KARELIA-AMMATTIKORKEAKOULU
Tietojenkäsittelyn koulutusohjelma

Leo Sormunen

REAALIAIKAISEN PALVELINARKKITEHTUURIN MALLINTAMINEN JA TOTEUTUS

Opinnäytetyö
Joulukuu 2015

Karelia AMMATTIKORKEAKOULU	**OPINNÄYTETYÖ** **Joulukuu 2015** **Tietojenkäsittelyn koulutusohjelma** Tikkarinne 9 80220 JOENSUU Keskuksen puhelinnumero

Tekijä(t)
Leo Sormunen

Nimeke
Reaaliaikaisen palvelinarkkitehtuurin mallintaminen ja toteutus

Toimeksiantaja
Karelia-Ammattikorkeakoulu

Tiivistelmä

Opinnäytetyön tavoitteena oli suunnitella ja toteuttaa palvelinarkkitehtuuri Node.js-sovellusalustaa käyttäen. Opinnäytetyön toimeksiantajana toimi Karelia-Ammattikorkeakoulu.

Opinnäytetyön käytännön osassa toteutettiin Node.js-sovelluskehyksen avulla yksinkertainen palvelin reaaliaikaista tiedonsiirtoa varten. Käytännön osassa pyrittiin luomaan toimiva palvelinarkkitehtuuri, joka tukee kolmea erilaista tietoliikenneprotokollaa.

Teoriaosuudessa esitellään tietoliikenteen toimintaan liittyviä malleja ja tunnettuja tietoliikenneprotokollia. Lisäksi esitellään palvelimeen sovellettavia arkkitehtuurityylejä ja käsitellään Node.js-sovelluskehyksen ja sen tärkeiden komponenttien käyttämistä palvelinarkkitehtuurin toteutuksessa.

Opinnäytetyön tuloksena syntyi palvelinsovellus ja siihen kohdistuvaa kuormaa simuloiva asiakassovellus, joka kerää suorituskykymittauksen aikana kertynyttä dataa ja esittää ne graafisesti websovelluksessa.

Kieli	Sivuja 46
suomi	

Asiasanat

Node.js, Javascript, palvelinarkkitehtuuri, tietoliikenne

Karelia
UNIVERSITY OF APPLIED SCIENCES

THESIS
December 2015
All Degree Programmes

Tikkarinne 9
80220 JOENSUU
FINLAND
Telephone number of the centre

Author (s)
Leo Sormunen

Title
Modeling and implementation of real-time server architechture

Commissioned by
Karelia University of Applied Sciences

Abstract

The goal of this thesis was to design and implement a realtime server architechture using Node.js framework. The thesis was commissioned by Karelia University of Applied Sciences.

In the practical part of the thesis a simple application that simulates a realtime server application was implemented using Node.js framework. The goal was to create a functioning server architechture that has support for three different transport layer protocols.

The theoretical part discusses about models related to data communications and well-known data communications protocols. In addition this part discusses about common architechture styles used in server application development and using Node.js and its core components in creating a realtime server architechture.

As a result a server application and a benchmarking application that measures the load directed to the server was created. The benchmarking application gathers data during its execution and displays the data graphically in a web application.

Language

Finnish

Pages 46

Keywords

Node.js, Javascript, server architechture, data transmission

Sisältö

1 Johdanto

Opinnäytetyön toimeksiannossa tavoitteenani oli suunnitella ja mallintaa palvelinarkkitehtuuri reaaliaikaiseen tiedonsiirtoon Node.js-kirjaston avulla. Tavoitteenani oli, että palvelinta tulisi pystyä käyttämään monessa erilaisessa reaaliaikaisuutta vaativissa sovelluksessa, jotka vaativat tietynlaisia ominaisuuksia tiedonsiirrossa: esimerkiksi asiakassovellusten pitäminen ajantasalla nopeasti päivittyvästä tiedosta. Tavoitteena sovelluksessa on tarjota laajennettava pohja palvelimen ja asiakassovellusten välisten reaaliaikaiseen tiedonsiirtoon tulevissa Node.js-sovelluksissa.

Toimeksianto on teknisestä näkökulmasta kiinnostanut minua jo hyvin kauan. Taustaltani olen suuntautunut pääosin verkkosovelluskehitykseen, joten Javascript on etenkin asiakaspään websovellusten yhteydessä tullut hyvinkin tutuksi. Minulla on myös jonkin verran aikaisempaa kokemusta palvelinpään Javascript-sovellusten ohjelmoinnista, etenkin Node.js-sovelluskehyksestä, jota olen käyttänyt aikaisemmin yksinkertaisten relaatiotietokantoja käyttävien HTTP-palvelimien rakentamiseen.

Opinnäytetyön toimeksiannon tuloksena syntyi Node.js-palvelinsovellus, joka pystyy vastaanottamaan asiakasohjelmien lähettämiä viestejä kolmen erilaisen tietoliikenneprotokollan avulla. Lisäksi syntyi palvelimen ja verkon suorituskykyä testaava sovellus, jolla voidaan simuloida verkkoon ja palvelimelle kohdistuvaa kuormaa erilaisin parametreilla. Sovellus luo valitun määrän asiakassovelluksia, jotka kommunikoivat palvelinsovelluksen kanssa. Asiakassovellusten suorituksen aikana kerätään dataa sovellusten toiminnasta mm viestien lähetys- ja saapumisnopeuksista, sekä lähetettyjen ja vastaanotettujen pakettien lukumäärästä ja esittää ne graafisesti kaaviona erillisessä websovelluksessa.

Tässä opinnäytetyössä kuvataan palvelinarkkitehtuuriin, tietoliikenteeseen ja opinnäytetyön toimeksiannon teknisiin valintoihin liittyviä asioita, jotka ovat hyvin keskeisiä toteutetun palvelimen arkkitehtuurin näkökulmasta.

2 Tietoliikennemallit

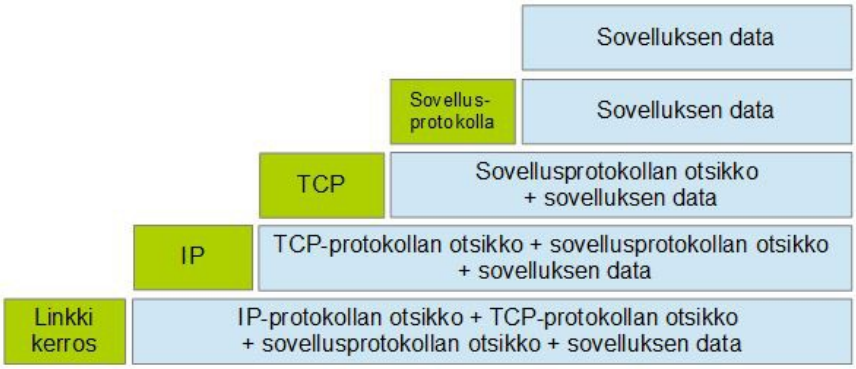
Verkolla tarkoitetaan tietokoneita tai muita laitteita, jotka ovat yhteydessä toisiinsa lähiverkon kautta. Internetillä tarkoitetaan verkkojen verkkoa, jossa yksittäiset verkot ja etenkin niiden laitteet voivat olla yhteydessä toisiinsa. Verkon ja internetin tiedonsiirron ymmärtämiseksi sen erilaiset toiminallisuudet voidaan jakaa useaan kerrokseen, jotka koostuvat ohjelmallisista tai laitteellisista elementeistä ja joilla on jokaisella oma merkityksensä verkon kokonaisuuden kannalta. (Kozierok 2005a, 94–96.)

Tietoliikennettä kuvataan kerroksittaisena mallina, jonka alin osa kuvaa pohjimmillaan laitteistoa ja ylös tultaessa mennään lähemmäksi loppukäyttäjän omin silmin näkemää ohjelmaa. Kerrosmallin yksittäinen kerros käyttää sitä alemman kerroksen tarjoamia palveluita oman toimintansa mahdollistamiseksi ja tarjoaa sen yläpuolella olevalle kerrokselle omia palveluitaan tueksi erilaisten menetelmien, protokollien avulla. Mitä ylemmäs kerrosmallia noustaan, sitä vähemmän yksittäisen kerroksen tarvitsee tietää alemman kerroksen toiminnasta, esimerkiksi siitä, että miten verkon kautta lähetettävä viesti fyysisesti lähetetään kohdetietokoneelle tai sen käyttämistä protokollista. Kun taas laskeudutaan alemmas, yksittäisen kerroksen ei tarvitse välittää esimerkiksi lähetetyn viestin sisällöstä. (Kozierok 2005a, 94–96.)

Verkossa viestiä lähettäessä jokainen kerros välittää sitä alemmalle kerrokselle kerroksen käyttämän protokollan mukaisen viestin. Koska lähetettävän viestin muoto vaihtuu kerrosten välillä käytettävän protokollan mukaan, käytetään kerrosten kehyksen yksikkönä PDU:ta (protocol data unit), eli protokollan datayksikköä. Jokainen kerros täydentää alaspäin tultaessa ylemmältä tasolta saatua datayksikköä omilla otsikkotiedoillaan käytetyn protokollan mukaisesti ja välittää otsikkotiedoista ja datasta koostuvan kehyksen alemmalle kerrokselle (kuvio 1). Tätä prosessia kutsutaan kapseloinniksi. Alimmalle kerrokselle tultaessa kaikkien ylinten kerrosten kehysyhdistelmä muutetaan biteiksi ja välitetään fyysisesti määriteltyyn kohteeseen. Kehystä vastaanottavassa

päässä prosessi on sama kuin lähettäessä, mutta käänteisenä. (Kozierok 2005a, 94–96.)

IP-verkkojen toiminnassa välisessä on isossa roolissa reititys. Reitityksellä tarkoitetaan prosessia, jossa verkko ottaa vastaan tulevia paketteja ja tarkistaa IP-protokollan otsikkotiedoista minne paketti on osoitettu. Jos kohdetietokone löytyy samasta verkosta, välitetään viesti kohteeseen. Jos kohdetta ei löydy verkosta, reititin ohjaa sen seuraavaan verkkoon ja sama prosessi toistuu, kunnes kohde löytyy. (Kozierok 2005a, 98–99.)



Kuvio 1. Datan kapselointi kerroksittain TCP/IP-mallissa.

Tietoliikennemalleja ovat OSI (Open Systems Interconnection Reference Model)-malli ja TCP/IP-malli. Vaikka malleissa on eri määrä kerroksia, niin niiden perusperiaate on sama. Erot muodostuvat siitä, mihin kerrokseen protokolla ja sen tarjoamat toiminallisuudet mielletään mallissa kuuluvan. Tässä luvussa kuvataan OSI-mallin ja TCP/IP-mallin periaatteita.

2.1 OSI-malli

ISO:n (International Standards Organization) OSI-mallin (Open Systems Interconnection) tarkoituksena on kuvata arkkitehtuuri, joka mahdollistaisi sujuvan kommunikoinnin arkkitehtuuria käyttävien sovellusten kesken verkossa. Vaikka mallia ei sellaisenaan käytetä, sen kerrosajattelua pidetään tietoliikenteen perustana. OSI-malli koostuu seitsemästä kerroksesta (kuvio 2),

sovellus-, esitys-, istunto-, kuljetus-, verkko-, siirto- ja fyysisestä kerroksesta. (Kaarlo 2002, 18.)

OSI-mallin alin kerros, fyysinen kerros huolehtii siitä, miten sovelluksesta lähetettävä data siirtyy fyysisesti internetin kautta kohdetietokoneelle. Se muuntaa lähetettävän datan kaikki osat binääriluvuiksi ja välittää sen langallisesti tai langattomasti. Fyysistä kerrosta lukuun ottamatta OSI-mallin kerrokset toimivat pääsääntöisesti ohjelmallisesti. (Kaarlo 2005, 20.)

Kuvio 2. OSI-mallin kerrokset.

Seuraavan kerroksen nimi on siirtokerros, jonka tehtävänä on yhdistää lähiverkkoon yhdistyneet tietokoneet toisiinsa ja mahdollistaa bittivirran siirto luotettavasti. Kerros tarkistaa siirrettävien bittien oikeellisuuden ja vuon valvonta. Vuon valvonnalla tarkoitetaan mahdollisuutta rajoittaa tarpeen mukaan fyysiselle kerrokselle siirrettävän datan määrää. (Kaarlo 2005, 20.)

Verkkokerroksen päätehtävänä on reititys. Verkkokerros mahdollistaa yhdessä fyysisen ja datayhteyskerroksen kanssa usean verkon yhdistäminen internetiksi ja näiden välisten datayhteyksien luonnin. Jokaisella verkkojen välillä tietoa siirtävällä laitteella on osoite, jolla laite tunnistetaan. Internetin kautta tietoa siirtävä laite käyttää yleensä IP-protokollaa ja tällöin laitteen yksilöivä tunnus, osoite, on IP-osoite. (Kozierok 2005a, 102–112.)

OSI-mallin neljännen kerroksen nimi on kuljetuskerros. Kuljetuskerros vastaa verkon kautta lähettävän datan läpinäkyvästä kuljetuksesta lähettäjän ja

vastaanottajan välillä. Kuljetuskerros toimii linkkinä alempien datan siirtämisestä vastaavien kerrosten ja sovellusprosessien välisessä kommunikoinnissa. (Kaarlo 2002, 21.) Kerros yksilöi tulevat viestit porttinumeroiden perusteella, joihin viestit on ohjattu menemään niin, että viesti saapuu lopulta oikeaan sovellusprosessiin (Kozierok 2005a, 102–112).

Seuraavan kerroksen nimi on istuntokerros, joka mahdollistaa ja ylläpitää sovellusprosessien välisen kommunikoinnin istunnon aikana. Seuraava kerros, esitystapakerros huolehtii siitä, että data on samanmuotoista laitteen ominaisuuksista, kuten käyttöjärjestelmästä riippumatta. (Kozierok 2005a, 102–112.)

OSI-mallin ylin kerros, sovelluskerros toimii lopullisena rajapintana käyttäjän sovelluksen ja tiedonsiirron välillä. Se määrittelee verkkoa käyttäville sovelluksille yhteisen kommunikointirajapinnan verkossa. Esimerkiksi sähköpostisovellus käyttää sovelluskerroksen protokollanaan SMTP:tä (Simple Mail Transfer Protocol*),* jota käytetään sähköpostiviestien lähettämiseen sähköpostipalvelimien välillä. (Kaarlo 2002, 21.)

2.2 TCP/IP-malli

TCP/IP on neljään kerrokseen jakautuva malli (kuvio 3). Sen nimi juontaa juurensa sen kahdesta keskeisestä protokollasta IP-protokollasta (Internet Protocol) ja TCP-protokollasta (Transmission Control Protocol) ja se koostuu niiden lisäksi lukuisista muista protokollista. IP ja TCP ovat keskeisiä osia TCP/IP-mallissa, koska mallin olennaisimmat toiminnot tehdään kerroksissa, joissa nämä protokollat ovat käytössä. (Kozierok 2005a, 128–130.)

TCP/IP-mallissa ei oteta tarkasti kantaa fyysisiin laitteisiin, toisin kuin OSI-mallin alimmalla tasolla, vaan TCP/IP-mallin alimman kerroksen verkkorajapinnan ajatellaan hoitavan datan lähetyksen fyysisten laitteiden avulla. Mallin alimman kerroksen tehtävä on yksinään huolehtia yhteyksistä lähiverkon muihin laitteisiin. Kerroksessa ei ole käytössä mitään IP-protokollan tehtäviä. (Kozierok 2005a, 128–130.)

Kuvio 3. TCP/IP-mallin kerrokset.

TCP/IP-mallin toisen kerroksen nimi on internet-kerros ja sen tehtävänä on laitteistojen paikannus osoitteiden perusteella, datan pakkaus, lähetys ja reititys. Tason pääasiallinen protokolla on IP-protokolla, sekä tukiprotokollat kuten IMCP (Internet Control Message Protocol) ja RIP (Routing Information Protocol). (Kozierok 2005a, 128–130.)

Kolmannen kerroksen, kuljetuskerroksen, tehtävänä on huolehtia laitteilla tapahtuvien prosessien välisestä datan lähetyksestä luotettavasti tai epäluotettavasti. Luotettavalla lähetyksellä tarkoitetaan lähetystä, jossa lähetyksestä huolehtiva protokolla valvoo lähtevää ja vastaanotettavaa dataa ja pitää huolen, että data todella vastaanotetaan ja vastaanotetaan samanlaisena kuin se on lähetetty ja lähetetään uudelleen, jos data on vahingoittunut. Epäluotettavalla lähetyksellä tarkoitetaan lähetystä, jossa edellä mainittua valvontaa ei ole. Luotettavaa protokollaa edustaa TCP (Transmission Control Protocol) ja epäluotettavaa protokollaa UDP (User Datagram Protocol). (Kozierok 2005a, 128–130.)

Kuljetuskerroksen protokollia käyttävät protokollat näkevät verkon loogisina yhteyksinä ja ne muodostetaan käyttämällä socketteja. Socketilla tarkoitetaan IP-osoitteen ja porttinumeron yhdistelmää. IP-osoitteen perusteella voidaan yksilöidä kohdelaite verkossa ja porttinumerolla yksilöidään kuljetuskerroksen yläpuolella olevan sovelluskerroksen sovellusprosessi. (Kaarlo 2002, 22.)

TCP/IP-mallin ylimmän kerroksen nimi on sovelluskerros ja siinä sijaitsee kaikki prosessit, jotka käyttävät kuljetuskerroksen protokollia tietojen siirtoon. Sovellusprotokollat tuottavat palveluja sovellusten käyttäjille. Protokollia ovat esimerkiksi HTTP, jota käytetään websivujen lukemiseen ja SMTP, jota käytetään sähköpostin välitykseen. (Hunt 1998, 21–22.)

3 Tietoliikenneprotokollat

Tietoliikenneprotokollat voidaan jakaa yhteydellisiin ja yhteydettömiin protokolliin sekä luotettaviin ja epäluotettaviin protokolliin.

Yhteydellisellä protokollalla tarkoitetaan sitä, että kommunikoivat laitteet muodostavat yhteyden toisiinsa. Tyypillisesti kommunikoinnin aloittava laite lähettää pyynnön yhteydestä ja vastaanottava pää reagoi viestiin vastaamalla siihen. Laitteet sopivat yhteyden muodostuksesta, miten yhteyttä hallinnoidaan ja miten se suljetaan. Vasta tämän toimenpiteen jälkeen varsinainen kommunikointi, esimerkiksi tiedostojen lähetys laitteiden välillä, voidaan aloittaa. Yhteydetön protokolla ei vaadi edellä mainittuja toimenpiteitä kuten yhteydellinen protokolla, vaan paketit lähetetään sellaisenaan ilman istunnon muodostamista. (Kozierok 2005a, 15–16.)

Epäluotettavalla tiedonsiirtoprotokollalla tarkoitetaan protokollaa, johon ei itsessään sisälly palveluita pakettien vahvistukselle sitä vastaanottaessa tai uudelleenlähetykselle, jos paketin vastaanottovahvistusta ei ole saatu. Luotettavasta protokollasta tällaiset palvelut löytyvät. (Hunt 1998, 17–18.)

Verkon yli välitettyjen pakettien, kehysten rakenne vaihtelee protokollasta riippuen. Yleensä paketti jaetaan otsikkotietoihin ja dataosaan. Otsikkotiedot sisältävät usein pieniä määriä kontrollitietoa siitä, miten dataosaa tulee tulkita ja miten sitä pakettia käytetään. Dataosa koostuu käyttäjän lähettämästä sanomasta. (Kozierok 2005a, 19.)

3.1 IP

IP (Internet Protocol) on internetin toiminnan keskiössä. IP on yhteydetön protokolla, eli ennen pakettien siirtoa kahden Internetiin yhdistyneen laitteen välille ei muodosteta yhteyttä. IP on myös epäluotettava protokolla, sillä protokollaan ei kuulu virheentarkistusta tai -korjausta. IP-protokolla luottaa siihen, että sen yläpuolella olevilla kerroksilla sijaitsevat protokollat hoitavat yhteyden muodostuksen tai luotettavuuden, jos niitä tarvitaan tiedonsiirrossa. (Hunt 1998, 12.)

IP-protokollan paketin muoto on nimeltään tietosähke, joka lähetetään lukemalla tietosähkeen kehyksen (kuvio 4) otsikkotiedoista kohdeosoite, joka on IP-osoite, joka kertoo kohteen verkon sekä verkossa sijaitsevan laitteen. Jos kohde on samassa verkossa lähettäjän kanssa, lähetetään tietosähke suoraan kohteeseen. Eri verkossa sijaitseva paketti välitetään yhdyskäytävään, joka reitittää tietosähkeen fyysisestä verkosta toiseen. (Hunt 1998, 12–13.)

Kuvio 4. IP-tietosähkeen kehysrakenne

IP-protokollan otsikkotiedot alkavat versionumerosta, joka on joko 4 tai 6. Seuraavana kenttänä on otsikon pituus, joka kertoo 32 bitin sanojen lukumäärän otsikossa. TOS-bittien avulla ilmoitetaan tietosähkeen reititykselle annettu laatuvaatimus. Laatuvaatimusvaihtoehtoja ovat viive, läpäisy, hinta, luotettavuus ja vain yksi niistä voi olla kerrallaan päällä. Esimerkiksi viive-bitin ollessa päällä pyydetään minimoimaan paketin viivettä ja luotettavuus-bitin ollessa päällä halutaan paketin luotettavaa toimitusta. Ilman laatuvaatimuksia lähetetty tietosähke on ns. ”best effort”-tietosähke, joka on tavallisin tapa tietoliikenteessä. (Kaarlo 2002, 47.)

Kehyksen kokonaispituus kertoo otsikkotietojen ja dataosuuden yhteispituuden, joka ilmoitetaan oktetteina. Vähentämällä kokonaispituudesta otsikon pituuden saadaan selville datan alkamiskohta tietosähkeessä. Tunnistekentän avulla tunnistetaan mistä ylemmän kerroksen protokollan datasta tietosähke on peräisin. (Kaarlo 2002, 48.)

Seuraavaksi tulee lippu-kenttä, jonka arvo voi olla ”M” (”More”) tai ”D” (”Do not fragment”). ”M” indikoi, että tietosähke on ositettu (fragmentation) ja sitä seuraa vielä paketteja, jotka ovat ositettu isommasta tietosähkeestä. ”D” ilmoittaa, että tietosähkettä ei enää saa osittaa. Seuraava kenttä, ”Fragment offset” ilmoittaa ositettujen tietosähkeiden järjestyksen. Kenttää tarvitaan, kun tietosähkeitä kootaan yhdeksi vastaanottopäässä. (Kaarlo 2002, 48.) Maximum Transmission Unit (MTU) määrää lähetettävän paketin maksimikoon kussakin verkossa. Jos verkon kautta kulkevan lähetettävän tietosähkeen koko on suurempi kuin verkon MTU, joudutaan tietosähke osittamaan. (Hunt 1998, 15–16.)

Oleellisina osana IP-protokollan toimintaa on reititys. Reititin toimii kahden verkon välisenä rajapintana ja sen tehtävänä on osoittaa ja ohjata tulevat tietosähkeet seuraavaa reititintä tai kohdetietokonetta varten, jos tämä löytyy verkosta. (Kozierok 2005b.) TTL-, eli ”Time to live”-kenttä ilmoittaa kuinka monen reitittimen kautta tietosähke voi kulkea ennen kuin se tuhotaan. TTL:n arvo on yleensä lähettäessä 255 ja sitä vähennetään aina yhdellä jokaista reitittävää laitetta kohden. Kun arvo on 0, paketti tuhotaan. Tällä estetään ikuisesti verkossa kohdetta etsivät tietosähkeet. (Kaarlo 2002, 49.)

Tietosähkeiden siirto kuljetuskerroksen protokollalle tapahtuu tietosähkeen otsaketietojen protokollan tunniste -tiedon perusteella. Kuljetuskerroksen protokollilla on oma yksilöivä numero, jolla IP tunnistaa ne. (Hunt 1998, 16.) Esimerkiksi TCP-protokollan protokollanumero on 6 ja UDP-protokollan 17. Tietosähke ohjataan kuljetusprotokollalle, joka kapseloi sen ja ohjaa sen oikealle sovellusprosessille porttinumeron perusteella. (Hunt 1998, 42.)

IP-protokollan tarkistussumma-kenttään lasketaan tietosähkeen otsikkotiedoista ja sen avulla vastaanottaja pystyy tarkistamaan, etteivät otsikkotiedot ole muuttuneet lähetyksen aikana. Seuraavat kentät ovat tietosähkeen lähde- ja

kohdeosoitteet, jotka ovat lähettäjän IP-osoite ja vastaanottajan IP-osoite. (Kaarlo 2002, 50.)

IP-protokollan versio 4 mahdollistaa unicast-, multicast- ja broadcast-tyyppisten tietosähkeiden lähetysmuotojen käytön. Unicast on lähetysmuoto, jossa tietosähkeet lähetetään kahden osapuolen välillä. Broadcast-tyyppiset tietosähkeet lähetetään kaikille verkkoon liittyneille osapuolille. Multicast-, eli ryhmälähetykset lähetetään kaikille ryhmään kuuluville osapuolille ja niitä. Ryhmään liitytään multicast-osoitteella ja kaikki tähän osoitteeseen lähetetyt viestit välitetään kaikille ryhmään kuuluville. Ryhmään kuuluvat osapuolet voivat sijaita eri verkoissa. (Kaarlo 2002, 136.)

3.2 TCP

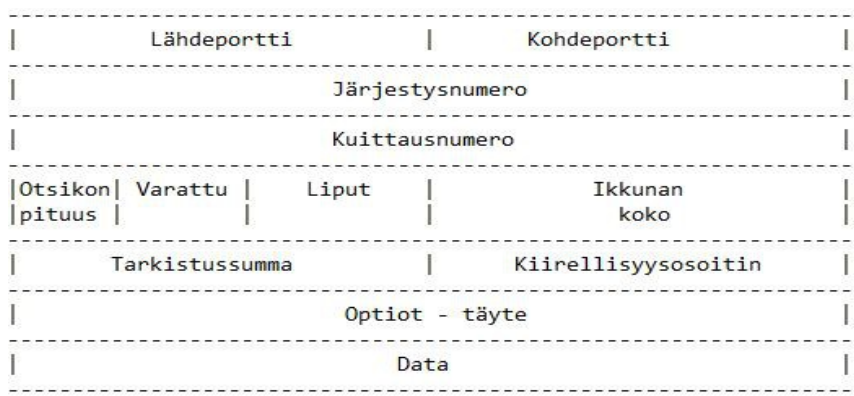
TCP (Transmission Control Protocol) tarjoaa sitä käyttäville sovelluksille luotettavan ja yhteydellisen datan kuljetuksen. TCP-protokollaa käyttävien laitteiden pakettien vaihto alkaa aina ”kättelyn” avulla, jossa yhteys neuvotellaan ennen varsinaista pakettien vaihtoa. (Kaarlo 2005, 166.) TCP-protokollassa lähetysmuoto on unicast-tyyppinen, eli yhteydet ovat aina kahden laitteen välisiä (Kaarlo 2005, 59).

TCP-protokollaa käyttävät sovellukset lähettävät ja ottavat vastaan itsenäisten pakettien sijaan tavuvirtoja (byte stream). Prosessit kirjoittavat ja varastoivat lähetettävän datan TCP:n lähetyspuskuriin ja lukevat sitä vastaanottopuskurista. Kuljetuskerroksesta alemmille kerroksille välitettävästä datasta TCP-protokollasta puhuttaessa käytetään nimeä segmentti, joka on tavuvirran osa. (Kaarlo 2005, 166.)

Lähetyspuskuriin talletettu data pilkotaan useaksi segmentiksi ja segmenttien sisällä dataokteteiksi ja IP välittää nämä kohdetietokoneeseen. Jokaiselle siirrettävälle dataoktetille annetaan järjestysnumero. Vastaanotettu segmentti käsitellään sen sisältämien osien järjestysnumeron mukaan ja kuitataan vastaanottavassa päässä. Tieto kuittauksesta lähetetään lähettävälle osapuolelle. Kuittausta ei lähetetä, jos vastaanotetun segmentin data on

vahingoittunut, jäänyt vastaanottamatta tai se on kopioitunut tai vastaanotettu väärässä järjestyksessä. Lähettävä osapuoli odottaa kuittauksen vastaanottoa ja jos sitä ei saada edellä mainituista syistä, siirretään segmentti ajastetusti uudelleen. (Postel 1981.) Vastaanotetun segmentin eheys ja muuttumattomuus tarkistetaan segmentin otsikkotiedoista löytyvän tarkistussumman perusteella (Hunt 1998, 18).

TCP-protokollan segmentin kehys (kuvio 5) on vähintään 20 tavun pituinen ja maksimissaan 60 tavua. Segmentti koostuu lähetettävän datan lisäksi lähdeportin ja kohdeportin numerosta. Segmentin järjestysnumerokenttä kertoo segmenttien ja dataoktettien lähetysjärjestyksen. Kuittausnumero kertoo seuraavaksi vastaanotettavan paketin järjestysnumeron ja se on aina viimeksi vastaanotetun segmentin viimeisen dataoktetin järjestysnumero plus yksi. Seuraava kenttä osoittaa otsikon pituuden ja sitä seuraavat 6 bittiä ovat ”varattuja tulevaisuuden varalta”. Seuraavaksi on tarkistussummakenttä, joka lasketaan TCP-segmentistä ja IP-kehyksen otsikkotiedoista: kohde- ja lähdeosoitteista, protokollatunnisteesta ja pituustiedoista. (Kaarlo 2005, 166.)



Kuvio 5. TCP-segmentin kehysrakenne.

TCP-protokollan otsikkotietoihin kuuluu myös ns. liput. ”URG”-lippu kertoo datan kiireellisyydestä, mutta se ei velvoita vastaanottajaa tekemään sen edistämiseksi mitään. ”ACK”-lippu ilmoittaa kuittausnumeron pätevyyden. PSH-lippu vaatii toista osapuolta lähettämään kaiken TCP-protokollan lähetyspuskurissa olevan datan lähetystä tilanteissa, joissa datavirrassa lähetyksessä on tauko. RST-lippu ilmoittaa yhteyden uudelleenaloituksesta. SYN-lippu synkronoi sekvenssinumerot ja FIN-lippu ilmoittaa, että osapuolella ei ole enää dataa lähetettävänä. (Kaarlo 2005, 169.)

TCP-protokollan ominaisuuksiin kuuluu vuonvalvonta, eli TCP tarkkailee verkon tilaa ja säätelee lähetysnopeutta verkon suorituskyvyn mukaan. Tämän Vuonvalvonnassa tärkeässä roolissa on ns. ”liukuva ikkuna”, johon kuuluu segmenttien järjestysnumerointi. Ikkuna on yksi TCP-segmentin otsikkotiedoista ja se ilmoittaa toiselle osapuolelle, kuinka paljon dataa ollaan valmiita vastaanottamaan. (Kaarlo 2005, 166–169.)

TCP-protokollan lisäksi useimmissa tietoliikenneprotokollissa yhteyden muodostus tehdään kolmivaiheisen kättelyn avulla, jossa kaksi laitetta lähettävät yhteensä kolme segmenttiä toisilleen yhteyden muodostamiseksi. Aloitteen tekevä osapuoli A lähettää segmentin B:lle ja B tulkitsee sen yhteyden aloitukseksi sen otsikkotiedoista löytyvän SYN (Synchronize sequence numbers) -lipun avulla. B näkee siitä samalla, mikä on segmentin A:n ensimmäisen vastaanotettavan segmentin järjestysnumero. B vastaa tähän segmentillä, josta löytyy sekä SYN- että ACK (Acknowledgement) -lippu. Tällä vahvistetaan A:lle, että B on ottanut vastaan ja huomioinut A:n aloitteen. Tämän jälkeen A lähettää vielä oman ACK-lipullisen segmentin B:lle, jonka jälkeen tiedonsiirto voi alkaa. (Hunt 1998, 18–19.)

TCP-protokollassa yhteyden sulkeminen tapahtuu toisen osapuolen aloitteesta. Kuten kättelyssä yhteyttä muodostaessa, lähettää yhteyden sulkeva osapuoli A FIN-lipullisen viestinsä osapuolelle B, johon A odottaa kuittausta. Tämän jälkeen B lähettää oman FIN-lipullisen viestin ja jää odottamaan A:n vastausta tähän, jonka jälkeen yhteys puretaan. Tällä pyritään varmistamaan, että kummatkin osapuolet ovat valmiita yhteyden sulkemiseen, ettei dataa olla enää lähettämässä eikä vastaanottamassa. Näin myös estetään datan katoaminen. (Kaarlo 2005, 176.)

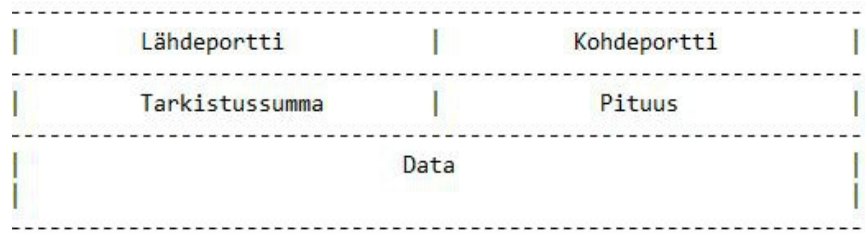
TCP sopii sovelluksille, joissa halutaan olla varmoja siitä, että lähetettävä data vastaanotetaan oikeassa järjestyksessä ja luotettavasti. TCP osaa rajoittaa omaa lähetysnopeuttaan ongelmatilanteissa, eikä tällöin ruuhkauta verkkoa liikaa. Reaaliaikaisuutta vaativissa palvelinsovelluksissa TCP-protokollassa on ongelmallisia ominaisuuksia. Etenkin virhetilanteissa, kuten verkkoyhteyden katketessa hetkellisesti TCP säilyttää lähetettävää dataa lähetyspuskurissaan ja kun katkos päättyy, jatkaa se lähetystä siitä mihin ennen katkoa jäätiin, jolloin

TCP joutuu lähettämään mahdollisesti jo vanhentunutta dataa. (Kaarlo 2005, 186–187.)

3.3 UDP

UDP on yhteydetön ja epäluotettava tiedonsiirtoprotokolla. UDP-tietosähkeiden lähetys ei vaadi yhteyden muodostusta vaan ne lähetetään itsenäisinä, toisistaan irrallisina paketteina kohteeseen. Vastaanotettujen tietosähkeiden vastaanottoa ei kuitata lähettäjälle eikä niiden lähetysjärjestystä valvota. (Hunt 1998, 17) UDP-tietosähkeiden lähetysmuoto voi olla unicast- multicast-, broadcast- tai anycast-tyyppinen. (Kaarlo 2002, 58.)

UDP-tietosähkeen kehys (kuvio 6) koostuu välitettävän sanoman lisäksi lähettävän sovelluksen porttinumerosta, vastaanottavan sovelluksen porttinumerosta, tarkistussummasta ja datan pituuden ilmoittavasta kentästä. Tietosähkeet ohjataan oikean sovelluksen käytettäväksi porttinumeron perusteella. Tarkistussumman avulla tarkistetaan lähetetyn tietosähkeen eheys ja muuttumattomuus. (Kaarlo 2002, 157.)



Kuvio 6. UDP-tietosähkeen kehysrakenne.

UDP sopii erityisen hyvin sovelluksiin, joiden toiminnan kannalta yksittäisten pakettien häviäminen lähetyksen aikana ei ole merkityksellistä. UDP-protokolla käyttävien sovellusten data voi olla hyvin nopeasti vanhentuvaa, eikä tällöin kadonneen paketin uudelleenlähetys ole välttämättä tarpeellista. (Kaarlo 2002, 156.) UDP on TCP-protokollaan verrattuna erityisen tehokas tapauksissa, joissa lähetettävien pakettien yhteenlaskettu koko on niin pieni, että yhteyden luominen itsessään käyttäisi enemmän resursseja kuin haluttujen pakettien välitys. UDP-protokollaa käyttävät sovellukset voivat myös itse varmistaa

esimerkiksi viestin vastaanoton vahvistuksen lähettävälle osapuolelle ja lähettävässä päässä uudelleen lähetyksen määrätyn ajan kuluttua, jos vastausta ei ole saatu. (Hunt 1998, 17–18.)

3.4 Websocket

Websocket-protokolla suunniteltiin mahdollistamaan kaksisuuntainen viestintä websovelluksissa asiakkaan ja palvelimen välillä, joka aiemmin mahdollista toteuttaa tekemällä jatkuvasti toistuvia HTTP-pyyntöjä palvelimelle tiedon lähetystä ja vastaanottoa varten. Toistuva erillisten pyyntöjen lähettäminen on raskasta palvelimelle, sillä jokaisella pyynnöllä on varsinaisen viestin lisäksi pyynnön omat HTTP-otsikkotiedot. Websocket-protokollien avulla edellä mainittu kaksisuuntainen viestintä voidaan hoitaa yhden TCP-yhteyden avulla webselaimen ja palvelimen välillä. (Fette & Melnikov 2011.)

Protokollaan kuuluu TCP-protokollan tapaan kättelyvaihe ja sen jälkeen datan vaihto. Kättely alkaa asiakkaan lähettämästä HTTP-pyynnöstä Websocket-protokollaa tukevaan palvelimeen. Kun palvelin ja asiakas ovat suorittaneet oman osansa kättelystä onnistuneesti, kommunikointi voi alkaa. Palvelimen ja asiakkaan välinen yhteys on kaksisuuntainen, jossa kumpikin osapuoli voi vastaanottaa ja lähettää dataa samanaikaisesti. Osapuolten välillä liikkuva data koostuu viestimuotoisista kehyksistä, jotka lähetetään yksinään tai osissa. Kehysten data voi olla UTF-8 muotoisia merkkijonoja, binäärimuotoista dataa tai protokollan toiminnan vaatimia hallinnallisia kehyksiä, jotka eivät kuljeta varsinaista sovellusdataa vaan viestivät esimerkiksi osapuolen halusta lopettaa avattu yhteys. (Fette & Melnikov 2011.)

Asiakassovelluksen lähettämä yhteyden avauspyyntö on osa kolmivaiheista kättelyä. Aloitteen tekevä pyyntö on yhteensopiva HTTP-protokollaan perustuvien palvelinohjelmistojen kanssa. Siten esimerkiksi websovelluksen käyttämää samaa porttinumeroa voidaan käyttää myös websocket-protokollan yhteydenpitoon, eikä palvelimen tarvitse varata erikseen omaa porttinumeroa websocket-yhteyksiä varten. Lähetettävä pyyntö on HTTP upgrade-pyyntö.

Palvelin vastaa tähän lähettämällä oman HTTP-upgrade vastauksen. Tämän jälkeen HTTP-yhteys korvataan websocket-yhteydellä. (Fette & Melnikov 2011.)

Websocket-protokolla on TCP-protokollaan pohjautuva protokolla, mutta toisin kuin TCP-protokollan tiedonsiirrossa, sovellukset kommunikoivat lähettämällä kehyksiä. Websocket-kehyksen ensimmäinen kenttä on FIN-bitti, joka kertoo, onko kyseinen kehys kokonaisen viestin viimeinen osa, jos viesti on lähettäessä ositettu. 3 RSV-kenttää on varattu tulevaisuutta varten. Jos vastaanotetun kehyksen RSV-kentät eivät ole arvoltaan 0, protokolla määrää, että yhteys on suljettava. (Fette & Melnikov 2011.)

”Opcode”-kentän arvo määrittelee miten kehystä tulee tulkita. Arvo voi mm. indikoida että kehys on jatkoa toisesta kehyksestä, kehyksen data osuus sisältää sovelluksen dataa: UTF-8-muotoisen merkkijonon, binäärimuotoista dataa tai osapuolen halua sulkea nykyinen yhteys. ”Opcode”-kentän ”ping”-arvoisella kehyksellä pyydetään toista osapuolta vastaamaan viestiin kehyksellä, jonka ”Opcode”-kentän arvo on ”pong”. ”Ping”- ja ”pong” arvoisilla kehyksillä voidaan pitää yhteyttä yllä ja toisaalta varmistetaan, että toinen osapuoli on yhä toimintakykyinen ja pystyy vastaamaan, eikä esimerkiksi kaatunut verkkovirheen vuoksi. Websocket-protokolla tekee tämän automaattisesti. (Fette & Melnikov 2011.)

”Mask”-kenttä kertoo tuleeko ”masking-key”-kentällä olla arvo. ”Payload length”-kenttä kertoo data-osuuden pituuden. ”Masking-key”-kenttään arvotaan satunnainen 32-bittinen arvo, jonka perusteella naamioitu data-osuus saadaan luettua. ”Payload”-kenttä koostuu datasta, jonka pituus on määritelty ”Payload length”-kentässä (Fette & Melnikov 2011.)

Websocket-yhteyden sulkeminen tapahtuu toisen osapuolen aloitteesta lähettämällä kehys, jonka ”Opcode”-arvo on ”Close”. ”Close”-kehyksen vastaanottava osapuoli vastaa viestin ja lähettää oman ”Close”-viestinsä vastauksena ja kun tämä saadaan vastaanotettua, TCP-yhteys voidaan sulkea. Kehyksen mukana voidaan välittää viesti esimerkiksi UTF-8-muotoinen tekstimuotoinen syy yhteyden sulkemiselle. ”Close”-kehyksen vastaanottava osapuoli voi lykätä vastauksen lähetystä, jos sillä on esimerkiksi viestin lähetys kesken. (Fette & Melnikov 2011.)

4 Arkkitehtuurityylit

4.1 Asiakas–palvelin-arkkitehtuuri

Asiakas–palvelin-malli on palveluperustainen arkkitehtuurityyli. Palvelulla tarkoitetaan jonkin resurssin hallintaa keskitetysti tarjoavaa palveluntarjoajaa eli palvelinta jota palvelun käyttäjät eli asiakkaat käyttävät. Palvelin ja asiakas toimivat usein omissa sovellusprosesseissaan ja fyysisesti omilla laitteellaan. Palvelimen ja asiakkaan välinen istunnon aloitus, kommunikaatio ja istunnon lopetus tapahtuu usein asiakkaan aloitteesta. Asiakas–palvelin-malliin kuuluu selvä roolijako, joka määrittelee asiakkaan ja palvelimen välisen suhteen sekä niiden prosessien tehtävät. Palvelin voi kuitenkin käyttää omassa toiminnassaan jonkin toisen palveluntarjoajan palveluita, jolloin palvelin on tässä tapauksessa myös asiakas. (Koskimies & Mikkonen 2005, 136–138.)

Asiakas- ja palvelinsovellukset suunnitellaan yleensä toisistaan riippumattomaksi. Esimerkiksi asiakassovelluksen kohdatessa ongelmatilanteen muiden asiakassovellusten pyyntöihin vastaaminen voi jatkua normaalisti. Palvelimen kaatuminen taas estää palvelun käytön kaikilta asiakkailta. Palvelimeen kohdistuvan kuorman kasvaessa palvelimeen kohdistuvaa kuormaa voidaan jakaa esimerkiksi usean palvelinprosessin kesken. (Koskimies & Mikkonen 2005, 137–138.)

Asiakas–palvelin-mallin etuna on tehtävien hajautus. Palvelin tarjoaa palveluita ja asiakas ottaa pyytää ja ottaa niitä vastaan tietämättä tarkemmin, miten palvelu toimii tai miten pyyntö käsitellään ja lähetetään takaisin. Vastaavasti palvelimen ei tarvitse tietää, miten vastauksen tietoja käsitellään. Palvelinsovelluksen asiakassovellusten kanssa jaettu tieto löytyy yhdestä paikasta, palvelimelta ja palvelin voi itse rajata, mihin tietoihin asiakassovelluksilla on oikeus päästä käsiksi. (Jia & Zhou 2005.)

Koska mallissa kaikkea tietoa hallinnoidaan palvelimelta käsin, koko verkon toiminnan luotettavuus perustuu siihen, että palvelimen toiminta ei hidastu tai että palvelin ei kaadu virheen vuoksi. Palvelimeen kohdistuva kuorma kasvaa mitä enemmän sillä on esimerkiksi yhtäaikaisia käyttäjiä. Tämän seurauksena pyyntöjen ja vastausten vasteaika kasvaa. (Jia & Zhou 2005.)

4.2 Kerrosarkkitehtuuri

Kerrosarkkitehtuuri on jonkin abstrahointiperiaatteen mukaiseen nousevaan järjestykseen järjestetyistä kerroksista koostuva arkkitehtuurityyli. Abstrahoinnilla tarkoitetaan asioiden määrittelyä toimintojen ja ominaisuuksien kautta ottamatta kantaa niiden tekniseen toteutukseen. Kerrosarkkitehtuurissa abstrahointiperiaate on sellainen, että ylimmällä tasolla on yleensä käyttäjän näkemän sovelluksen toiminnot ja alaspäin mentäessä lähestytään fyysisen laitteen tarjoamia toimintoja. (Koskinen & Mikkonen 2005, 126.)

Arkkitehtuurissa yksittäisen kerroksen tulee käyttää sitä alemman kerroksen palveluita ja tarjota sitä ylemmälle kerrokselle omia palveluitaan. Jos esimerkiksi kerros 3 käyttää alemman kerroksen 2 sijaan kerroksen 1 palveluita, puhutaan ohituksesta, joka voi tapahtua esimerkiksi kun kerroksen 3 vaatima palvelu ei löydykään suoraan kerrokselta 2. Jos taas alempi kerros 1 käyttää suoraan kerroksen 3 palveluja, on varmistettava, ettei kerros 1 tule riippuvaiseksi 3:n palveluista. Kerroksilla on rajapintoja, jotka ne tarjoavat ylemmälle kerrokselle ja rajapintoja, joita ne tarvitsevat alemmalta. Rajapinnat mahdollistavat palveluiden käytön kerrosten välillä. Esimerkiksi kerroksen 1 tarjoamien rajapintojen tulisi täsmätä kerroksen 2 tarvitsemien rajapintojen kanssa ja niin edelleen ylöspäin mentäessä. (Koskinen & Mikkonen 2005, 126–128.)

Kerrosarkkitehtuurin etuna on, että sen avulla järjestelmä kokonaisuutena on helpompi ymmärtää. Kerrosarkkitehtuuri tukee hyvin järjestelmän muutoksia, jos kerrokset ovat riippuvaisia vain itseään alemmasta kerroksista. Se myös ohjaa riippuvuuksia minimoivaan ohjelmistosuunnitteluun. Arkkitehtuuri tukee myös

järjestelmän uudelleenkäyttöä, sillä uusia järjestelmiä voidaan rakentaa arkkitehtuurin alempien kerrosten päälle. (Koskinen & Mikkonen 2005, 130–131.)

Kerrosarkkitehtuurin ongelmana voidaan pitää tehottomuutta tapauksissa, joissa ylemmän kerroksen 3 tulee käyttää sen alla olevaa kerrosta 2 seuraavan kerroksen 1 palvelua. Tällöin palvelu voidaan joutua tekemään tarpeettomasti kerrokseen 2, jotta kerrosten riippuvuussuhteet saadaan pidettyä aina vierekkäisten kerrosten välisinä. (Koskinen & Mikkonen 2005, 131.)

5 Node.js

Node.js on avoimeen lähdekoodiin perustuva Javascript-sovelluskehys. Se on rakennettu Googlen kehittämän Google Chrome -selaimen V8 Javascript-moottorin päälle. V8:n tehtävänä on Javascriptin suoritus, Javascript-olioiden muistin allokointi ja automaattinen roskienkeräys. (Google Developers, 2014.) V8 on erityisesti suunniteltu suurten Javascript-sovellusten nopeaan suoritukseen. Nopeus näkyy erityisestä sovelluksissa, joissa yksittäistä funktiota toistetaan useasti. (Google Developers, 2012.)

Node.js mahdollistaa Javascript-kielen käytön palvelinsovelluksissa. Node.js:n periaatteina ovat asynkroninen keskeytymätön (non-blocking) I/O. Tämä tarkoittaa, että lukiessa tiedostoa kovalevyltä tai suorittaessa toista sovellusprosessia, sovelluksen muun suorituksen ei tarvitse jäädä odottamaan, että tämä saadaan suoritettua vaan sovelluksen muu toiminta voi jatkua normaalisti. Kun prosessi saadaan suoritettua, käsitellään sen syöte takaisinkutsu-funktiossa (callback). Synkronisessa suorituksessa sama tehtävä suoritettaisiin kerralla alusta loppuun ja ohjelman muu suoritus jatkuisi vasta tämän jälkeen. Node.js on erityisestä suunniteltu nopeiden ja kevyiden tehtävien suoritukseen ja esimerkiksi suurta laskentatehoa vaativat tehtävät on syytä ulkoistaa omalle sovellusprosessilleen. (Capan 2013.)

Node.js-sovelluskehyksen vahvuus on sen ohjelmointirajapinta, jota on yksinkertaista käyttää ja kolmannen osapuolen kehittämien moduulien käyttöönotto on helppoa. Node.js:n eduksi voidaan lukea myös itse Javascript-kieli, joka on maailmanlaajuisesti hyvin tunnettu websovelluskehittäjien keskuudessa. (Teixeira 2013, 3–4.) Node.js:lle ominaista on sille kirjoitetun ohjelmakoodin yksinkertaisuus sekä tapahtumavetoisuus ja asynkronisuus. Tapahtumavetoisella ohjelmoinnilla tarkoitetaan tyyliä, jolla ohjelman eri osien suoritus ja suoritusten virta perustuu tapahtumiin. Tällainen tapahtuma voi olla esimerkiksi tietokantaoperaation valmistuminen. Tapahtumien käsittelyä hoitavat tapahtumankäsittelijät tai takaisinkutsu-funktiot, jotka suoritetaan kun yksittäinen tapahtuma laukaistaan. Node.js mahdollistaa siirrännän asynkronisesti, eli sovelluksen suoritus ei keskeydy ja sovellus voi jatkaa muiden funktioiden suoritusta vaikka edellisen funktion suoritus on vielä kesken. (Teixeira 2013, 15–17.)

5.1 JSON

JSON (JavaScript Object Notation) on kevyt, usean eri nykyaikaisen ohjelmointikielen tukema datan esitysmuoto. JSON on tekstimuotoinen: sitä on helppo lukea ja kirjoittaa käsin ja ohjelmat osaavat jäsentää ja tuottaa omia JSON-olioita. (JSON 2015.)

JSON-muotoinen olio kirjoitetaan alku- ja loppuaaltosulkujen sisään, joiden väliin määritellään JSON-olion attribuutit nimi / arvo-pareina, joiden nimet ovat olion sisällä yksilöllisiä ja lainausmerkkien välissä. Arvon on myös oltava lainausmerkkien välissä, jos se on merkkijono. Jos arvo on numero-, boolean-, tai null-arvoinen, se merkitään ilman lainausmerkkejä. Taulukko- tai lista-muotoinen data merkitään hakasulkujen väliin. JSON mahdollistaa myös sisäkkäiset JSON-oliot, jotka merkitään samaan tapaan kuin muutkin olion arvot nimi / arvo-pareihin. Kuvassa 1 on esitelty JSON-merkkijonon syntaksia mahdollisimman monipuolisesti. (JSON 2015.)

```
{
        "merkkijono":"Tämä on merkkijono 1",
        "numero": 100,
        "boolean":true,
        "lista": [
                {
                        "merkkijono":"Tämä on merkkijono 2",
                        "numero": 10.50,
                        "boolean":false,
                },
                {
                        "merkkijono":"Tämä on merkkijono 3",
                        "numero": null,
                        "boolean":true,
                        "lista": [
                                {
                                "merkkijono":"Tämä on merkkijono 4",
                                "numero": 1.50,
                                "boolean":false,
                                },
                        ]
                }
        ]
}
```

Kuva 1. Esimerkki JSON-muotoisesta merkkijonosta.

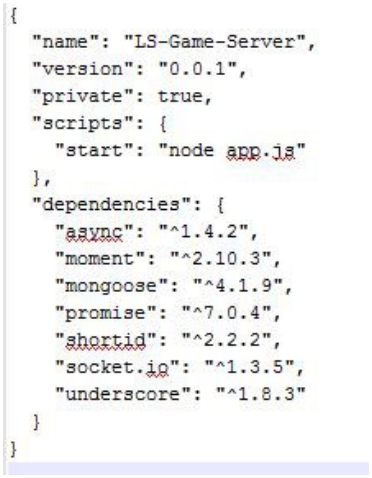
5.2 NPM

NPM (Node Package Manager) on komentorivityökalu, joka tarjoaa verkkopalvelun Node.js-moduulien lataamiselle komentoriviltä, asennettujen moduulien hallinnoinnin – asennus, päivitys ja poisto – omalla työasemalla ja lisäksi se standardisoi Node.js-sovelluksen riippuvuudet muihin moduuleihin ”package”-tiedoston avulla. NPM on ollut mukana Node.js:n asennuspaketissa Node.js:n versiosta 6 lähtien, joten sitä ei tarvitse asentaa erikseen. (Teixeira 2013, 8.)

NPM:n avulla asennetut moduulit voidaan asentaa globaalisti tai paikallisesti. Globaaliasennus tapahtuu komentorivikomennolla ”npm install” sekä antamalla lippu ”-g” ja asennettavan moduulin nimi. Globaalisti asennettaessa moduuli asennetaan Node.js:n asennuksessa määriteltyyn ”node_modules”-nimiseen kansioon, josta käsin moduuli on käytettävissä ”require”-funktiolla työaseman mistä tahansa Node.js-sovelluksesta. Paikallisesti asennettuna, ilman ”-g”-lippua, moduuli asennetaan komentorivin sen hetkiseen kansioon, joka on tyypillisesti Node.js-sovelluksen juurihakemisto. Tällöin kansioon luodaan sovelluksen oma ”node_modules”-kansio, jonne ladatut moduulit asennetaan ja josta sovellus etsii ”require”-funktiolla määriteltyä moduulia (Teixeira 2013, 9–10.)

Moduulien päivitys ja poisto voidaan suorittaa NPM:n avulla komentorivikomennolla. Syöttämällä komentoriville komennon ”npm uninstall” ja antamalla poistettavan moduulin nimen NPM poistaa paikallisesti asennetun moduulin nykyisen kansion ”node_modules”-kansiosta. Moduulin viimeisimmän version päivitys tapahtuu kutsumalla komentoa ”npm update” ja antamalla halutun paketin nimen. Halutun version lataaminen tehdään syöttämällä paketin nimen perään ”@” ja haluttu versionumero, esimerkiksi ”@0.1”. Globaalisti asennetut moduulit poistetaan ja päivitetään antamalla komentoon ”-g”-lippu, esimerkiksi ”npm uninstall -g gulp” poistaa ja ”npm update -g gulp” päivittää globaalisti asennetun ”gulp”-nimisen moduulin. (Teixeira 2013, 11.)

Package-tiedosto määrittelee Node.js-sovelluksen tarvitsemat moduulit ”dependancies”-kohdassa. Tiedosto on JSON-muotoinen tiedosto, joka sisältää tarvittavien moduulien nimet sekä versionumeron, sekä tietoa Node.js-sovelluksesta, kuten sovelluksen kehittäjän sovellukselle antamasta nimestä ja nykyisestä versionumerosta. (Kuva 2) Kutsumalla NPM:n komentorivikomentoa ”npm install” samassa kansiossa, jossa package-tiedosto sijaitsee, asennetaan tiedostossa määritellyt moduulit ”node_modules”-kansioon. (Teixeira 2013, 12–13.)



```
{
  "name": "LS-Game-Server",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "async": "^1.4.2",
    "moment": "^2.10.3",
    "mongoose": "^4.1.9",
    "promise": "^7.0.4",
    "shortid": "^2.2.2",
    "socket.io": "^1.3.5",
    "underscore": "^1.8.3"
  }
}
```

Kuva 2. Esimerkki package -tiedoston sisällöstä.

5.3 Tapahtumankäsittely

Tapahtuma on tilanne, jonka aiheuttaa sovelluksen osa sovelluksen suorituksen aikana ja jolta saatetaan edellyttää reagointia toiselta sovelluksen osilta. Tapahtuman synnyttävää osapuolta sanotaan lähteeksi ja siihen reagoivaa osapuolta tarkkailijaksi. Lähteen ja tarkkailijan välinen kommunikointi voi tapahtua myös pelkästään tapahtumien kautta. Tarkkailijat rekisteröivät lähteelle, joka laukaisee rekisteröidyn takaisinkutsu-funktion tapahtuman sattuessa. Funktiolle annetaan tapahtumaan liittyvät tiedot parametreina ja tarkkailija pääsee niihin käsiksi takaisinkutsussa. (Koskimies & Mikkonen 2005, 85–86.)

EventEmitter on Node.js-olio, joka mahdollistaa tapahtumapohjaisten kommunikoinnin Node.js-sovelluksen olioiden välillä. Ohjelman oliot voivat tarkkailla toisen tapahtumia ja laukaista oman takaisinkutsu-funktionsa, kun tapahtuma laukaistaan. Tätä prosessia kutsutaan tapahtumankuunteluksi. Tapahtumankuuntelu voi olla kertaluontoista tai jatkuvaa ja yhdellä tapahtumalla voi olla useita yhtäaikaisia tapahtumankuuntelijoita. (Takada 2015.)

Tapahtumankuuntelija rekisteröidään kuuntelemaan tapahtuman laukaisua ”on”- tai ”once”-metodilla määritellyllä tapahtuman nimellä ja takaisinkutsufunktiolla, joka ottaa parametreikseen laukaisun yhteydessä annetut parametrit, jos niitä on määritelty. Tapahtuman laukaisu tapahtuu kutsumalla ”emit”-metodia ja antamalla sille tapahtuman nimen ja takaisinkutsu-funktioon parametreiksi annettavat arvot. Metodissa määritellään tapahtuman yksilöivä nimi ja mahdolliset välitettävät muuttujat parametreina, joita tapahtumankäsittelijät voivat käyttää. (Takada 2015.)

Tapahtumia valvoo Node.js-sovelluksen osa nimeltä tapahtumasilmukka. Se valvoo kaikkia sovelluksen tapahtumia ja suorittaa tapahtumankäsittelijöille määrätyt suoritettavat takaisinkutsu-funktiot synkronisesti vuorotellen: kun yhden tapahtuman takaisinkutsu-funktio saadaan suoritettua, siirrytään käsittelemään seuraava. (Teixeira 2013, 16–17.)

5.4 Buffer ja Stream

Buffer-luokka mahdollistaa binäärimuotoisen datan käsittelyn Node.js-sovelluksessa. Buffer-olio on kuin taulukko, joka koostuu tavuista, mutta taulukon kokoa ei voida sen luonnin jälkeen muuttaa. Merkkijonoja käsiteltäessä yksittäinen tavu on kokonaislukuarvo 0-255 väliltä. Kokonaisluvun merkitys riippuu Buffer-olion koodauksesta, joka on joko ascii, utf-8, utf-16, hex tai base64. (Takada 2015.)

Olioon voidaan kirjoittaa esimerkiksi merkkimuotoista dataa ”write”-metodilla ja se saadaan käännettyä jälleen merkkijonoksi ”toString”-metodilla. Binäärimuotoista dataa tarvitaan esimerkiksi kirjoittaessa tiedostoon tai lukiessa siitä ja Stream-olioiden lukemiseen. (Takada 2015.)

Stream-oliot mahdollistavat datan käsittelyn palasista koostuvana virtana, eli sitä mukaa kun yksittäinen palanen on saatavilla. Stream-oliot voivat olla kirjoittavia, lukevia tai molempia yhtä aikaa. Kirjoittava Stream-olio ottaa vastaan käyttäjän kirjoittamia syötteitä ”write”-metodin avulla ja kirjoitus päätetään ”end”-metodilla. Lukeva Stream-olio luo tapahtuman, kun yksittäinen palanen datavirrasta on saatu luettua. Tapahtuma on ”data”-niminen ja se palauttaa tapahtumankäsittelijälle luetun palasen muun ohjelman käytettäväksi. ”End”-tapahtuma laukaistaan, kun koko datavirta on saatu luettua loppuun. (Takada 2015.)

5.5 Net

Net-moduuli mahdollistaa TCP-palvelinten ja -yhteyksien luonnin Node.js-sovelluksessa. Palvelin luodaan ”CreateServer”-metodilla ja asetetaan kuuntelemaan valittua porttinumeroa listen-metodilla. ”CreateServer”-metodi palauttaa server-olion, joka laukaisee ”connection”-tapahtuman aina kun uusi TCP-yhteys on luotu palvelimelle. ”Connection”-tapahtuman tapahtumankäsittelijä palauttaa socket-olion, joka on käytännössä kaksisuuntainen Stream-olio, jonka syötettä voidaan lukea ja siihen voidaan

kirjoittaa kuten mikä tahansa kirjoittava ja lukeva Stream-oliota. Socket-olio on keskeisessä roolissa TCP-palvelimen ja asiakkaan välisessä tiedonsiirrossa. (Teixeira 2013, 85–86.)

Osapuolen sulkiessa yhteyden, toinen osapuoli näkee tämän ”end”-tapahtuman laukaisuna. Tavallisesti TCP-yhteyttä pidetään yllä osapuolten välillä niin kauan kunnes socket-olion ”end”-metodia kutsutaan tai yhteys katkeaa esimerkiksi verkkovirheen vuoksi. Yhteydelle voidaan asettaa aikakatkaisu setTimeout-metodilla, joka sulkee yhteyden määrätyn ajan kuluttua, jos socket-olion ja palvelimen välillä ei ole liikennettä tänä aikana. Yhteyttä voidaan pitää yllä myös ”keep-alive”-mekanismin avulla, jossa lähetetään säännöllisin väliajoin ACK-liputettu segmentti, johon jäädään odottamaan vahvistusta. Kun tämä vastaanotetaan, lähetetään vahvistusviesti. Näin osapuolilta estetään yhteyden aikakatkaisu, kun liikennettä ei ole määritetyn ajan sisällä ollut, mutta sitä odotetaan tulevaisuudessa. (Teixeira 2013, 86–87.)

Socket-olion ”write”-metodilla välitettyjä viestejä palvelimelle tai asiakkaalle ei välttämättä lähetetä heti. Kutsumalla socket-olion ”SetNoDelay”-metodia arvolla ”true” TCP-protokolla pakotetaan lähettämään data heti, kun se on kirjoitettu socket-olioon. Oletusehtoisesti socket-olioon kirjoitettu data puskuroidaan ennen lähetystä, jolloin TCP saattaa jäädä odottamaan lähetyspuskurin täyttymistä ennen datan varsinaista lähetystä. Tämä saattaa kasvattaa viivettä lyhyiden yksittäisten viestien saapumiselle, sillä useita eri aikaan kirjoitettuja toisistaan riippumattomia sanomia saatetaan niputtaa yhteen ja lähettää vasta kun lähetyspuskurin koko sopii TCP:lle. (Teixeira 2013, 87–88.)

5.6 Dgram

Dgram-moduulin avulla voidaan vastaanottaa ja lähettää UDP-protokollan mukaisia datasähkeitä. UDP-socketin luonti tapahtuu ”createSocket”-metodilla, joka palauttaa socket-olion. Node.js asetetaan kuuntelemaan haluttua porttia bind-metodilla, muutoin Node.js käyttää satunnaista porttia. Dgram-moduulissa ei itsessään ole erillistä metodia palvelimen tai asiakassovelluksen luonnille,

vaan socket-olion luonti on roolista riippumatta samanlainen. (Teixeira 2013, 130.)

Datasähkeiden vastaanotto tapahtuu kuuntelemalla socket-olion ”message”-tapahtumia. Tapahtuman takaisinkutsu-funktio välittää parametrinaan vastaanotetun viestin buffer-oliona sekä JSON-olion, joka sisältää lähettävän osapuolen IP-osoitteen, sovelluksen porttinumeron ja lähetettyjen tavujen lukumäärän. (Teixeira 2013, 130–132.)

Tietosähkeen lähetys tehdään kutsumalla socket-olion metodia ”send” ja antamalla parametreiksi buffer-olioksi muutettu merkkijono, viestin alkamis- ja lopetusmerkin sijainnin puskurioliossa sekä vastaanottajan porttinumeron ja osoitteen. Viimeisenä parametrina voidaan antaa takaisinkutsufunktio, joka suoritetaan, kun datasähke on lähetetty. Takaisinkutsu-funktion parametreina saadaan mahdollinen virheilmoitus ja lähetetyn tietosähkeen tavumäärä. Socket-olio suljetaan kutsumalla ”close”-metodia. (Teixeira 2013, 134.)

Dgram mahdollistaa socket-olion liittymisen multicast-ryhmään. Liittyminen tapahtuu kutsumalla socket-olion metodia ”addMembership” ja antamalla parametriksi multicast-osoitteen. Lähettämällä viestin tähän osoitteeseen lähettää saman viestin tällöin kaikille ryhmään liittyneille Node.js-sovelluksille. (Teixeira, 2013, 136–138.)

5.7 Socket.io

Socket.io on kolmannen osapuolen tarjoama Node.js-kirjasto, joka mahdollistaa reaaliaikaisen kommunikoinnin Node.js-sovelluksen sekä websovelluksen välillä Websocketien avulla. Websocket-palvelin luodaan kutsumalla listen-metodia ja antamalla sille halutun porttinumeron. Metodi palauttaa Server-olion, joka laukaisee ”connection”-tapahtuman, kun asiakassovellus ottaa luotuun palvelimeen yhteyden. ”Connection”-tapahtuma välittää takaisinkutsu-funktiossaan ohjelman käyttöön socket-olion, jonka tapahtumia kuuntelemalla vastaanotetaan toisen osapuolen lähettämiä viestejä. Socket-olio toimii samanlaisten metodien avulla kuin EventEmitter-olio. Kutsumalla olion ”emit”-

metodia ja antamalla sille parametrina tapahtuman nimi ja lähetettävä arvo, toinen samaa tapahtumaa kuunteleva osapuoli vastaanottaa viestin tapahtumankuuntelijan takaisinkutsu-funktiossa (Teixeira 2013, 243–244.)

Socket.io tarjoaa samanlaisen ohjelmointirajapinnan palvelimeen sekä websovellukseen, eli viestin lähetys ja vastaanotto palvelimelle tai asiakassovellukselle tapahtuu käyttämällä samoja metodeja ja kuuntelemalla Socket-olion tapahtumia. Lisäksi broadcast-metodin avulla voidaan lähettää sama viesti monelle käyttäjälle yhdellä kutsulla. (Teixeira 2013, 244.)

5.8 Cluster

Cluster-moduuli mahdollistaa yksittäisen Node.js-prosessin jakamisen useaksi prosessiksi. Sovelluksen klusterointi tapahtuu metodilla ”fork”, jolloin sovellus aloittaa uuden worker-prosessin. ”Fork”-metodia kutsuvan sovelluksen nimi on master. Worker-prosessin olio tallentuu cluster-moduulin workers-olion ja olio tarjoaa mahdollisuuden mm. worker-prosessin sammuttamiselle tai uudelleen käynnistymisestä master-prosessista käsin. (Node.js 2015.)

Klusterointi mahdollistaa käytännössä yksittäisten tehtävien jakamisen usean prosessin suoritettavaksi yhden prosessin sijaan. Tällainen tehtävä voi olla esimerkiksi tuleva HTTP-pyyntö, johon sovelluksen on vastattava tavalla tai toisella. Worker-prosessien kuormanjako-algoritmina on ”round-robin”, eli tehtävien jako worker-prosessille vuorotellen. Toisessa vaihtoehdossa tehtävät jaetaan ”kiinnostuneiden” worker-prosessien kesken. Jälkimmäisen vaihtoehdon väitetään ainakin teoriassa varmistavan parhaan suorituskyvyn, mutta todellisuudessa suoritettavien tehtävien jako worker-prosessien kesken on jakaantunut suurimmaksi osaksi kahdelle worker-prosessille kahdeksasta mahdollisesta. (Node.js 2015.)

Master- ja worker-prosessien välinen viestintä onnistuu asettamalla master-prosessi kuuntelemaan luodun workerin tapahtumia. Workerilta masterille välittyy käytännössä kaikki worker-prosessin I/O-viestit. Lisäksi voidaan kuunnella worker-prosessin käynnistyminen ja kaatumisesta. (Node.js 2015.)

Koska ”fork”-metodi luo käytännössä uuden Node.js-sovellusprosessin, worker- ja master-prosessien välillä ei ole yhteistä muistia, vaan jokainen sovellus tallentaa mm. oliot ja muuttujat omaan prosessimuistiinsa. Tämä tuo haasteita sovelluksille, joissa tiedon ajantasaisuus on avainasemassa, koska tallennetun tiedon on pysyttävä samanlaisena kaikissa worker-prosesseissa. (Node.js 2015.)

6 Palvelinsovelluksen toteutus

Toteutetussa opinnäytetyössä tavoitteena oli suunnitella ja toteuttaa Node.js-sovelluskehyksellä toteutettu reaaliaikainen palvelin. Työn tuloksena syntyi palvelinsovellus ja sen suorituskykyä testaava benchmarking-sovellus, joka luo annettujen parametrin mukaisen määrän asiakassovelluksia kommunikoimaan palvelinsovelluksen kanssa. Asiakassovelluksiin on mahdollista parametrisoida myös lähetettävien viestien lukumäärä ja niiden lähetystiheys. Samanaikaisesti sovellukset keräävät tilastoja lähetetyistä ja vastaanotetuista viesteistä ja mm. viestien lähetyksen ja vastaanoton vasteajasta, latenssista. Palvelinsovelluksen ja benchmarking-sovelluksen julkaisuun käytettiin GitHub-palvelua, jossa molemmat sovellukset ovat tarkasteltavissa ja ladattavissa lähdekoodeineen (Sormunen 2015a, Sormunen 2015b).

6.1 Palvelinsovelluksen arkkitehtuuri

Toteutettu palvelinsovellus perustuu asiakas–palvelin-malliin, jossa asiakassovellusten välinen kommunikointi tapahtuu aina palvelimen kautta, eikä suoraan asiakassovellukselta toiselle. Asiakassovellukset lähettävät viestejä liikkeistään palvelimelle, palvelin ottaa ne vastaan, laskee mitä liikkeestä seuraa ja välittää vastauksen takaisin käyttäjälle. Palvelin päättää ensisijaisesti mitä tietoa asiakassovellukset saavat toisistaan tietää.

Palvelinsovelluksen arkkitehtuurityyli on kolmesta tasosta koostuva kerrosarkkitehtuuri. Ylin kerros on tiedonsiirtokerros, jonka tehtäviä ovat tulevien ja lähtevien viestin vastaanotto ja lähetys. Sen jälkeen tulee sovelluslogiikkakerros, joka toteuttaa tiedonsiirtokerroksen rajapinnat tarjoamista palveluista ja joka vastaa palvelimen sovelluslogiikasta ja käsitteiden yhteydestä toisiinsa. Alimpana kerroksena on datakerros, joka tarjoaa rajapinnat sovelluslogiikkakerroksen tarvitsemille tiedon talletus-, poisto-, päivitys- ja hakuoperaatioille.

6.1.1 Abstraktiotasot

Datakerroksen tehtävän on ensisijaisesti huolehtia tiedon talletuksesta. Se tarjoaa sovelluslogiikkakerrokselle tarvittavat palvelut tallennetun tiedon hakemiselle, päivittämiselle, tallentamiselle ja poistamiselle. Datakerros on alin kerros ja se ei sen vuoksi ole riippuvainen muista kerroksista.

Sovelluslogiikkakerros käyttää datakerroksen tarjoamia tiedon talletus-palveluita oman toimintansa tukena. Se ottaa vastaan tiedonsiirtokerrokselta tulevat viestit ja päättelee mitä viestistä seuraa ja muodostaa tilanteeseen sopivan viestin vastauksena käyttäjälle. Vastauksen lähetys tapahtuu kutsumalla tiedonsiirtokerroksen lähetys-palvelua. Olion tehtävänä on myös identifioida asiakassovelluksilta tulevat viestit luotuun käyttäjään ja tallentaa käyttäjien tiedot datakerroksessa.

Tiedonsiirtokerroksen tehtävinä on palvelimen ja asiakassovellusten välillä kulkevien viestien vastaanotto ja lähetys. Kerros pitää yllä rajapintaa asiakassovelluksiin ja se määrittelee sovelluksen porttinumeron, johon viestit tulee lähettää. Se huolehtii viestin lähetyksessä haluttuun osoitteeseen vastaanottajan IP ja porttinumero -yhdistelmän avulla. Tiedonsiirtokerros vastaa myös lähetettävän tiedon koodaamisesta sen käyttämän tiedonsiirtoprotokollan tukemassa muodossa. Vastaanotetut viestit muunnetaan muiden sovelluksen kerrosten lukemaksi muodoksi.

6.1.2 Tekninen toteutus

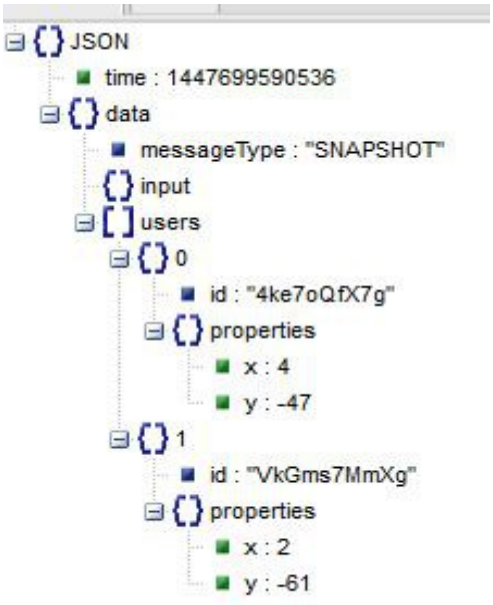
Sovelluksen datakerroksessa toimiva UserService-olio tallentaa kaikki sen ylläpitämät käyttäjäkohtaiset tiedot käyttäjäolioista koostuvaan taulukkomuuttujaan. Olion kutsuttavilla metodeilla saadaan haettua kaikki tallennetut käyttäjäoliot kerralla tai sitten hakemalla käyttäjän tilan tai tiedonsiirtokerroksen käyttämän Socket-oliolle annetun ID-tunnisteen avulla. Käyttäjä lisätään kutsumalla lisäysmetodia ja antamalla parametriksi tiedonsiirtokerrokselta saatu Socket-olion ID, jolloin uusi käyttäjäolio luodaan ja tallennetaan taulukkoon. Käyttäjäolion poisto tapahtuu kutsumalla poistometodia ja antamalla viittaus poistettavaan käyttäjäolioon tai käyttäjäoliolle annetun ID-tunnisteen perusteella.

Sovelluslogiikkakerroksessa toimiva Server-olio käyttää UserService-olion datan hallinta -palveluita luotujen käyttäjäolioiden hallintaan tiedonsiirtokerrokselta tulevien viestien perusteella. Viestit vastaanotetaan kuuntelemalla valitun tiedonsiirtokerroksen palvelun ”message”-tapahtumia ja käsittelemällä viestit määrätyillä olion metodeilla tapahtuman takaisinkutsu-funktiossa. JSON-muotoisista viesteistä luetaan attribuutti ”messageType”, joka määrittää viestin tyypin, joka määrittää millaisia toimia sovellukselta vastaanotettuun viestiin vaaditaan. Viestin tyypin oletetaan olevan joko ”Connect”, ”Verify” tai ”State” ja asiakassovelluksen odotetaan asettavan viestin tyypiksi jokin edellä mainituista.

Tiedonsiirtokerrokselta vastaanotettuihin viesteihin reagoidaan tyypillisesti tunnistamalla viestin lähettäjä ”socketInformation”-muuttujan arvon avulla, hakemalla käyttäjän tiedot UserService-oliosta, päivittämällä olion tiedot ja tai suorittamalla viestissä oleva komento, luomalla vastaus viestiin ja lähettämällä se takaisin asiakassovellukselle. Esimerkiksi ”state”-tyyppisiin viesteihin (kuva 3) reagoidaan ottamalla vastaan asiakassovelluksen viesti, lukemalla viestistä komento, esimerkiksi ”MOVE_UP” ja muuttamalla käyttäjän User-olion y-koordinaatin arvoa korottamalla sitä yhdellä. Tämän jälkeen asiakassovellukselle välitetään User-olion muuttuneet X- ja Y-arvot viestissä.

Kuva 3. Ote asiakassovellusten palvelimelle lähettämistä ”state”-viesteistä.

Kaikkien asiakassovellusten pitäminen synkronissa palvelimen User-oliohin liittyvistä sijaintitiedoista tapahtuu yksinkertaisen tapahtumasilmukan sisällä. Tapahtumasilmukka suoritetaan palvelimen parametreissa annetun ajan välein, esimerkiksi 20 kertaa sekunnin aikana. Yhden tapahtumasilmukan kierroksen aikana välitetään ”snapshot”-tyyppinen viesti kaikille palvelimeen yhdistyneille asiakassovelluksille. Viestiin kerätään kaikkien User-olioiden ID-tunnisteet ja sen hetkinen sijainti koordinaatistossa. Kuvassa 4 on esitetty asiakassovelluksen vastaanottama JSON-muotoinen viesti, jossa on kahden muun asiakassovelluksen koordinaatit.



Kuva 4. ”Snapshot”-tyyppisen viestin rakenne asiakassovelluksessa.

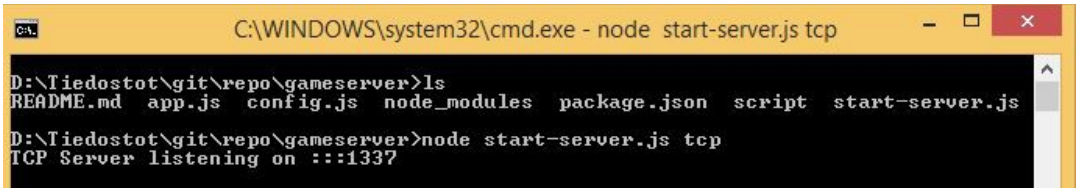
Tiedonsiirtokerroksessa toimivat TCPService-, UDPService- ja WSService-oliot huolehtivat datan vastaanottamisesta ja lähettämisestä verkon yli olion

käyttämän tiedonsiirtoprotokollan avulla. Vastaanotetut viestit välitetään laukaisemalla käytössä olevan olion ”message”-tapahtuma. Kaikissa kolmessa oliossa viestin vastaanotto noudattaa samaa kaavaa: viesti vastaanotetaan jonkin muotoisena, lähettäjä rekisteröidään, viesti kuitataan lähettäjälle, viesti käsitellään sekä muutetaan muun sovelluksen lukemaan muotoon ja välitetään ”message”-tapahtumassa. Viestin kuittauksessa lähettävälle asiakassovellukselle lähetetään palvelimen vastaanottama viesti takaisin.

Viestien lähettäminen tapahtuu kutsumalla olion ”send”-metodia ja antamalla parametrina lähetettävä viesti. Viesti kapseloidaan toisen JSON-olion data-attribuuttiin, jossa ilmoitetaan viestin lähetysaika ”time”-attribuutissa. Viesti lähetetään Service-olion käyttämällä protokollalla ja viestin lähetys kuitataan laukaisemalla tapahtuma ”send”, jossa takaisinkutsu-funktioille välitetään Service-olion lähettämä kapseloitu viesti.

6.1.3 Palvelinsovelluksen suorittaminen

Palvelinsovelluksen ”config.js”-tiedostossa voidaan määritellä palvelimen osoite, porttinumero. Palvelinsovellus käynnistetään ajamalla komentorivikomento ”node start-server.js” palvelinsovelluksen hakemistossa. Antamalla komentoon parametrin ”tcp” (kuva 5), käytetään palvelimen tiedonsiirtoprotokollana TCP-protokollaa, valitsemalla ”udp” käytetään UDP-protokollaa ja ”ws” parametrilla protokollana on Websocket. Palvelin käynnistyy ”config.js”-tiedostossa määrättyyn porttinumeroon.



Kuva 5. Palvelinsovelluksen käynnistäminen komentoriviltä.

6.2 Benchmarking-sovellus

6.2.1 Tekninen toteutus

Benchmarking-sovelluksen käynnistämien asiakassovellusprosessien arkkitehtuuri on hyvin samankaltainen kuin palvelinsovelluksen. Ainoana erona on, että asiakassovellusprosessit kommunikoivat ainoastaan palvelimen kanssa. Asiakassovelluksille on valittavana kolmesta tiedonsiirtoprotokollasta yksi käytettäväksi yhden suorituskykymittauksen aikana. Suorituksen aikana lähetetyistä ja vastaanotetuista viesteistä kerätään suorituksen aikana tietoa ja ne esitetään graafisesti erillisessä websovelluksessa.

Yhteyden laatua arvioidaan mittaamalla latenssia ja pakettien häviämistä. Latenssia mitataan liittämällä lähetettävään viestiin mukaan lähetyksen aikaleima ja odottamalla viestin kuittausviestiä palvelimelta, jossa on ilmoitettu sama aikaleima. Kun kuittausviesti saapuu, latenssi lasketaan vähentämällä kuittausviestin saapumishetken aikaleimasta alkuperäisen viestin aikaleima. Pakettien häviämistä voidaan mitata laskemalla jokainen lähetetty ja vastaanotettu paketti ja vertaamalla näitä arvoja odotettuihin, parametreina annettuihin lähetettäviin pakettilukumääriin.

6.2.2 Toiminta

Samanaikaisesti suoritettavien asiakassovellusten lukumäärää voidaan rajoittaa erillisellä parametrilla. Jos käynnistettävien asiakassovellusten lukumäärä on 50 ja yhtäaikaisten 25, suoritetaan ensin ensimmäiset 25 rinnakkain. Kun yksittäinen sovellus ensimmäisestä 25 sovelluksesta lopettaa suorituksensa, aloittaa seuraavana jälkimmäisestä 25 sovelluksesta vuorossa ensimmäisenä oleva suorituksen heti tämän jälkeen. Sovelluksen suoritus loppuu, kun kaikki 50 asiakassovellusta ovat saaneet suorituksensa päätökseen.

Benchmarking-sovellukselle annetut parametrit syötetään ”Config.js”-tiedostoon (kuva 6). ”Host”- ja ”port”-kohdat ovat sovelluspalvelimen IP-osoite ja

porttinumero ja ”protocol”-kenttä kuvaa käytettävää tiedonsiirtoprotokollaa. ”Children”-kenttä kertoo kuinka monta asiakassovellusprosessia käynnistetään suorituskyvyn testausta varten. ”Repeat count”-kenttä kertoo kuinka monta viestiä asiakassovelluksella on tavoitteena yhteensä lähettää. ”Repeat interval” ja ”Update per interval” ilmaisevat sekunteina millaisissa sykleissä viestejä lähetetään. ”Concurrent”-kenttä rajaa sen, montako yhtäaikaista suorittavaa asiakassovellusprosessia on käynnissä. ”Duration”-kenttä asettaa asiakassovelluksen suorituskykymittaukselle aikarajan sekunneissa. Jos arvo on -1, aikarajaa ei ole).

```
module.exports = {
    host: '127.0.0.1',
    port: 1337,
    protocol: 'tcp', // protocol used
    benchmark: {
        children:2, // amount of peers
        repeatCount: 500, // messages to be sent. if duration is assigned
        repeatInterval: 1, // in seconds
        updatePerInterval: 60, // repeats per repeatInterval (ie. 1 / 60)
        concurrent: 25, // amount of concurrent peers
        duration: -1, // in seconds, -1 = no duration
    }
}
```

Kuva 6. Benchmarking-sovelluksen parametrit ”Config.js”-tiedostossa

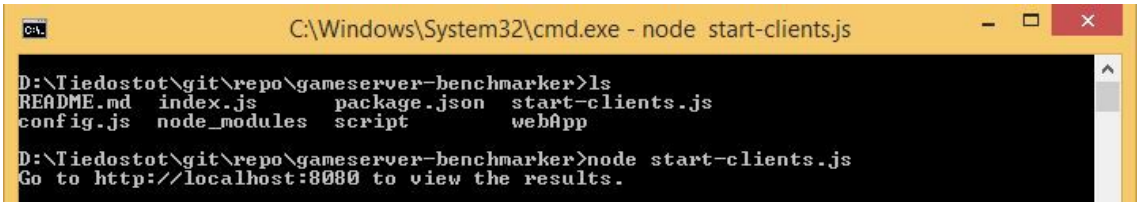
Yksittäisen asiakassovelluksen elinkaareen kuuluu tilaviestien lähetys ja tulosten esitys. Tilaviestit simuloivat asiakassovelluksen näppäimistön syötettä. Palvelin vastaa jokaiseen lähetettyyn tilaviestiin lähettämällä asiakassovelluksen sijainnin koordinaatistossa. Esimerkiksi jos asiakas lähettää viestin, jossa kerrotaan asiakkaan liikkuneen ylöspäin, palvelin laskee User-olion x-koordinaatille uuden arvon lisäämällä nykyiseen x-koordinaattiin luvun 1. Vastauksena palvelin lähettää käyttäjän lasketut x- ja y-koordinaattien arvot (kuva 7).

Kuva 7. Palvelimen vastausviesti asiakassovelluksen ”state”-viestiin.

Samanaikaisesti muut asiakassovellukset ottavat vastaan palvelimelta tulevia ”snapshot”-viestejä, joissa ilmoitetaan kaikkien muiden asiakassovellusten tilat.

6.2.3 Suorituskykymittaus

Palvelimen suorituskykyä testaava Node.js-sovellus käynnistetään komentorivikomennolla ”node start-clients.js” (kuva 8) Palvelinsovelluksesta poiketen testaava sovellus lukee valitun tiedonsiirtoprotokollan ”config.js”-tiedostosta. Valitun tiedonsiirtoprotokollan tulee olla sama kuin käynnissä olevan palvelimen sen hetkinen käytössä oleva tiedonsiirtoprotokolla.



Kuva 8. Benchmarking-sovelluksen käynnistäminen komentoriviltä.

Suorituskykymittaukset esitetään websovelluksessa menemällä selaimella osoitteeseen ”http://localhost:8080”. Sovelluksen keräämä data välitetään websovelluksen käyttöön Socket.io-moduulin välityksellä.

Asiakassovellukset lähettää ja vastaanottaa viestejä sovelluksen ”config.js”-tiedostossa määrättyyn palvelimen IP-osoite ja porttinumero-yhdistelmään. Benchmarking-sovellukselle annetut parametrit kerrataan websovelluksen ylälaidasta löytyvästä ”Configurations”-kohdasta (kuva 9).

Kuva 9. Parametrien esitys websovelluksessa.

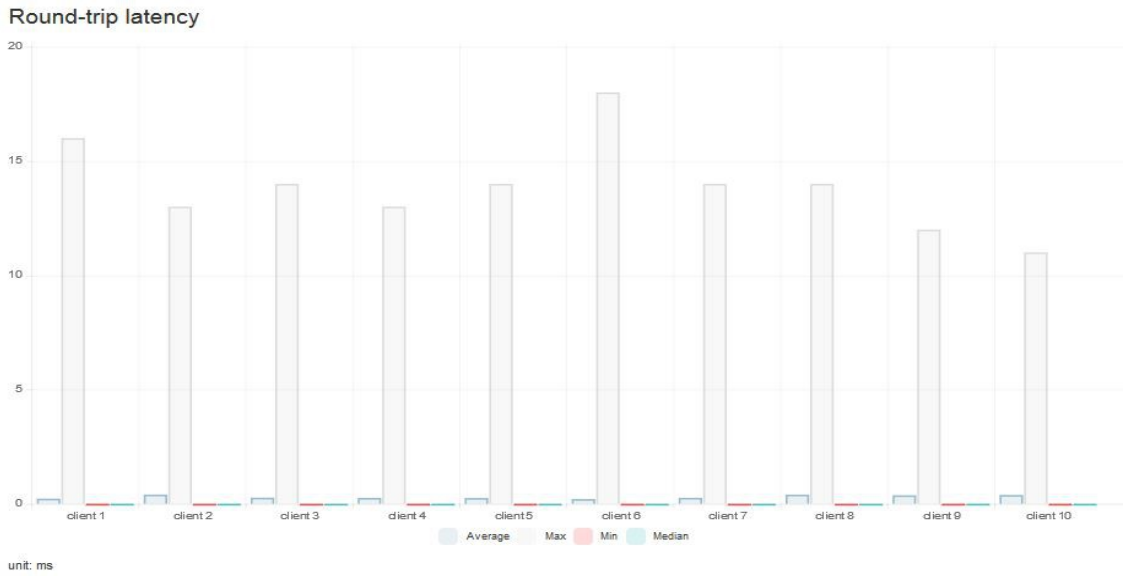
Suorituskykymittauksen etenemisestä ilmoitetaan reaaliaikaisesti websovelluksen ylälaidassa sijaitsevassa ”status”-laatikossa. Kun mittaus on käynnissä, ilmoitetaan siitä kuvan 10 laatikossa tekstillä ”status: running benchmark tests”.

Kuva 10. Suorituskykymittauksen suorituksesta ilmoittava status-ilmoitus.

Kun suorituskykymittaus on saatu päätökseen, valmistelee suorituskyky-sovellus websovelluksessa esitettävän datan. Websovellus ilmoittaa tästä vaiheesta kuvassa 11 näkyvässä laatikossa tekstillä ”status: waiting for results”.

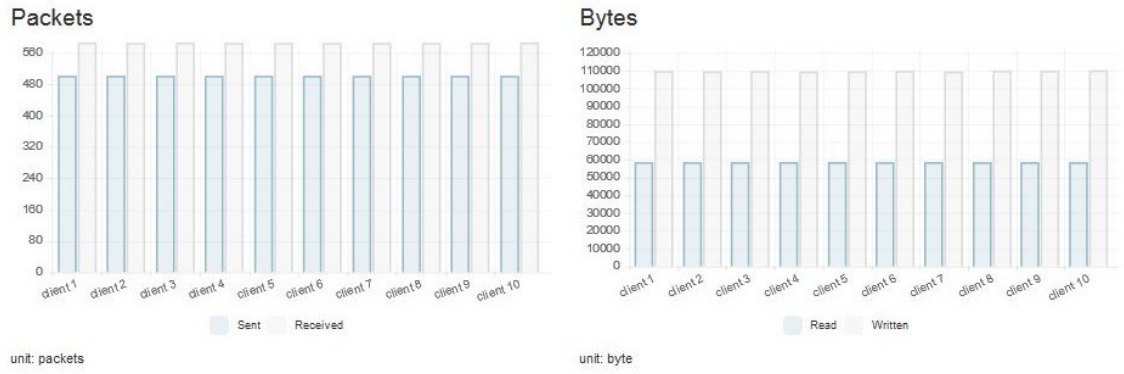
Kuva 11. Kertyneen datan valmistelusta ilmoittava status-ilmoitus.

Kun data on valmis esitettäväksi, tuodaan se näkyviin websovelluksen. Suorituskykymittauksen päätyttyä tulokset esitetään graafisessa muodossa websovelluksessa. Status-ilmoitus muuttuu vihreäksi ja tekstissä ilmoitetaan ”status: ready”. Kuvassa 12 on websovelluksessa ensimmäisenä esitettävä kaaviokuva ”Round-trip latency”, joka esittää viestien välityksen kokonaisviiveen keski-, minimi- ja maksimi- sekä mediaaniarvoa. Kaavion alapuolella ilmoitetaan pylväissä käytetty yksikkö, joka on millisekunti (ms). Kaavion alapuolella on selitteet pylväiden väreille.



Kuva 12. ”Round-trip latency”-kaavion esitys

Kuvassa 13 on esitetty websovelluksen ”Packets”- ja ”Bytes”-kaaviot. ”Packets”-kaavio kuvaa sovelluksen lähetettyjen ja vastaanotettujen pakettien lukumäärää ja ”Bytes”-kaavio kuvaa viesteistä kirjoitettua ja luettua tavumäärää sovelluksittain. Kaavioiden alapuolella on selitteet kaavioissa esitetyille palkeille ja kaaviossa esitetyn datan yksikkö on ”Packets”-kaaviossa lähetetty kokonainen viesti eli paketti ja ”Bytes”-kaaviossa yksikkö on tavu.



Kuva 13. Lähetettyjen ja vastaanotettujen pakettien luku- ja tavumäärä websovelluksessa esitettynä.

Kuvassa 14 esitetään websovelluksen alalaidassa benchmarking-sovelluksen keräämä data taulukkomuotoisena. Taulukon sarakkeiden arvoja on mahdollista järjestää nousevassa tai laskevassa järjestyksessä klikkaamalla sarakkeen nimeä.

Name	Latency avg	Latency max	Latency min	Latency median	Packets sent	Packets received	Bytes written	Bytes read
client 1	0.5876494023904383	22	0	0	502	587	58683	110033
client 2	0.7888446215139442	18	0	1	502	586	58679	109807
client 3	0.9243027888446215	18	0	1	502	587	58689	109908
client 4	0.6434262948207171	17	0	0	502	586	58690	109662
client 5	0.5239043824701195	15	0	0	502	586	58699	109810
client 6	0.5298804780876494	16	0	0	502	586	58671	110126
client 7	0.5617529880478087	19	0	0	502	586	58700	109570
client 8	0.8545816733067729	15	0	1	502	586	58685	110223
client 9	0.5278884462151394	16	0	0	502	586	58694	110202
client 10	0.4940239043824701	11	0	0	502	587	58688	110317

Kuva 14. Ruutukaappaus benchmarking-sovelluksen datan esityksestä taulukkomuodossa websovelluksessa.

7 Pohdinta

Opinnäytetyön toimeksiannossa tavoitteena oli suunnitella palvelinarkkitehtuuri sekä toteuttaa Node.js-palvelinsovellus sen pohjalta. Lisäksi tarvittiin erillinen sovellus, joka ottaa palvelimeen yhteyden, lähettää palvelimen tunnistamia viestejä ja vastaanottaa palvelimelta tulevia viestejä samalla keräten dataa ja mitaten palvelimen suorituskykyä sekä esittää saadut tulokset graafisessa muodossa websovelluksessa.

Oma aikaisempi kokemukseni Javascript-kielestä, Node.js:stä ja npm-sovelluksesta mahdollisti mielestäni sujuvan tiedon soveltamisen käytäntöön. Esimerkiksi ymmärrys Node.js:n tapahtumapohjaisuuden ja asynkronisuuden vaikutuksesta sovelluksen ohjelmakoodissa auttoivat opinnäytetyön toimeksiannossa. Vaikka Node.js-sovelluskehityksessä olen aikaisemmin keskittynyt pääosin websovellusten toteutukseen, oli tästä kokemuksesta mielestäni paljon hyötyä opinnäytetyöprosessin aikana.

Minulla ei ollut aikaisempaa kokemusta opinnäytetyössä käytetyistä tiedonsiirtoprotokollista, kuten UDP- ja TCP-tiedonsiirtoprotokollista, joten tämä

toi omat haasteensa uusien asioiden ymmärtämisessä ja palvelinsovelluksen tiedonsiirron toteutuksessa. Uusien asioiden opiskelu toi kuitenkin näkemystä niiden ominaisuuksista ja miten niitä voidaan käyttää, esimerkiksi edellä mainitut tiedonsiirtoprotokollat.

7.1 Javascriptin tulevaisuuden näkymät

Opinnäytetyö on teknisestä näkökulmasta mielestäni hyvin relevantti. Javascriptiä näkee nykyisin käytettävän hyvin monessa yhteydessä, kun sitä aikaisemmin saattoi nähdä ainoastaan apuna selainpohjaisten websovellusten käyttöliittymien muuttamisessa interaktiivisiksi. Javascript on tästä huolimatta kuitenkin asiakaspään sovelluksissa luonteva kieli, sillä se on ainoa laatuaan. Sen opiskelu ja käyttö on myös helppoa, sillä se ei vaadi selaimen ja tekstinkäsittelyohjelman lisäksi muita ohjelmia. (Fankhauser 2012.)

Esimerkiksi Node.js-sovelluskehys mahdollistaa käytännössä koko websovelluksen rakentaminen puhtaasti Javascript-kielellä. Relaatiotietokantojen sijaan voidaan käyttää MongoDB:n kaltaisia tekstitietokantoja ja sovellusten yhteys palvelimeen voidaan suorittaa AJAX-kutsujen avulla. Kokonaisen sovelluksen arkkitehtuurin rakentaminen yhdellä kielellä helpottaa esimerkiksi kirjoitetun koodin uudelleenkäytettävyyttä: Node.js-sovellukseen kirjoitettua ohjelmakoodia voidaan käyttää joissain tilanteissa myös asiakaspään ohjelmakoodissa muuttamatta sellaisenaan. (Fankhauser 2012.)

Javascriptin ajankohtaisuudesta kielii myös kesällä 2015 julkaistu Javascript-standardin päivitys ECMAScript 6:en. Versio 6 on kokoluokaltaan ensimmäinen suuri päivitys Javascriptiin yli 15 vuoteen. ECMAScript 6 muuttaa Javascriptia suuntaan, jossa monimutkaisten sovellusten kehitys ja hallinta on virtaviivaisempaa uudistetun moduulisyntaksin ja luokkien avulla. (Marvin 2015a.)

Uusi standardi asettaa paineita etenkin selaimille, palvelinpään ohjelmistoille ja Javascript-ohjelmakoodia kirjoittaville aloittaa käyttämään täysin ECMAScript 6-

standardia. Useat selaimet ovat jo aloittaneet uuden standardin toteuttamisen selainten omissa Javascript-moottoreissaan. Selainten osalta prosessi on hidas ja muutokset tulevat näkymään vaiheittain, sillä selainten tavoitteena on standardin uusien ominaisuuksien toteutuksen lisäksi myös niiden suorituskyvyn optimointi selaimessa. (Marvin 2015b.)

7.2 Jatkokehitys

Palvelimen toimintaa voitaisiin konkreettisesti testata esimerkiksi luomalla yksinkertaisen graafinen peli, joka käyttää vastaavia tiedonsiirtoprotokollia ja viestejä kuin opinnäytetyössä syntynyt benchmarking-sovellus. Pelillä voitaisiin havaita visuaalisesti tiedonsiirrossa kohdatut ongelmat reaaliaikaisuudessa, esimerkiksi miten pakettien viive ja häviäminen näkyy loppukäyttäjälle pelissä. Vaikka sovelluspalvelin on toteutettu Node.js-sovelluskehykseen, voi varsinainen asiakassovellus olla käytännössä millä tahansa ohjelmointikielellä kirjoitettu. Ainoana vaatimuksena asiakassovelluksen olisi se, että se tukee sovelluspalvelimen käyttämiä tiedonsiirtoprotokollia.

Palvelimen suorituskykyä olisi mahdollista parantaa hajauttamalla sovellus useaan sovellusprosessiin esimerkiksi Node.js:n Cluster-moduulin tai toisen fyysisen palvelintietokoneen avulla. Sekä Cluster-moduulin että usean palvelimen käyttöönotto vaatisi muutoksia sovelluksen arkkitehtuurin datakerrokseen, joka tulisi muuttaa käyttämään esimerkiksi erillistä tietokantaa apuna tiedon tallennukseen ja lukemiseen, koska yksittäisen sovellusprosessin tallentamaa tietoa ei välttämättä voida sujuvasti jakaa suoraan muille prosesseille. Erillinen tietokanta varmistaisi myös, että tiedon tallennus keskittyy yhteen paikkaan ja viimeisin versio datasta on aina kaikkien sitä tarvitsevien sovellusten saatavilla.

Opinnäytetyössä käytetyn UDP-tiedonsiirtoprotokollan Multicasting-lähetysmuodon hyödyntäminen saman viestin lähetyksessä usealle asiakassovellukselle voisi mahdollisesti vaikuttaa palvelimen suorituskykyyn ja tiedonsiirron nopeuteen. Tällöin palvelinsovelluksen ei tarvitse lähettää samaa

viestiä usealle käyttäjälle, vaan se hoituisi keskitetysti yhdellä viestin lähetyskutsulla.

Palvelinsovelluksen käyttämien tiedonsiirtoprotokollien yhteiskäyttö mahdollistaisi haluttujen tehtävien jakamisen käyttämään tiettyä tiedonsiirtoprotokollaa tehtävän luonteen mukaan nykyisen karkean jaon sijaan, jossa käytössä on aina vain yksi tiedonsiirtoprotokolla. Esimerkiksi reaaliaikaisuutta vaativat tehtävät voisivat käyttää UDP-protokollan tiedonsiirtotapaa ja vastaavasti vaativien, luotettavaa tiedonsiirtoa vaativien tehtävien kuten kokonaisten tiedostojen siirrosta vastaisi TCP-protokolla.

Lähteet

Capan, T. 2013. Why the hell would I use Node.js? A Case-by-Case tutorial. Toptal http://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js. 15.11.2015

Fankhauser D. 2012. Is Javascript the Future of Programming? Mashable. http://mashable.com/2012/11/12/javascript/#qwIsp2f3Z5qt. 17.11.2015

Fette, I & Melnikov, A. 2011. The WebSocket Protocol. Internet Engineering Task Force (IETF) http://tools.ietf.org/html/rfc6455. 5.10.2015

Google Developers. 2014. Introduction. Google Developers. https://developers.google.com/v8/intro. 20.10.2015

Google Developers. 2012. Design Elements. Google Developers. https://developers.google.com/v8/design. 1.10.2015

Hunt, C. 1998. TCP/IP-verkonhallinta. Jyväskylä: Suomen ATK-kustannus

Jia W, Zhou W. 2005. Distributed Network Systems From Concepts to Implementations. Springer. http://www.springer.com/us/book/9780387238395. 20.10.2015

JSON. 2015. Introducing JSON. JSON.org. http://www.json.org/. 14.11.2015

Kaarlo, K. 2002. TCP/IP-verkot. Jyväskylä: Docendo.

Kaazing. 2015. About HTML5 WebSockets. Kaazing. https://www.websocket.org/aboutwebsocket.html. 30.9.2015

Koskimies K & Mikkonen T. 2005. Ohjelmistoarkkitehtuurit. Helsinki: Talentum

Kozierok, C. 2005. The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference. https://books.google.fi/books?id=Pm4R-gYV2w4YC&printsec=frontcover&hl=fi&source=gbs_atb#v=onepage&q&f=false. 5.10.2015a

Kozeriok, C. 2005. TCP/IP Routing Protocols (Gateway Protocols). The TCP/IP Guide. http://www.tcpipguide.com/free/t_TCPIPRoutingProtocols-GatewayProtocols.htm. 30.9.2015b

Marvin R. 2015. The future of JavaScript is (almost) now. SDtimes. http://sdtimes.com/the-future-of-javascript-is-almost-now/. 17.11.2015a

Marvin R. 2015. The future of JavaScript is (almost) now. SDtimes. http://sdtimes.com/the-future-of-javascript-is-almost-now/4. 17.11.2015b

Node.js. 2015. Node.js v4.2.1 Documentation. Node.js. https://nodejs.org/api/cluster.html 21.10.2015

Npm. 2015. About npm. Npm. https://www.npmjs.com/about. 29.9.2015

Postel, J. 1981. Transmission Control Protocol. Information Sciences Institute University of Southern California. https://tools.ietf.org/html/rfc793. 29.9.2015

Pusher. 2015. What are websockets? Pusher. https://pusher.com/websockets. 30.9.2015

Sormunen L. 2015. Node.js server application. Github. https://github.com/LeoSorm/gameserver. 28.11.2015a

Sormunen L. 2015. Node.js server benchmarking application. Github https://github.com/LeoSorm/gameserver-benchmarker. 28.11.2015b

Takada M. 2015. Mixu's Node book. http://book.mixu.net/node/single.html 3.11.2015

Teixeira, P. 2013. Professional Node.js Building Javascript Based Scalable Software. USA: John Wiley & Sons.