaa ja sen funktioita.

5.2 Client (asiakas)

Client-luokka omistaa yhden Connection-olion. Luodessa Client-olion, saa se tiedon porttinumerosta, jota tullaan käyttämään datan vastaanottamiseen. Konstruktorissa avataan myös UDP-soketti annettuun porttiin, mutta vastaanotettua dataa ei käsitellä.

5.2.1 Yhteyden luonti

Yhteyden client luo connectTo funktiossa (ks. kuva 22). Funktio vaatii tiedon palvelimen IP-osoitteesta sekä portista, johon pyritään ottamaan yhteys sekä mahdollinen viesti, joka halutaan lähettää samalla. Funktiossa luodaan uusi Connection-olio, jolle annetaan tieto soketista, jota käytetään datan lähettämiseen sekä vastaanottamiseen. Samalla luodaan UdpReceiver-olio, joka vaatii tiedon soketista, jota kuunnella, sekä osoittimen olioon, jolle saadut viestit välitetään eli tässä tapauksessa osoitin äsken luotoon Connection-olioon. Samalla pyydetään UdpReceiver-oliota aloittamaan pakettien kuuntelu. Seuraaviksi Connection-oliota pyydetään lähettämään liittymispyyntö aikaisemmin annetun viestin kera. Lopuksi vielä käynnistetään Boost.Asio-kirjaston io_service-olio.

```
void connectTo(boost::asio::ip::udp::endpoint endpoint,
  std::vector<unsigned char> data)
{
  connection_.reset(new Connection(socket_));
  udp_receiver_.reset(new UdpReceiver(socket_, connection_.get()));
  udp_receiver_->startReceive();
  connection_->connect(endpoint, data);
  service = std::thread(&Client::run, this);
}
```

Kuva 22. Client-luokan connectTo-funktio.

Lisäksi Client-luokkaan luotiin vastaavat isConnected-, isConnecting-, isDisconnected- sekä congestionAvoidance- funktiot, jotka palauttavat tietoa client-olion omistamasta connection-oliosta.

5.2.2 Pakettien lähettäminen ja vastaanottaminen

Pakettien lähettäminen onnistuu Client-olion sendMessage-funktiolla. Sen parametrit ovat täysin identtinen connection-luokan sendMessage-funktion kanssa, ja se ainoastaan kutsuu connection-olion vastaavaa funktiota.

Paketit vastaanotetaan UdpReceiver-olion avulla, joka käynnistetään samalla, kun yhteyttä pyritään luomaan. Client-oliolta voi pyytää kaikkia vastaanotettuja paketteja popReceivedMessages-funktiolla. Se kutsuu connection-luokan popReceivedPackets-funktiota, joka palauttaa kaikki viime kutsun jälkeen saadut paketit

5.2.3 Yhteyden sulkeminen.

Yhteyden sulkeminen client-oliolla onnistuu samalla tavalla kuin Connection-luokallakin. Client-oliolla pystyy lisäksi kutsumaan funktiota disconnectWithMessage-funktiota, joka lähettää DISCONNECT_NOTICE -tyyppisen paketin yhteyden toiselle osapuolelle ennen disconnect-funktion kutsua. Disconnect-funktio kutsuu connection-luokan disconnect-funktiota sekä pysäyttää io_service-olion.

5.3 Server (palvelin)

Server-olio poikkeaa client-oliosta siten, että yhden connection-olion sijaan server-olio voi ylläpitää useampaa connection-oliota. Server ei pysty kuitenkaan aloittamaan yhteyttä vaan odottaa asiakasohjelmilta yhteydenottopyyntöjä. Server käyttää myös UdpReceiver-oliota samalla tavalla kuin client-oliokin. Erona on kuitenkin se, että Server asettaa UdpReceiver-olion välittämien pakettien vastaanottajaksi itsensä.

5.3.1 Virtuaaliyhteydet

Virtuaaliyhteyksiä varten server ylläpitää hashmappia, jossa avaimena on endpoint ja arvona connection-olio. Käyttäjä voi myös antaa server-oliota luodessa oman ConnectionAssigner-rajapinnan toteuttavan olion. ConnectionAssigneria käytetään, kun halutaan luoda uusi virtuaaliyhteys. Ennen kun virtuaaliyhteys luodaan, kysytään Connec-

tionAssigner-rajapinnan toteuttavalta oliolta lupaa virtuaaliyhteyden lisäämiseksi. ConnectionAssigner saa myös tiedon connection-olion osoittimesta ja tätä kautta käyttäjä pääseekin halutessaan suoraan kiinni connection-olioihin.

5.3.2 Pakettien vastaanottaminen

Server vastaanottaa receiveMessage-funktioon UdpReceiver-olion välittämät paketit. Ensimmäisenä server luo paketista Message-tyyppisen structin helpompaa käsittelyä varten. Seuraavaksi tarkistetaan, onko lähettäjän osoitteeseen jo luotu yhteys. Jos yhteys on jo luotu ja paketin tyyppinä ei ole DISCONNECT_NOTICE, viesti välitetään sitä vastaavalle connection-oliolle. Mikäli tyyppinä on DISCONNECT_NOTICE, tiedetään, että asiakas on sulkemassa yhteyden, voi server myös sulkea yhteyden, ja poistaa sitä vastaavan connection-olion.

Jos lähettäjän osoitetta vastaavaa connection-oliota ei ole vielä luotu, luodaan sellainen, mikäli kyseessä on CONNECTION_REQUEST-tyypin paketti. Ehtona on kuitenkin, että server-luokassa on tilaa uudelle yhteydelle. Tämän lisäksi tarkistetaan vielä käyttäjän aikaisemmin antaman ConnectionAssigner-rajapinnan toteuttaman olion ehdot liittymiselle. Mikäli palvelimella on tilaa ja ConnectionAssigner hyväksyy yhteydenottopyynnön, lisätään hashmappiin uusi yhteys.

5.3.3 Pakettien lähettäminen

Server-oliolla pakettien lähettäminen toimii server-luokan kautta. Saatavilla on kaksi eri funktiota, joita voi käyttää pakettien lähettämiseen: sendTo tai sendToAllConnections. Ensimmäinen funktio lähettää paketin vain yhdelle asiakasohjelmalle ja toinen kaikille. Molemmat funktiot käyttävät connection-luokkaa datan lähetykseen. Molemmille annetaan samat parametrit kuin connection-luokan sendMessage-funktiolle. Lisäksi sendTo-funktio tarvitsee tiedon, kenelle paketti lähetetään. sendTo-funktiossa haetaan hashmapista endpointtia vastaava connection-olio ja lähetetään paketti sille (ks. kuva 23), kun taas sendToAllConnections-funktio iteroi koko hashmapin lähettäen paketin jokaisella connection-oliolla (ks. kuva 24).

```
bool sendTo(udp::endpoint endpoint, std::vector<uint8_t> data, MessageType type,
            Reliability reliability)
{
  auto entry = connections_.find(endpoint);
  if (entry != connections_.end())
  {
    entry->second->sendMessage(data, type, reliability);
    return true;
  }
  return false;
}
```

Kuva 23. sendTo-funktio.

```
void sendToAllConnections(std::vector<unsigned char> message, MessageType type,
                          Reliability reliability)
{
  for (auto entry : connections_)
  {
    entry.second->sendMessage(message, type, reliability);
  }
}
```

Kuva 24. sendToAllConnections-funktio

6 Yhteenveto

Tämän insinöörityön tarkoituksena oli luoda luotettava UDP-pohjainen protokolla, jonka avulla käyttäjä voi toteuttaa luotettavan UDP-yhteyden luomisen ja ylläpidon ohjelmassaan. Lisäksi protokollan avulla luotiin valmis rajapinta yksinkertaisen asiakas-palvelinmallin mukaisen rakenteen toteuttamiseksi. Tavoitteena oli luoda tehokas, pienilatenssinen, mutta silti luotettava UDP-yhteys.

Projektia varten tutkittiin IP-, TCP- ja UDP-protokollien heikkouksia ja vahvuuksia. Heikkouksien sekä vahvuuksien pohjalta valittiin oman protokollan pohjaksi UDP-protokolla. Lisäksi avuksi otettiin Boost.Asio-kirjasto, jonka avulla pystyttiin helposti toteuttamaan UDP-pakettien lähetys sekä vastaanotto. Protokollan ominaisuuksien suunnittelu ja toteutus olivat insinöörityön keskeisin asia ja tähän paneuduttiin eniten projektin aikana. Rajapinta asiakas-palvelin-mallin mukaiseen verkkokommunikointiin syntyi testauksen sivutuotteena. Ratkaisu oli kuitenkin toimiva, ja sitä voidaan varmasti hyödyntää tulevissa projekteissa.

Kokonaisuudessaan insinöörityön aikana suunniteltu ja toteutettu protokolla onnistui hyvin. Protokollan avulla voidaan luotettavasti lähettää UDP-paketteja sekä pakettien kadotessa luotu protokolla automaattisesti lähettää paketin uudestaan. Yhteyden luominen, ylläpito sekä sammuttaminen toimivat hyvin. Protokolla tunnistaa toisen osapuolen yhteyden katkaisemisen vaikka erillistä ilmoitusta siitä ei tulisikaan. Protokolla tunnistaa myös ruuhkatilanteet hyvin RTT-arvoa tarkkailemalla.

Parannettavaa olisi erityisesti protokollan suorituskyvyssä. Monissa tilanteissa aikaa ei jäänyt optimoinnille ja suurilta osilta protokolla onkin enemmän prototyyppi. Protokollan suorituskykyyn tulisi perehtyä tarkemmin, jotta se ei aiheuta ylimääräistä latenssia yhteyteen. Tällä hetkellä paketin RTT-arvo paikallisesti yhtä yhteyttä testatessa on noin viisi millisekuntia. Lisäksi protokollaa tulisi testata useammalla yhteydellä samanaikaisesti.

Protokollan rakennetta on myös jaoteltu luokkiin liian vähän. Connection-luokka pitää sisällään käytännössä kaiken paitsi UDP-paketin lähetyksen ja vastaanoton. Eri ominaisuuksia olisi voinut pitää omissa luokissaan ja esimerkiksi kompositio-mallia olisi voinut hyödyntää. Tämä olisi tehnyt koodista helpommin ylläpidettävää, mahdollistanut yksikkötestauksen helpommin sekä tehnyt uusien ominaisuuksien lisäämisestä helpompaa.

Lisäksi olisi mahdollisesti voinut kehittää kadonneen paketin uudelleenlähetystä. Kadonneen paketin olisi voinut mahdollisesti upottaa toisen paketin sisään, jolloin lähetettävien pakettien määrä pysyy samana. Tämä saattaisi helpottaa yhteyden rasitusta erityisesti ruuhkanhallinta tilanteissa. Tällä hetkellä virtuaaliyhteys saattaa poikkeuksellisen huonoissa olosuhteissa pahentaa tilannetta, jos paketteja lähetetään jatkuvasti uudestaan, sillä ylimääräiset paketit ruuhkauttavat reitittimiä entisestään.

Lähteet

1 W. Richard Stevens (2001) TCP/IP Illustrated Volume 1

2 Mikä on TCP/IP? Verkkojulkaisu. http://www.webopas.net/mika_tcpip.html (Luettu 3.4.2016).

3 Fiedler, G. Udp vs tcp. Verkkojulkaisu. http://gafferongames.com/networking-for-game-programmers/udp-vs-tcp/ (Luettu 2.2.2016).

4 Fiedler, G. Virtual Connection over UDP. Verkkojulkaisu. http://gafferongames.com/networking-for-game-programmers/virtual-connection-over-udp/ (Luettu 2.2.2016).

5 Fieldler, G. Reliability and flow control. Verkkojulkaisu. http://gafferongames.com/networking-for-game-programmers/reliability-and-flow-control/ (Luettu 2.2.2016).

6 Fiedler, G. what every programmer needs to know about game networking. Verkkojulkaisu. http://gafferongames.com/networking-for-game-programmers/what-every-programmer-needs-to-know-about-game-networking/ (Luettu 2.3.2016).

7 Darpa internet program protocol specification. Verkkojulkaisu. https://tools.ietf.org/html/rfc791 (luettu 14.3.2016).

8 F. Baker & P. Savola. Ingress Filtering for Multihomed Networks Verkkojulkaisu. ftp://ftp.rfc-editor.org/in-notes/rfc3704.txt (luettu 14.3.2016).

9 Michael, M. Exploring the anatomy of a data packet. Verkkojulkaisu. http://www.techrepublic.com/article/exploring-the-anatomy-of-a-data-packet/ (luettu 14.3.2016).

10 Hidenari, S. & Yoshiaki, H. & Hideki, S. Characteristics of UDP Packet Loss: Effect of TCP Traffic. Verkkojulkaisu. http://www.isoc.org/INET97/proceedings/F3/F3_1.HTM#Results. (Luettu 3.2.2016).

11 J. Postel. User Datagram Protocol. Verkkojulkaisu. https://www.ietf.org/rfc/rfc768.txt (luettu 14.3.2016).

12 The Client/Server Model. Verkkojulkaisu. https://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.ieak500/ieak511.htm (luettu 2.4.2016).