Petri Rautiainen

Reaaliaikaisen verkkosivun toteuttaminen

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Mediatekniikan koulutusohjelma

Insinöörityö

3.4.2016

Metropolia

Tekijä Otsikko Sivumäärä Aika	Petri Rautiainen Reaaliaikaisen verkkosivun toteuttaminen 34 sivua 3.4.2016
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Mediatekniikka
Suuntautumisvaihtoehto	Digitaalinen media
Ohjaajat	Ryhmäpäällikkö Pasi Nissinen Lehtori Ilkka Kylmäniemi

Insinöörityön tavoitteena oli kehittää jo olemassa olevaa taksinkuljettajien tilauslistaa niin, että siitä saatiin entistä käytettävämpi. Tilauslista päivitettiin vanhanaikaisin menetelmin, mikä aiheutti erilaisia käytettävyyteen ja toiminnallisuuteen liittyviä ongelmia. Tilauslista on verkkosivu, mutta sitä käytetään upotettuna Android-sovelluksessa.

Projektin tavoitteena oli tutkia nykyaikaisia reaaliaikaisia web-teknologioita, tehdä tutkimustulosten pohjalta suunnitelma tilauslistan parantamiseksi ja lopuksi toteuttaa suunnitelman mukainen päivitys järjestelmään.

Insinöörityössä tutkittiin teknologioita, joiden avulla web-sivuston ja palvelimen välille voidaan luoda jatkuva yhteys niin, että palvelin voi päivittää tietoa sivulle heti, kun uutta tietoa on saatavissa. Tutkimustulosten avulla laadittiin suunnitelma, jossa tilauslistan palvelinpuolen rinnalle tehdään WebSocket-palvelin, jonka avulla tilauslistaa päivitetään reaaliajassa.

Suunnitelman pohjalta kehitettiin Ratchet-PHP-kirjastolla WebSocket-palvelin, joka toimi muuten moitteettomasti, mutta se ei ollut tuettu vanhemmissa Android-laitteissa WebView-elementissä. Paremman selaintuen saavuttamiseksi toinen versio toteutettiin Node.js-palvelimella ja Socket.io-kirjastolla. Socket.io:lla toteutettu versio toimi myös vanhemmilla Android-versioilla, joten se otettiin käyttöön.

Insinöörityön lopputuloksena syntyi tilauslista, johon mobiilisovelluksesta tehdyt uudet tilaukset ilmestyivät reaaliajassa. Toteutuksesta tuli kehittyneempien teknologioiden hyödyntämisen seurauksena skaalautuvampi ja reaaliaikaisuuden tuomien etujen vuoksi myös käytettävämpi.

Avainsanat	reaaliaikainen web, WebSocket, Socket.io, Node.js, Ratchet, PHP, JavaScript, HTML5

Author Title Number of Pages Date	Petri Rautiainen Creating a real-time web application 34 3 April 2016
Degree	Bachelor of Engineering
Degree Programme	Media Technology
Specialisation option	Digital Media
Instructors	Pasi Nissinen, Head of Team Ilkka Kylmäniemi, Senior Lecturer

The aim of this study was to further develop an already existing order list for taxi drivers, in order to increase its usability. The order list is a website, but it is used embedded inside an android application. The order list used to be updated using outdated technologies, which resulted in a variety of usability and functionality related problems.

The aim of the project was to research modern real-time web technologies to create a plan to improve the order list based on the results of the study and finally to implement an upgrade for the system by utilizing the plan.

A number of technologies were researched, which enable a server and a website to create a persisting connection between the two parties so that the server can update the webpage immediately when new information is available. Based on the results of the study, a plan was made to implement a WebSocket server that would update the order list in real-time.

The plan was executed by implementing the WebSocket server using Ratchet PHP library. The server functioned well in new browsers, but the technology was not supported by older Android devices. In order to achieve better browser support, a second version was implemented utilizing Node.js and Socket.io library. The second version functioned well in older Android devices.

As a result of this study, new orders placed in a mobile application now appear in the order list in real-time. By using more advanced modern technologies, the application's scalability has been greatly improved and the usability is considerably better because of the real-time nature of the application.

Keywords	realtime web, WebSocket, Socket.io, Node.js, Ratchet, PHP, JavaScript, HTML5

Sisällys

Lyhenteet

1 Johdanto 1

2 Reaaliaikaisen webin käyttökohteet 1

3 Asiakasohjelman ja palvelimen välinen tiedonsiirto 4

3.1 Asiakas-palvelinmalli 4
3.2 Reaaliaikaiset web-teknologiat 6
3.2.1 Polling-tekniikka 6
3.2.2 Long polling -tekniikka 7
3.2.3 WebSocket-teknologia 9
3.2.4 Server Sent Events -teknologia 13
3.2.5 WebRTC-teknologia 15

4 Reaaliaikaisen webin työkalut 17

4.1 Ratchet-WebSocket-kirjasto 18
4.2 Node.js-palvelin 20
4.3 Socket.io-kirjasto 21
4.4 AutobahnJS-WAMP-rajapinta 22

5 Tilausnäkymän suunnittelu ja toteutus 22

5.1 Suunnittelu 22
5.2 Toteutus 26
5.2.1 Ratchet-kirjasto 26
5.2.2 Socket.io-kirjasto 29
5.2.3 Yhteyden salaaminen 31
5.2.4 Viimeistely 32

6 Yhteenveto 33

Lähteet 35

Lyhenteet

IDE	Integrated development environment. Tietokonesovellus sovelluskehitykseen.
HTTP	Hypertext transfer protocol. Protokolla, jota selaimet ja palvelimet käyttävät tiedonsiirtoon.
CRM	Customer relationship management. Asiakaslähtöinen ajattelutapa organisaatiossa ja siihen liittyvät tietojärjestelmät.
WLAN	Wireless local area network. Langaton lähiverkkotekniikka, jolla voidaan yhdistää laitteet ilman kaapelia.
IETF	Internet Engineering Task Force. Internet-protokollien standardoinnista vastaava organisaatio.
API	Application programming interface. Ohjelmointirajapinta.
SHA1	Secure Hash Algorithm 1. Kryptografinen tiivistefunktio.
TCP	Transmission Control Protocol. Tietoliikenneprotokolla, jolla luodaan yhteyksiä tietokoneiden välille.
GPS	Global Positioning System. Maailmanlaajuinen satelliittipaikannusjärjestelmä.
WAMP	Web Application Messaging Protocol. WebSocket-protokollan alaprotokolla.
CORS	Cross-origin resource sharing. Mekanismi, joka mahdollistaa resurssien noutamisen eri verkkotunnuksista.
SIP	Session Initiation Protocol. IP-puhelinyhteyksien luonnista vastaava tietoliikenneprotokolla.

XMPP	Extensible Messaging and Presence Protocol. Viestipainotteinen tietoliikenneprotokolla, joka perustuu XML:ään.

1 Johdanto

Asiakas-palvelinmalli, joka on keskeinen osa HTTP-protokollan toimintatapaa, on yhä yleisimmin käytetty arkkitehtuuri webissä. Asiakasohjelman on tehtävä pyyntö palvelimelle, jotta se voi saada vastauksena uutta sivulla esitettävää tietoa. Tällainen malli on epäkäytännöllinen nykyaikaisissa dynaamisissa web-sovelluksissa, joissa tieto voi muuttua erittäin nopeasti. Muuttuva tieto sijaitsee palvelimella, mutta palvelin ei voi itse kertoa asiakasohjelmalle, että uutta tietoa olisi saatavilla, vaan sen täytyy odottaa, kunnes asiakasohjelma pyytää sitä. Asiakasohjelma ei voi kuitenkaan tietää, milloin uutta tietoa on saatavissa.

Tämän insinöörityön tarkoituksena on kehittää jo olemassa olevaa taksinkuljettajien tilausnäkymää. Tilausnäkymä on verkkosivu, joka päivittää tilaukset AJAX-kyselyllä 15 sekunnin väliajoin. Sivu kärsii edellä mainituista HTTP:n rajoitteista. Tavoitteena on tutkia nykyaikaisia reaaliaikaisen webin teknologioita ja laatia tutkimustulosten pohjalta suunnitelma ja toteutus sivun nykyaikaistamiselle.

Työn tilaaja on Finnish Net Solutions Oy, joka on ohjelmistoalan yritys. Projektin tavoitteena on tehdä tilauslistasta käytettävämpi ja reaaliaikainen. Reaaliaikaisuudella tarkoitetaan sitä, että uusi tieto näytetään asiakkaalle 500 millisekunnin sisällä siitä, kun tieto saapuu palvelimelle. Uusien tilausten tulee siis näkyä tilauslistassa lähes välittömästi, kun palvelin vastaanottaa ne. Projektiin sisältyy paljon tutkimustyötä, ja tuloksia voidaan hyödyntää tulevissa projekteissa.

2 Reaaliaikaisen webin käyttökohteet

Reaaliaikaisten web-sovellusten kehittäminen on ollut mahdollista jo pitkään. Ne ovat kuitenkin yleistyneet huomattavasti vasta tämän vuosikymmenen aikana [1]. Jo 1990-luvun lopulla oli Flash- ja Java-pohjaisia selaimessa toimivia sovelluksia, joissa pystyi lähettämään ja vastaanottamaan viestejä reaaliajassa muiden osallistujien kesken. Osa sovelluksista oli myös monimutkaisia – esimerkiksi peli, jossa yksi pelaaja sai tehtäväkseen kuvailla muille piirtämällä satunnanvaraisesti valitun sanan, jonka muut yrittivät arvata. Yleistymiseen vaikuttaa varmasti paljon se, että teknologiat ovat kehittyneet helppokäyttöisemmiksi ja standardisoituneet. Pääsyy on kuitenkin se käyttökokemus, jota

käyttäjät ovat saaneet etenkin sosiaalisen median sovelluksista kuten Facebook, Twitter ja Google+. [1.] Tällaisesta käyttökokemuksesta on vaikea luopua, joten käyttäjät vaativat reaaliaikaisuutta myös muilta sovelluksilta.

Vuosien kuluessa reaaliaikaisille web-sovelluksille on kehittynyt useita käyttökohteita. Finanssiala on eräs ensimmäisiä reaaliaikaisia web-teknologioita käyttänyt ala. Tämä on ymmärrettävää, koska osake- ja rahastomarkkinoilla osto- ja myyntipäätösten täytyy olla nopeita. [2.] Nopeus on juuri se ominaisuus, jota reaaliaikaisuus edustaa. Heti kun jotain uutta tietoa on saatavissa, se välitetään käyttäjälle. Jopa alle sekunnin myöhästyminen voi aiheuttaa suuret tappiot tai mitätöidä tuottomahdollisuudet. Vastaavasti nopeasti ennen kilpailijoita suoritettu transaktio voi johtaa suuriin tuottoihin. [2.]

Phil Legetter [2] kirjoittaa artikkeleita, pitää esityksiä ja konsultoi reaaliaikaisesta webistä. Hän kertoo liittyneensä Caplin Systems -nimiseen yritykseen vuonna 2001. Caplin Systems on luonut kehitysympäristön sijoituskaupankäyntiä varten. Ideana on se, että kauppiaat voivat räätälöidä Caplin Systemsin kehitysympäristön avulla itselle sopivan kaupankäyntityökalun. Alun perin heidän tarkoituksenaan oli vain näyttää kurssitiedot reaaliajassa selainpohjaisessa sovelluksessa. Myöhemmin ominaisuudet kehittyivät ja myös kommunikointi kauppiaiden ja kaupankäyntisovellusten kanssa sekä kauppojen tekeminen tuli mahdolliseksi. Readwriten [3] artikkelissa esitellään Kaazing-niminen yritys, joka käyttää HTML5 WebSocket-teknologiaa välittämään finanssidataa reaaliajassa pankeille. Sen asiakkaina toimivat pankit käyttivät aiemmin erillisiä paikallisille työasemille asennettuja sovelluksia. Kaazing toimii verkkoselaimessa, ja se lupaa järjestelmän suorituskyvyn riittävän erittäin suureen käyttäjä- ja viestimäärään.

Yhtiöt voivat hyödyntää reaaliaikaisia web-teknologioita aktiivisuuden mittaamiseen. Mitattavia asioita voivat olla esimerkiksi sähköpostien ja muiden viestien tai sivunlatauksien määrä intranetissä tai verkkosivuilla. [3.] Myös asiakkaiden käyttäytymisen seuraaminen verkkopalveluissa on mahdollista. Tällainen tiedonkeruu muistuttaa Big dataa, joka on ollut viime vuosina selkeä trendi. Reaaliaikaisilla web-teknologioilla ei voi suoraan analysoida tietoa, mutta niitä voidaan käyttää väylänä, jonka kautta tieto toimitetaan analysointia varten.

Big data ei sovellu kaikkiin analysointiongelmiin, koska se on hidas prosessi. Tällaisia ongelmia ovat esimerkiksi sijoituskaupankäynti, vilpinhavainta ja järjestelmien valvonta. Näissä esimerkkitapauksissa tulokset on saatava nopeasti. [4.] Analysointi suoritetaan

suurelle ajan myötä kerätylle datamäärälle, josta iso osa saattaa olla täysin turhaa tietoa. Nopea data on uusi yleistyvä termi, joka kuvaa prosessia, jossa tieto lähetetään välittömästi analysointiin. Analysointi on jatkuva prosessi, jonka seurauksena myös analysointitulos päivittyy jatkuvasti. [4.] InfoWorld [5] kutsuu nopeaa dataa Big datan seuraavaksi askeleeksi.

Reaaliaikaisista web-teknologioista voi olla hyötyä asiakkaiden käyttäytymisen seuraamisessa, mutta yritykset voivat hyödyntää teknologioita myös asiakassuhteiden ylläpitämisessä. Yritysten CRM-järjestelmiin (Customer relationship management) voidaan lisätä sosiaalisia ulottuvuuksia. Asiakkaat voivat kirjoittaa tuotteen tai palvelun käytössä ilmaantuvista ongelmista tukipyyntöjä, jotka näytetään tukihenkilökunnalle välittömästi. Asiakkaan ja tukihenkilön välille on mahdollista avata suora reaaliaikainen keskustelu, jolloin tiedon vaihtaminen onnistuu nopeasti. [10, s. 14.] Yhteen ongelmaan liittyvä tukitoiminta voi kestää huomattavasti kauemmin, jos lisäkysymyksiä lähetetään sähköpostitse. Tukihenkilö ehtii todennäköisesti siirtyä jo seuraavaan tapaukseen, ennen kuin lisäkysymykseen tulee vastaus.

Yhtiöiden kaltaisten suurten yhteisöjen lisäksi reaaliaikaisista web-teknologioista on paljon hyötyä myös tiimitasolla. Yhteistyötä edistävät sovellukset ovat eräitä edistyksellisimpiä sovelluksia, jotka hyödyntävät näitä teknologioita [6]. Tuttuja esimerkkejä tällaisista sovelluksista ovat Googlen web-pohjaiset toimistosovellukset: Google Docs, Google Sheets ja Google Slides. Nämä sovellukset mahdollistavat saman asiakirjan muokkaamisen usealta eri työasemalta samaan aikaan. Yhden osallistujan asiakirjaan tekemät muutokset näkyvät käytännössä välittömästi myös muiden osallistujien näytöillä. Yhteistyösovelluksilla voi säästää paljon aikaa, koska koko tiimi voi osallistua muokkaamiseen ja keskusteluun samaan aikaan, vaikka he eivät olisikaan fyysisesti samassa paikassa [2]. Phil Legetter [2] odottaa paljon uusia innovaatioita tämäntyyppisten sovellusten keskuudessa.

Muutamia hyvin innovatiivisia yhteistyösovelluksia on jo olemassa. Cloud 9 IDE (Integrated development environment) on sovelluskehitysympäristö, jossa sovelluskehittäjät voivat ohjelmoida yhdessä reaaliajassa. Sovelluksessa voi ohjelmoinnin lisäksi keskustella, käyttää Unix-komentoriviä ja jopa testata ja suorittaa kehityksessä olevaa sovellusta [1]. Cloud 9 IDE tukee useita ohjelmistokieliä ja kirjastoja: Django, WordPress, Meteor, C++, Ruby on Rails, Node.js, PHP ja HTML5 [7]. Toinen innovatiivinen sovellus on

Murally, joka on suunnitteluun ja ideointiin kehitetty työkalu. Sovelluksella voi luoda nopeasti seinämaalaustyyppisiä visuaalisia suunnitelmia yhteistyössä muiden kanssa. [1.] Kehittäjät kuvailevat sovellusta työkaluksi, jolla voi työskennellä samassa tilassa ikään kuin koko tiimi olisi samassa huoneessa [8].

Pelialalla reaaliaikaisuus on tarpeellinen useissa moninpeleissä. Pelin tyyppi saattaa vaikuttaa siihen, riittävätkö HTTP:n perinteiset GET- ja POST-metodit päivittämään pelitilanteen kaikille pelaajille. Esimerkiksi shakki ja korttipelit saattavat toimia tarpeeksi hyvin ilman reaaliaikaisia web-teknologioita, koska muutoksiin ei tarvitse reagoida nopeasti. Shakissa siirtoaika voi olla päivien, viikkojen tai jopa kuukausien pituinen. GET- ja POST-metodit ovat liian hitaita pelitilanteen päivittämiseen reaaliaikaisuutta vaativissa peleissä. Jos pelissä useat eri pelaajat liikuttavat pelihahmoaan pelimaailmassa, tilannepäivityksiä tarvitaan mahdollisesti jopa 33–66 sekunnissa. HTTP:llä näin nopean päivitystahdin toteuttaminen ei ole mahdollista. [9.]

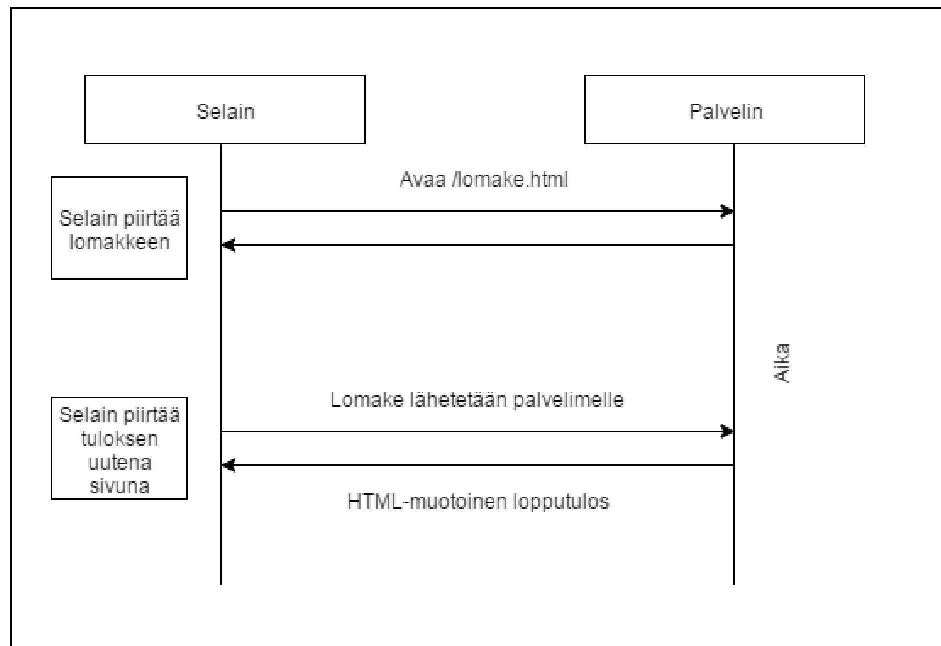
Esineiden internetin (Internet of things) sovelluksissa reaaliaikaisia web-teknologioita voidaan käyttää kaikkeen esineen ja sen hallintalaitteen väliseen kommunikaatioon [1]. Kommunikaatio voi olla sensoridatan näyttämistä hallintalaitteella tai esineen ohjaamista komennoilla. Esineiden internet on suosittu Arduino- ja Raspberry Pi -pientietokoneiden käyttäjien parissa. Pientietokoneet voidaan yhdistää internetiin WLAN-moduulin ja langattoman verkon kautta, mutta myös 3G-moduuleita on markkinoilla. 3G-yhteyden luominen on mahdollista, kunhan laite on mobiiliverkon kantaman sisällä. Esimerkkiprojekteja ovat älypuhelimella seurattavat lämpötila-, ilmanpaine- ja maankosteusmittarit, etäohjattavat kahvinkeittimet ja kotiautomaatio [11; 12]. Kotiautomaatiolla tarkoitetaan järjestelmää, joka ohjaa kodin muita laitteita automaattisesti.

3 Asiakasohjelman ja palvelimen välinen tiedonsiirto

3.1 Asiakas-palvelinmalli

Valtaosa internetistä perustuu yhä asiakas-palvelinmalliin. Tätä mallia toteutetaan sitä varten suunnitellulla HTTP:llä. [13.] Malli kuvaa ympäristöä, jossa sovellus on jaettu kahteen osaan, joista molemmat toimivat todennäköisesti eri käyttöjärjestelmissä ja tietokoneissa. Osat toimivat kuitenkin yhdessä, ja niiden tarkoituksena on tuottaa loppukäyttä-

jälle jokin palvelu. Asiakas on se osa, jonka käyttäjä näkee, ja yleensä se toimii työasemalla, jota loppukäyttäjä käyttää. Palvelin sen sijaan ei ole suoraan havaittavissa loppukäyttäjälle, vaan se koetaan osaksi palvelun toimintaa. [14.] Kuvassa 1 on esitetty yksinkertainen verkkoselaimen ja palvelimen välinen vuorovaikutus. Käyttäjä avaa sivun lomake.html, jolloin selain pyytää tiedoston palvelimelta. Palvelin lähettää pyydetyn tiedoston vastauksena selaimelle, joka piirtää sivunäkymän HTML-tiedoston ohjeiden mukaisesti.



Kuva 1. Selaimen ja palvelimen välitys vuorovaikutus HTTP:tä käyttäen [10, s. 9].

Tässä esimerkissä sivu on lomake. Käyttäjä syöttää lomakkeelle kysytyt tiedot ja lähettää lomakkeen palvelimelle, joka palauttaa selaimelle jälleen HTML-muotoisen vastauksen. Lopuksi selain piirtää sivun vastaanotetun vastauksen ohjeiden mukaisesti.

Asiakas-palvelinmalli on hyvä siihen tarkoitukseen, johon se on suunniteltu. Se ei ole kuitenkaan kestävä ratkaisu, jos halutaan hyödyntää dynaamisempia sovelluksia [15]. Ongelma mallissa on sovelluksen jakautuminen kahteen osaan ja se, että asiakasohjelman täytyy tehdä pyyntö saadakseen dataa palvelimelta. Tieto sijaitsee palvelimella, ja useissa sovelluksissa se on nopeasti muuttuvaa tai sitä tulee jatkuvasti lisää, mutta asiakasohjelma ei voi tietää, milloin sen pitäisi pyytää uutta dataa palvelimelta. Olisi luonnollisempaa, että palvelin välittäisi tiedon muutoksista tai uudesta datasta välittömästi asiakasohjelmalle. Jason Hoffmanin [15] mukaan perinteiseen pyyntöön ja vastaukseen perustuva datansiirto ei ole kestävää, kun datamäärän kysyntä kasvaa. Hän ennustaa

myös nopeaa loppua asiakas-palvelinmallille. Itse en usko koko mallin katoamiseen. HTTP:n rinnalle tarvitaan vain muita työkaluja.

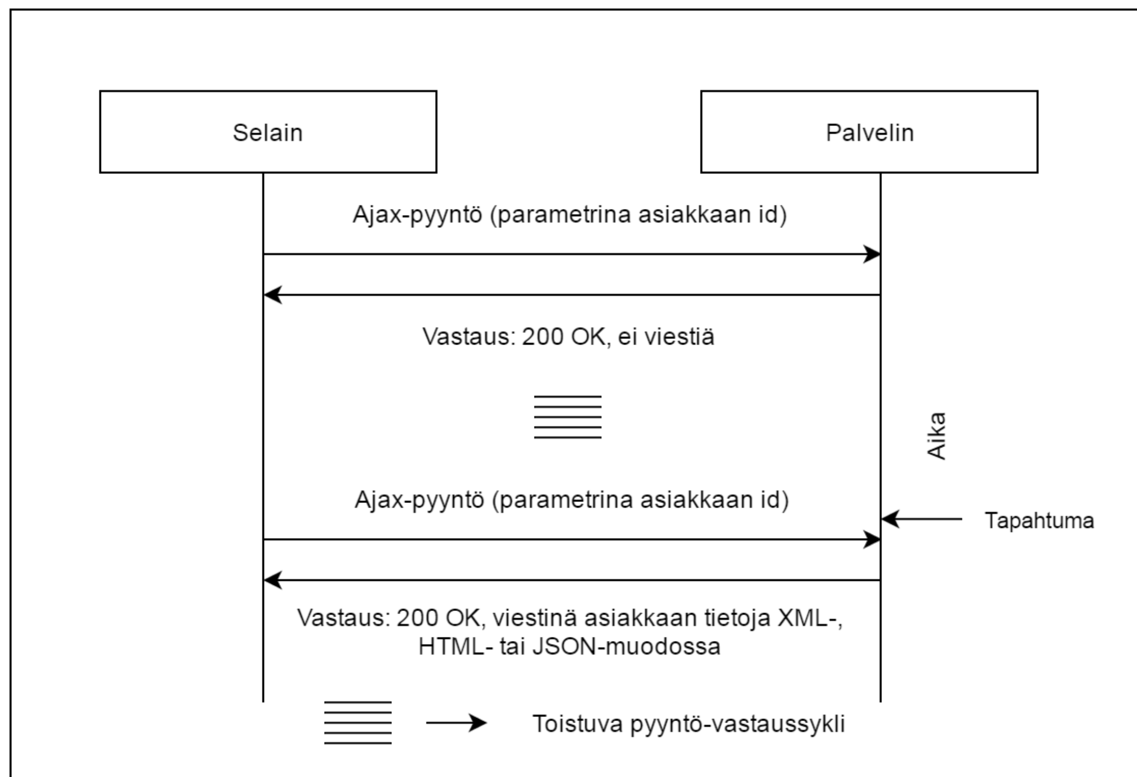
3.2 Reaaliaikaiset web-teknologiat

Reaaliaikaisten tai reaaliaikaisilta vaikuttavien sovelluksen kehittämiseen on keksitty useita erilaisia tapoja. Osa niistä on ollut hieman epämääräisiä ratkaisuja, jotka soveltavat HTTP:n ominaisuuksia niin, että sen rajoittavia tekijöitä on voitu kiertää. Epämääräisissä ratkaisuissa on paljon huonoja puolia etenkin skaalautuvuuden suhteen, mutta hyvin toteutettuna niillä voidaan saada aikaan kohtuullinen lopputulos. Uusimmat teknologiat sen sijaan ovat täysin hyväksyttyjä ja standardoituja menetelmiä, jotka on suunniteltu jo protokollatasolla reaaliaikaiseen käyttöön. Uusien teknologioiden varjopuolena on, että voi mennä vuosia, ennen kuin ne saavat täydellisen selain- ja ohjelmistotuen. [10, s. 8.]

3.2.1 Polling-tekniikka

Polling on tekniikka, jolla voidaan saada reaaliaikainen vaikutelma verkkosivustolle. Sitä hyödynsi ensimmäisenä Microsoft Internet Explorerin oletuskotisivulla. Microsoftin menetelmä sai nimen XMLHTTP. Sitä käytettiin lataamaan asynkronisesti XML-dataa sivustolle JavaScriptin avulla niin, ettei koko sivua tarvinnut päivittää. Selainkehittäjät Mozilla, Safari ja Opera omaksuivat menetelmän, ja se sai uuden nimen XMLHttpRequest (lyhennettynä XHR). Lopulta tunnetuimmaksi kutsumanimeksi tuli AJAX Jesse James Garettin artikkelin ”Ajax: A New Approach to Web Applications” myötä. [10, s. 10.] Polling on AJAX-pyyntöjen toistuvaa suorittamista tietyin aikavälein [16, s. 26].

Kuvassa 2 on havainnollistettu prosessi, jossa sivulle päivitetään tietoa polling-tekniikalla ilman sivunpäivitystä. Selaimesta tehdään JavaScriptillä AJAX-pyyntö palvelimelle. Pyynnön mukana voidaan lähettää myös parametreja, joilla saattaa olla vaikutusta vastauksen sisältöön. Tässä esimerkissä parametrina käytetään asiakkaan tunnistetta (id), jota palvelin käyttää käyttäjän tunnistamiseen. Jos mitään uutta tietoa ei ole saatavissa, palvelin antaa vain vastauskoodin 200, joka tarkoittaa onnistunutta kyselyä. Kun jotain uutta tietoa on saatavissa, palvelin palauttaa saman vastauskoodin lisäksi myös vastausviestin, joka voi olla teksti-, XML-, HTML- tai JSON-muotoinen. Pyyntöjä tehdään toistuvasti niin kauan, kuin käyttäjä vierailee sivulla. [16, s. 26; 10 s. 11.]



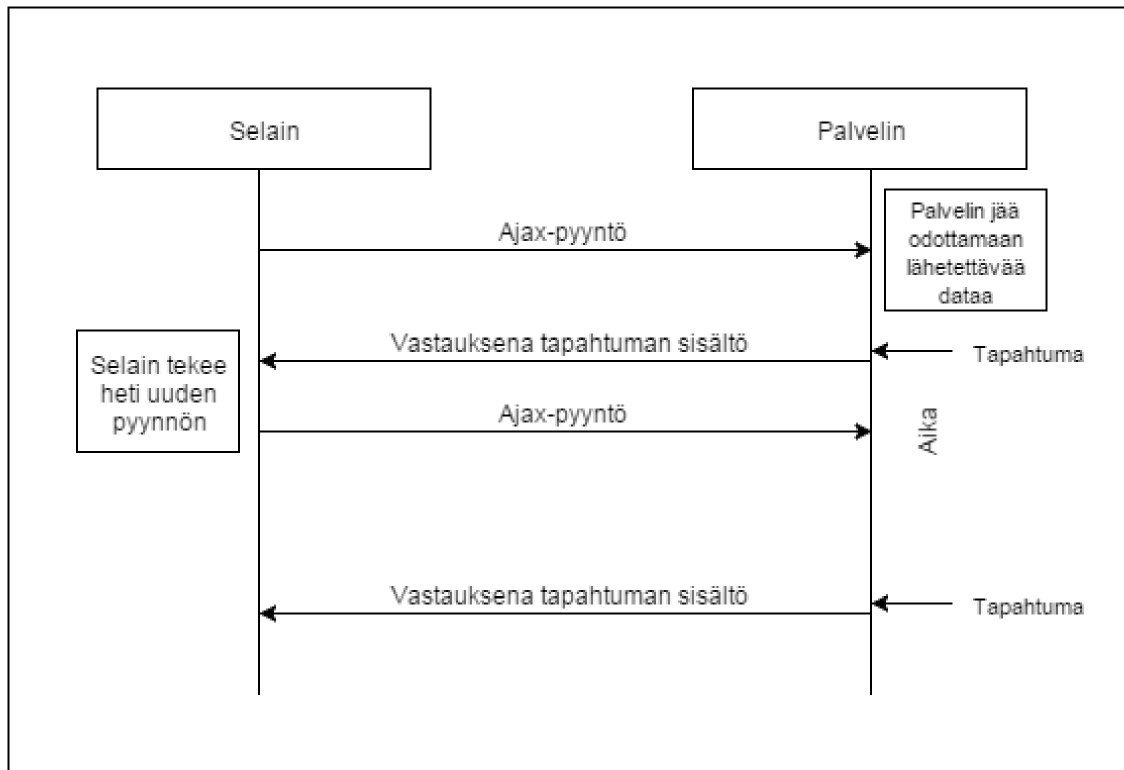
Kuva 2. Tiedon päivittäminen sivulle Polling-tekniikalla ilman sivunlatausta [16, s. 26; 10, s. 11].

Polling soveltuu vain käyttötapauksiin, joissa samanaikaisia käyttäjiä on hyvin rajattu määrä tai käytettävissä olevat laitteet tai asiakasohjelmat eivät tue uudempia menetelmiä [16, s. 26]. Sen toteuttaminen on helppoa, mutta palvelin palauttaa aina tyhjän vastauksen ja asiakasohjelma tekee turhan kyselyn silloin, kun uutta tietoa ei ole. Lisäksi tieto ei ole oikeasti reaaliaikaista, vaan kaikki riippuu täysin päivitystahdista. Tämä aiheuttaa paljon turhaa tietoliikennettä asiakkaan ja palvelimen välillä. Palvelin joutuu myös käsittelemään tietoa silloin, kun mitään lähetettävää ei ole. Näiden heikkouksien vuoksi tekniikka on huonosti skaalautuva. [10, s. 11.] Se ei ole tehokas tekniikka käsittelemään massiivisia käyttäjämääriä, joita on nykyään useilla eri sovelluksilla [16, s. 26].

3.2.2 Long polling -tekniikka

Long polling on kehittyneempi versio polling-tekniikasta. Se on suunniteltu paikkaamaan joitakin polling-tekniikan heikkouksia. Tässä tekniikassa palvelimelle lähetetty pyyntö joko aikakatkaistaan tai palvelin lähettää vastauksena asiakasohjelmalle uutta tietoa. [16, s. 28.] Palvelin ei siis vastaa pyyntöön heti, vaan se keskeyttää pyynnön, kunnes se saa jotain uutta asiakasohjelmalle välitettävää tietoa. Kun palvelin vastaa pyyntöön, yhteys katkeaa. Asiakasohjelma aloittaa uuden syklin heti, kun se vastaanottaa palvelimen

vastauksen. Tämä tapahtuu yksinkertaisesti niin, että asiakasohjelma tekee uuden pyynnön palvelimelle heti, kun se saa vastauksen tai yhteys aikakatkaistaan. [10, s. 12.] Kuvassa 3 esitellään esimerkki long polling -tekniikan käytöstä.



Kuva 3. Long polling -tekniikan käyttö [10, s. 12].

Kalalin ja Mehtan [16, s. 28] mukaan long polling -tekniikan etu polling-tekniikkaan verrattuna on pienempi pyyntöjen määrä, mikä vähentää resurssien kulutusta ja lisää siten skaalautuvuutta. Sovelluskehittäjien välisistä keskusteluista ilmenee kuitenkin, että parempi skaalautuvuus tällä tekniikalla ei ole itsestäänselvyys. Esimerkiksi käyttäjä Minko Gechev [17] kirjoittaa ongelmista long polling -tekniikan toteuttamisessa Apache-web-palvelinohjelmiston päälle. Hän kertoo, kuinka Apachessa jokainen yhteys käyttää myös yhden säikeen, joita on käytettävissä rajoitettu määrä. Kun määrä tulee täyteen, palvelin ei enää pysty vastaamaan uusiin yhteyksiin, ennen kuin säikeitä vapautuu [17].

Samanaikaisten yhteyksien määrän voi muuttaa Apachen asetuksissa, mutta jokainen säie varaa myös huomattavan määrän muistia [18; 19]. Yksi yhteys vie noin 2–15 Mt muistia riippuen siitä, toimittaako Apache asiakkaalle vain staattisen sivun vai dynaamista sisältöä esimerkiksi PHP:n ja tietokannan avulla [19]. Tuhat samanaikaista long polling -yhteyttä saattaa varata palvelimelta 2 000 Mt muistia tai jopa enemmän. Lisäksi

kuten aiemmin mainittiin, yhteys ei ole täysin jatkuva, vaan se katkeaa aina joko aikakatkaisuun tai siihen, kun palvelin lähettää tietoa asiakasohjelmalle. Käytännössä Apachella täytyisi olla aina suuri määrä säikeitä valmiina uusia yhteyksiä varten, mutta tämä varaisi paljon resursseja myös silloin, kun yhteyksiä ei ole yhtään. Muussa tapauksessa Apache joutuisi aloittamaan uusia säikeitä aina, kun uusi yhteys luodaan. Säikeiden käynnistäminen ja sulkeminen kuluttaa paljon resursseja, eikä palvelin pysty vastaamaan pyyntöihin samaan aikaan [20].

Kuten edellä mainituista ongelmista voidaan päätellä, pieniä projekteja lukuun ottamatta long polling asettaa rajoituksia palvelinpuolen ohjelmistoihin. Jos toteutukseen valitaan sopivat ohjelmistot ja työkalut, long polling voi olla kohtuullisen tehokas tekniikka reaaliaikaisen web-sovelluksen kehittämiseen. Hyvä web-palvelinohjelmisto tähän tarkoitukseen olisi esimerkiksi Nginx, joka ei aloita erillistä säiettä jokaiselle yhteydelle. Vaihtoehtoisesti perinteisen palvelinohjelmiston rinnalle voi kirjoittaa oman palvelimen hyödyntämällä jotain asynkronista tapahtumapainotteista kirjastoa tai työkalua, kuten Twisted (Python), Node.js (JavaScript) tai ReactPHP (PHP). Tekniikka on hyvin laajasti tuettu eri selaimissa, joka on ehdottomasti sen vahvuus. Se tuottaa kuitenkin paljon turhaa tietoliikennettä, koska HTTP-yhteyden muodostamiseen sisältyy aina palvelimen ja asiakkaan välinen kättely, ennen kuin varsinaista dataa voidaan siirtää. Uudemmissa tekniikoissa ei myöskään tarvita HTTP-otsikkotietoja erikseen jokaiselle viestille.

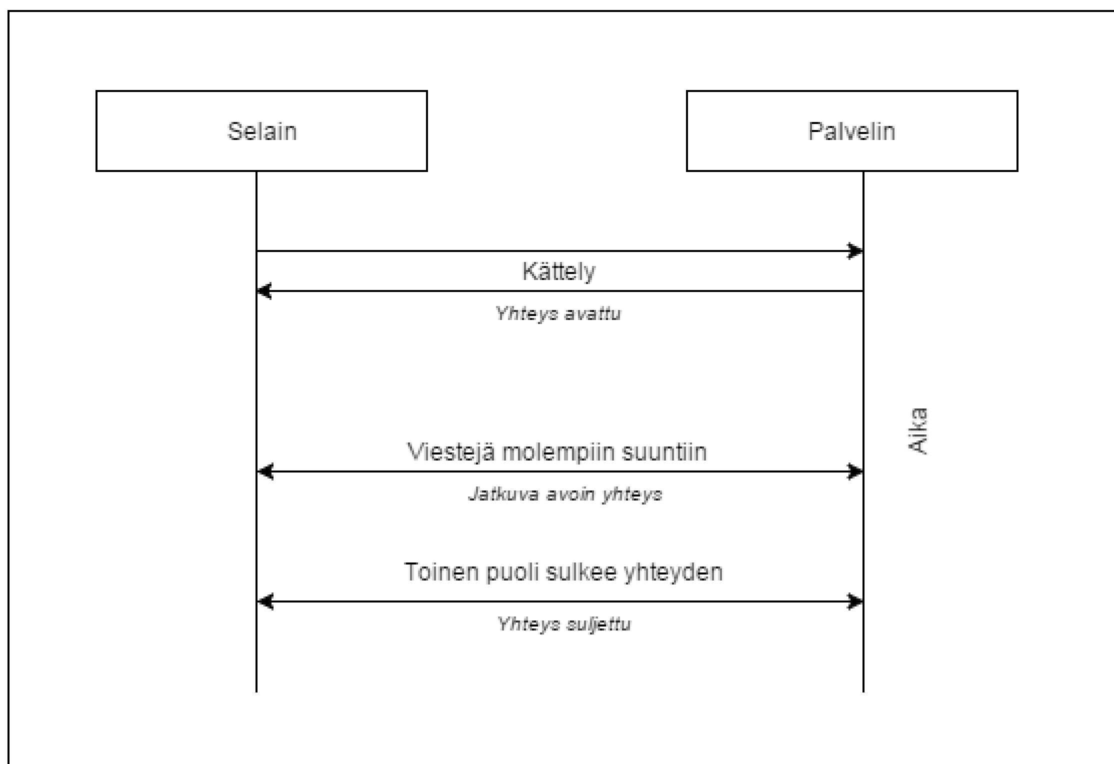
3.2.3 WebSocket-teknologia

WebSocket-teknologia on ensimmäinen tässä raportissa esiteltävä niin sanottu uusi teknologia, joka on varta vasten suunniteltu asiakasohjelmien ja palvelinten väliseen reaaliaikaiseen vuorovaikutukseen. Se on osa HTML5-kieltä, joka on HTML:n uusin versio. WebSocket-teknologia sisältää kaksi standardia, jotka molemmat julkaistiin vuonna 2011. Nämä standardit ovat IETF:n standardoima WebSocket-protokolla ja W3C:n standardoima WebSocket API. WebSocket-protokolla määrittelee esimerkiksi, miten yhteydestä tehdään turvallinen, miten yhteys muodostetaan ja miten data siirretään. WebSocket API määrittelee WebSocket-ohjelmointirajapinnan. Rajapinnan standardisointi mahdollistaa sen, että verkkosivustolla käytetty WebSocket-teknologia toimii esimerkiksi kaikilla eri selainkehittäjien selaimilla. [21; 22.]

WebSocket-teknologia on työkalu, joka vastaa nykyaikaisten web-sovellusten skaalautuvuus- ja joustavuustarpeisiin. WebSocket-yhteys on täysin kaksisuuntainen, minkä

vuoksi sekä asiakasohjelma että palvelin voivat lähettää toisilleen dataa minä hetkenä hyvänsä niin kauan, kuin yhteys on muodostettu. Tämä tarkoittaa myös, että palvelin voi lähettää asiakasohjelmalle dataa ilman, että se vastaanottaa ensin asiakasohjelmalta erillisen pyynnön. Samoin asiakasohjelman ei tarvitse muodostaa uutta yhteyttä palvelimeen, kun se lähettää palvelimelle dataa. [16, s. 37.] Kaksisuuntainen yhteys on juuri sitä, mitä tarvitaan reaaliaikaisen web-sovelluksen toteuttamiseen, koska nyt palvelin voi lähettää dataa asiakasohjelmalle heti, kun sitä on saatavilla.

Kuvassa 4 on esitetty WebSocket-protokollaan perustuvan yhteyden eri vaiheet ajan suhteen. Yhteys aloitetaan aina palvelimen ja asiakasohjelman välisellä kättelyllä. WebSocket-protokolla on päivitetty versio HTTP:stä, jotta se voi käyttää samoja portteja kuin HTTP ja HTTPS (80 ja 443). Kättely tehdään HTTP:tä käyttäen, ja se päivittää yhteyden HTTP:stä WebSocket-protokollaan. Yritysten verkoissa liikenne on usein avattu ainoastaan harvoille ja valituille porteille. Yhteys saattaa myös pysähtyä palomuuriin. Näin jo valmiiksi tunnetuilla porteilla toimimisesta on huomattava etu. Lisäksi kättely on täysin taaksepäin yhteensopiva, joten palvelin ymmärtää asiakasohjelman pyynnön, vaikka WebSocket-yhteyden tarjoaminen ei olisikaan mahdollista. [21; 23.]



Kuva 4. WebSocket-yhteyden eri vaiheet [24].

Asiakasohjelma lähettää kättelyn yhteydessä palvelimelle avaimen, jota palvelin käyttää toisen avaimen muodostamiseen. Palvelin luo avaimen lisäämällä asiakasohjelman avaimeen tietyn merkkijonon ja tiivistämällä muodostuneen merkkijonon SHA1-algoritmilla (Secure hash algorithm 1). Tämän jälkeen palvelin koodaa tiivistelmän vielä Base64-muotoon ja lisää kättelyvastauksen otsikkoriviksi. Asiakasohjelma tutkii tämän otsikkorivin. Jos rivin arvo ei ole odotettu tai se puuttuu kokonaan, asiakasohjelma tulkitsee, että yhteyden päivittäminen WebSocket-yhteydeksi ei ole onnistunut. [21; 23.]

Kun WebSocket-yhteys on muodostettu, viestien lähettäminen molempiin suuntiin voi alkaa. WebSocket-protokolla perustuu viesteihin, toisin kuin TCP (Transmission control protocol), joka perustuu bittijonoihin. TCP:tä käyttävä sovellus lähettää bittijonon, jossa täytyy olla selkeä merkki siitä, missä datansiirto päättyy. WebSocket-protokolla yksinkertaistaa tätä lisäämällä kehyksen jokaisen datapalasen ympärille. [21; 23.] Kokonainen viesti jaotellaan yhdeksi tai useammaksi datapalaseksi riippuen viestin pituudesta. Jaottelua voidaan kutsua myös sirpaloinniksi, ja yksittäinen sirpale tarkoittaa yhtä datapalasta. Vastaanottaja saa tapahtumatyyppisen ilmoituksen, kun koko viesti on saatavilla. Viestin jakaminen palasiin ja yhdistäminen tapahtuu asiakasohjelman ja palvelimen kehystämiskoodissa ja on käytännössä piilossa web-sovelluksen kehittäjältä. Websovelluksen kehittäjän ei siis tarvitse ymmärtää kehystyksen toimintaperiaatetta WebSocket-teknologiassa. Jos ollaan kuitenkin kehittämässä esimerkiksi omaa verkkoselainta tai WebSocket-palvelinkirjastoa, tiedonsiirto on kehitettävä itse spesifikaation mukaiseksi. [25.]

Molemmat osapuolet voivat sulkea yhteyden milloin tahansa. Yhteyden sulkeminen tapahtuu lähettämällä tietyntyyppinen kehys toiselle osapuolelle. Asiakasohjelma antaa palvelimelle aikaa yhteyden sulkemiseen, ennen kuin se yrittää sitä itse. Yhteyden sulkeminen muistuttaa hyvin paljon kättelyä, jolla yhteys muodostettiin. Sulkemistapahtuma lähetetään asiakasohjelmalle, kun yhteys on suljettu. Mukana voidaan lähettää syy yhteyden sulkemiselle. [23.]

Datakehyksellä on otsikkotietoja samaan tapaan kuin esimerkiksi TCP-paketilla. Kehyksen rakenne on esitetty kuvassa 5. Otsikkotiedot sisältävät tärkeää tietoa kehyksen vastaanottajalle. Ensimmäinen bitti (FIN) kertoo, onko kyseessä viestin viimeinen sirpale. On myös mahdollista, että viestissä on ainoastaan yksi kehys. Seuraavat kolme bittiä saavat tällä hetkellä aina arvon 0. Nämä bitit on varattu tulevia protokollamuutoksia varten, eivätkä ne ole tällä hetkellä käytössä. Opcode on neljän bitin kokoinen arvo, joka

kertoo kehyksen tyypin, joita on yhteensä viisi erilaista. Opcode voi saada arvot 1, 2, 8, 9 tai 10. Niiden merkitykset samassa järjestyksessä ovat: tekstimuotoinen data, binäärimuotoinen data, yhteyden sulkeminen ja yhteystestit ping ja pong. Kolme viimeistä ovat niin kutsuttuja kontrollikehyksiä. [21; 23; 25.]

<table>
<tr><th>Bit</th><th colspan="3">+0..7</th><th colspan="2">+8..15</th><th>+16..23</th><th>+24..31</th></tr>
<tr><td>0</td><td>FIN</td><td></td><td>Opcode</td><td>Mask</td><td>Length</td><td colspan="2">Extended length (0–8 bytes) ...</td></tr>
<tr><td>32</td><td colspan="7">...</td></tr>
<tr><td>64</td><td colspan="5">...</td><td colspan="2">Masking key (0–4 bytes) ...</td></tr>
<tr><td>96</td><td colspan="5">...</td><td colspan="2">Payload ...</td></tr>
<tr><td>...</td><td colspan="7">...</td></tr>
</table>

Kuva 5. WebSocket-datakehyksen rakenne [25].

Opcoden jälkeen seuraava yksittäinen bitti kertoo, onko kehyksen tietosisältö maskeerattu vai ei. Sitä seuraa kehyksen pituuden kertova arvo, jolle on varattu tilaa 7 bittiä. Jos pituusarvo on väliltä 0–125, se tarkoittaa myös suoraan kehyksen pituutta. Jos pituusarvo on 126, seuraavat 16 bittiä kertovat kehyksen pituuden, ja vastaavasti, jos se on 127, seuraavat 64 bittiä kertovat pituuden. Maskiavaimella saadaan alkuperäinen tieto esiin maskeeratusta tietosisällöstä. Maskiavaimelle on varattu tilaa 32 bittiä. Loppuosa kehyksestä sisältää kehyksen tietosisällön. [21; 23; 25.]

Hyvä web-sovellus pitää huolen siitä, että yhteys säilyy toimivana myös silloin, kun asiakasohjelman ja palvelimen välillä ei ole aktiivista tiedonsiirtoa. Tämän voi ratkaista kehittämällä sydämenlyöntijärjestelmän. Järjestelmä tarkoittaa käytännössä pienien vastaanottovarmistusta pyytävien viestien vaihtamista osapuolien välillä. Välityspalvelimet ja sisällönsuodatuslaitteistot saattavat sulkea toimettomat yhteydet. Palvelin voi myös kaatua ilman, että asiakasohjelma on siitä tietoinen. Asiakasohjelma tietäisi tästä vasta, kun se yrittää seuraavan kerran lähettää dataa palvelimelle. Sydämenlyöntijärjestelmän avulla yhteys ei vaikuta toimettomalta, joten yhteyden säilyminen avoinna on todennäköisempää. Järjestelmän voi toteuttaa käyttämällä ping- ja pong-kehyksiä. [23.]

3.2.4 Server Sent Events -teknologia

Server Sent Events (SSE) on WebSocket API:n tapaan osa HTML5-kieltä. Joskus SSE:tä kutsutaan myös nimellä EventSource. SSE tarjoaa samanlaisen toiminnallisuuden kuin long polling, mutta ilman kaikkia long polling -tekniikan huonoja puolia. Merkittävimmät eroavaisuudet SSE:n ja WebSocket-teknologian välillä ovat SSE:n yksisuuntainen kommunikaatio ja se, että SSE käyttää HTTP-protokollaa. Koska HTTP on yleisesti käytössä oleva protokolla, SSE ei tarvitse mitään erityistä palvelinpuolen toteutusta. Teknologia on kätevä sellaisissa sovelluksissa, joissa reaaliaikaisia päivityksiä tarvitaan ainoastaan palvelimen suunnasta asiakasohjelmalle. Asiakasohjelma voi tarvittaessa lähettää palvelimelle dataa tai pyyntöjä kuten tavallisestikin POST-metodin tai esimerkiksi AJAX-pyynnön avulla. [16, s. 31; 29.]

Tavallisesti selaimen täytyy tukea SSE:tä, jotta sitä voidaan käyttää. Koska teknologia käyttää HTTP:tä, SSE:n ohjelmointirajapinnan voi kuitenkin toteuttaa myös puhtaalla JavaScriptillä. JavaScriptillä toteutetun rajapinnan avulla teknologia toimii hyvin vanhoillakin selaimilla, kuten Internet Explorerin versiolla 8. Tämän vuoksi SSE saavuttaa huomattavasti laajemman tuen kuin WebSocket-teknologia, jonka rajapinnan toteutuksen täytyy olla osa selainta erillisen protokollan vuoksi. [29; 30.]

SSE:n käyttäminen asiakaspuolella on melko helppoa. Koodiesimerkissä 1 on esitetty liittyminen SSE:n tapahtumavirtaan ja tapahtumakuuntelijoiden määrittely yhteyden avaamista, yhteyden sulkemista sekä viestien vastaanottamista varten. Selaimen tuki SSE:lle voidaan tarkastaa kokeilemalla, onko EventSource-objekti määritelty. Tapahtumavirtaan liitytään luomalla EventSource-objekti ja antamalla sille parametrina osoite, josta viestit lähtevät. Jos osoite sijaitsee jonkin toisen verkkotunnuksen alla kuin sivu, jolta yhteyttä ollaan muodostamassa, osoitteen lisäksi täytyy antaa asetustaulukko. Asetustaulukossa täytyy määritellä withCredentials-muuttujan arvoksi true. [29.] Tämä asetus ottaa käyttöön CORS-mekanismin, joka sallii JavaScript-pohjaiset HTTP-pyynnöt eri verkkotunnusten välillä. Kaikki selaimet eivät välttämättä tue CORS-mekanismia (Cross-origin resource sharing). Skripteissä suoritetut verkkotunnusten väliset HTTP-pyynnöt on rajoitettu selaimissa turvallisuussyistä. [31.]

```
if (!!window.EventSource) {
  var source = new EventSource('stream.php');
}

source.addEventListener('message', function(e) {
  // e.data sisältää viestin palvelimelta
  console.log(e.data);
}, false);

source.addEventListener('open', function(e) {
  // Yhteys avattu
}, false);

source.addEventListener('error', function(e) {
  if (e.readyState == EventSource.CLOSED) {
    // Yhteys suljettiin.
  }
}, false);
```

Koodiesimerkki 1. Tapahtumavirtaan liittyminen ja tapahtumakuuntelijoiden määrittäminen asiakaspuolen JavaScript-koodissa [29].

Tapahtumakuuntelijat määritellään JavaScriptissä tuttuun tapaan addEventListener-metodilla. Parametreina annetaan tapahtuman nimi ja funktio, joka suoritetaan, kun uusi tapahtuma vastaanotetaan. Kuuntelijat yhteyden avaamiselle ja virheille eivät ole pakollisia, mutta saattavat olla hyödyllisiä tapauksesta riippuen. Sovelluksen kehittäjän ei tarvitse huolehtia uudelleenyhdistämisestä, jos yhteys katkeaa. Selain yrittää yhdistää automaattisesti uudelleen oletuksena 3 000 millisekunnin väliajoin yhteyden katkeamisen jälkeen. Palvelin voi säädellä uudelleenyhdistämisen intervallia lähettämällä tietynlaisen viestin selaimelle. [29.]

Tapahtumavirran formaatti on hyvin yksinkertainen. Palvelimelta tulevassa vastauksessa täytyy olla Content-Type-otsikkotieto, jonka arvoksi on määritelty *text/event-stream*. Koodiesimerkissä 2 on viisi erilaista esimerkkiä palvelimelta lähetetyistä viesteistä. Ensimmäinen esimerkki on yksinkertaisin viesti, jossa on ainoastaan yksi rivi. Varsinaisen tekstimuotoisen viestin sisältö seuraa aina ”data:”-tekstiä. Viesti päättyy kahteen rivinvaihtomerkkiin. Toisessa esimerkissä esitetään, miten viestin voi jakaa useammalle riville. Rivit jaetaan käyttämällä yhtä rivinvaihtomerkkiä, ja jokainen viestirivi alkaa ”data:”-tekstillä. [29; 32.]

```
data: Ensimmäinen viestini\n\n

data: Ensimmäinen rivi\n
data: Toinen rivi\n\n

data: {\n
data: "msg": "Hello World!",\n
data: "id": 12345\n
data: }\n\n

retry: 10000\n\n

id: 12345\n
event: update\n
data: {"username": "John123", "status": "offline"}\n\n
```

Koodiesimerkki 2. Tapahtumavirran rakenne [29].

Usean rivin viestiä voi hyödyntää myös esimerkiksi JSON-muotoisen vastauksen muodostamiseen, kuten kolmannessa esimerkissä on tehty. Neljäs esimerkki säätelee automaattisen uudelleenyhdistämisen intervallin millisekunneissa. Selain yrittäisi yhdistää 10 sekunnin välein vastaanotettuaan esimerkin viestin, jos yhteys katkeaisi. Viimeisessä esimerkissä viestille on asetettu tunnisteluku (id) ja tapahtumalle on annettu nimi ”update”. Tapahtuman nimellä voi säädellä mille kuuntelijalle viesti välitetään. Asiakaspuolen sovellus voi siis reagoida eri tavalla erinimisiin viesteihin. Tunnisteluvun avulla selain tietää, mikä on viimeisin viesti, jonka se on vastaanottanut. Jos yhteys katkeaa, selain asettaa Last-Event-ID-otsikkotiedon, kun se yrittää muodostaa uutta yhteyttä palvelimeen. Palvelin voi tämän tiedon avulla lähettää kaikki tunnisteluvun jälkeiset tapahtumat selaimelle, kun uusi yhteys on muodostettu. [29; 32.]

3.2.5 WebRTC-teknologia

WebRTC on avoimen lähdekoodin projekti, jonka tukijoita ovat Google, Mozilla ja Opera. W3C kehittää WebRTC API:a, ja IETF (Internet Engineering Task Force) kehittää tarvittavia protokollia. WebRTC:n ideana on tarjota reaaliaikainen kommunikaatio selaimessa ilman mitään lisäohjelmia. Teknologia tarjoaa kommunikaation teksti-, ääni- ja videomuodossa. Myös tiedostojen siirtäminen on mahdollista. Erikoista teknologiassa on, että sillä voidaan muodostaa yhteys kahden eri selaimen välille. Esimerkkinä voidaan käyttää vaikkapa videopuhelusovellusta. Kun yhteys on luotu, selaimet lähettävät videon datavirtana toisilleen ilman, että data kulkee erillisen palvelimen kautta. WebRTC käyttää tiedonsiirtoon vertaisverkkoa. [33; 34.]

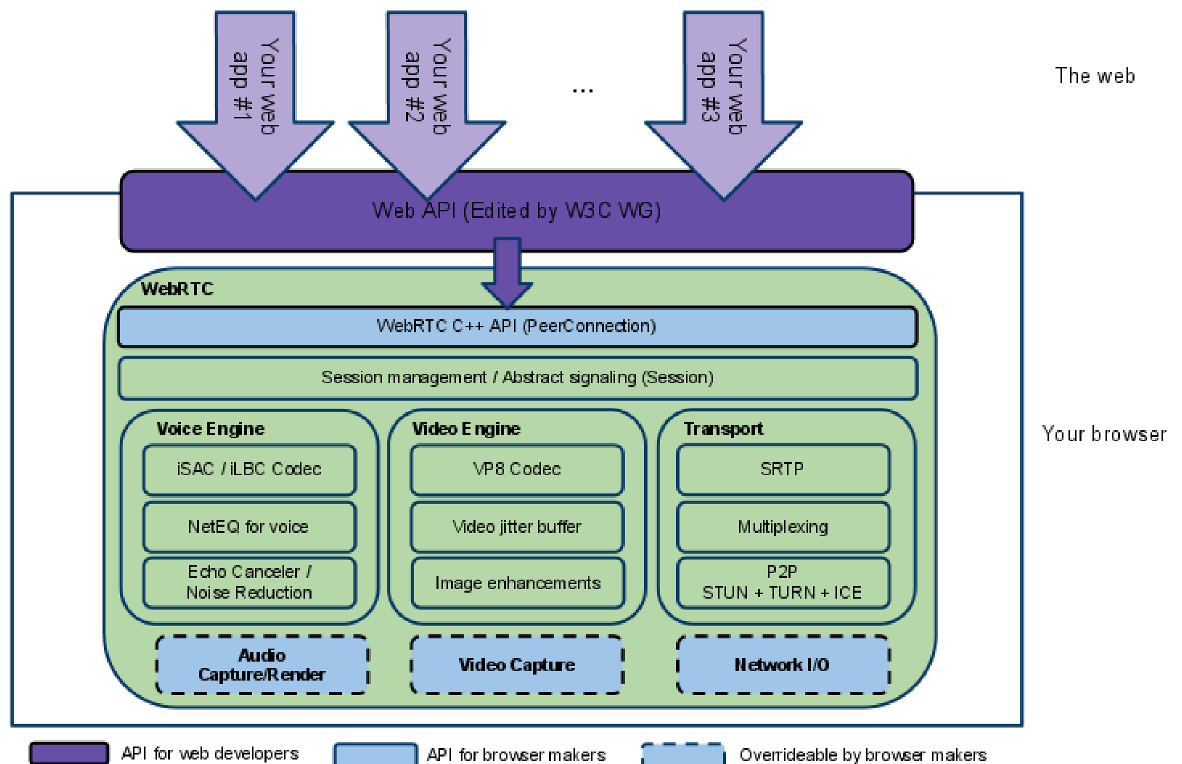
WebRTC on hyvin nuori teknologia, joten sen ohjelmointirajapinta saattaa vielä muuttua. Teoriassa on mahdollista, että teknologiaa käyttävä web-sovellus ei toimikaan enää ollenkaan tai halutulla tavalla selainpäivityksen jälkeen. Vain uusimmat Mozillan, Googlen ja Operan selaimet tukevat sitä, ja niissäkin teknologian uusimmat ominaisuudet vaativat kokeellisten toimintojen aktivoimista selaimen asetuksissa. Myös Microsoft on kehittänyt joitakin WebRTC:n toimintoja uuteen Edge-selaimeen. [33; 35.]

Osa WebRTC:n käyttämistä teknologioista ei ole uusia. Google on käyttänyt niitä jo vuodesta 2008 asti, kun Gmail sai videopuhelutoiminnon ja myöhemmin Google julkaisi Hangouts-viestintäsovelluksen vuonna 2011. Historiassa reaaliaikainen kommunikaatio on vaatinut kalliiden ääni- ja videoteknologioiden lisensoimista tai kehittämistä. Teknologioiden käyttöönotto on ollut vaikeaa erityisesti web-sovelluksissa. WebRTC tarjoaa nämä teknologiat avoimina standardeina ilmaiseen käyttöön. Teknologialle on kysyntää, koska aiemmin toteutukset ovat tarvinneet erillisiä sovelluksia tai lisäosia selaimiin. Lisäosien asentaminen ja päivittäminen voi olla hankalaa, ja lisäksi usein käyttäjät eivät halua edes asentaa lisäosia. [33; 35.]

Vaikka selainten välille voidaan muodostaa suora yhteys, teknologia tarvitsee myös palvelimen. Palvelinta tarvitaan kommunikaation koordinoimiseen ja lähettämään kontrolliviestejä. Tätä toimintaa kutsutaan signaloinniksi. Signalointi ei ole varsinaisesti osa WebRTC API:a. Sovelluskehittäjät voivat valita minkä tahansa kaksisuuntaisen kommunikointiprotokollan signaloinnin toteuttamiseen. Tällaisia protokollia ovat esimerkiksi SIP (Session Initiation Protocol) ja XMPP (Extensible Messaging and Presence Protocol). Eräs suosittu vaihtoehto on Socket.io, joka toimii Node.js-palvelimella. [35.]

Signalointia käytetään kolmen erityyppisen tiedon vaihtamiseen selainten välillä. Istunnonhallintaviesteillä alustetaan tai suljetaan kommunikaatioväylä ja raportoidaan virheistä. Verkkoasetusviesteillä välitetään selainten ip-osoitteet ja portit, joita tarvitaan yhteyden muodostamiseen. Media-asetusviesteillä kerrotaan, mitä koodekkeja ja resoluutioita selain tukee, jotta voidaan valita molemmissa päissä toimivat tekniikat. Signaloinnilla vaihdettu tieto täytyy suorittaa onnistuneesti, ennen kuin datavirta vertaisverkon välityksellä voi alkaa. [35.]

Kuvassa 6 on esitelty WebRTC:n arkkitehtuurin kuvaaja. RTCPeerConnection on WebRTC:n komponentti, joka mahdollistaa vakaan ja tehokkaan kommunikaation selainten välillä. Vihreällä pohjalla olevat osiot ovat hyvin monimutkaisia. Sinisellä pohjalla on merkitty selainkehittäjille tärkeät osat.



Kuva 6. WebRTC:n arkkitehtuuri [35].

Kuvasta 6 voi huomata, että RTCPeerConnection suojaa web-kehittäjät kaikelta monimutkaisuudelta, jota teknologiaan liittyy. WebRTC:n käyttämät koodekit ja protokollat tekevät taustalla paljon työtä, jotta reaaliaikainen kommunikaatio on mahdollista jopa silloin, kun yhteys on epäluotettava. Taustalla tapahtuu muun muassa kaiun- ja kohinanpoistoa, videon virheenkorjausta, kuvanparantelua, mukautumista siirtonopeuksien muutoksiin ja pakettihäviöön. [35.]

4 Reaaliaikaisen webin työkalut

Tässä luvussa esitellään joitakin reaaliaikaisten web-sovellusten kehitykseen soveltuvia kirjastoja ja ohjelmistoja. Esitellyt työkalut eivät ole välttämättä parhaita vaihtoehtoja

kaikkiin ongelmiin. Lisäksi luvussa ei missään tapauksessa esitellä kaikkia aihealueeseen liittyviä työkaluja, vaan niitä on olemassa paljon enemmän. Olen valinnut listaan sellaisia työkaluja, joiden hyödyntämistä harkitsin tässä insinöörityössä.

4.1 Ratchet-WebSocket-kirjasto

Ratchet on hyvin dokumentoitu ja ylläpidetty WebSocket-kirjasto PHP-ohjelmointikielelle. Sen käytössä on nopea ja helppo päästä alkuun esimerkkien avulla. Ratchet on täysin yhteensopiva WebSocket-standardien kanssa binäärimuotoisia viestejä lukuun ottamatta, joten se toimii kaikilla WebSocket-teknologiaa tukevilla selaimilla. Kirjasto tarjoaa useita eri komponentteja, joiden joukosta voi valita omaan sovellukseen tarvittavat osat. Sovellukseen voi myös helposti lisätä uusia toimintoja ottamalla käyttöön uusia komponentteja. [26.] Taulukossa 1 on esitetty Ratchetin sisältämät komponentit ja niiden selitteet.

Taulukko 1. Ratchet-kirjaston sisältämät komponentit [36].

Komponentti	**Selite**
WsServer	WsServer on WebSocket-palvelimen toteutus, joka mahdollistaa kommunikoinnin WebSocket API:a tukevien selainten kanssa.
WampServer	WampServer lisää tuen WAMP-protokollalle, joka helpottaa asiakasohjelman ja palvelimen välistä kommunikaatiota.
SessionProvider	SessionProvider-komponentin avulla palvelin saa vain luku -oikeuden asiakkaan istuntodataan.
IoServer	IoServer on sovelluksen perusta. Se huolehtii uusien yhteyksien vastaanottamisesta, yhteyksien sulkemisesta, virheenhallinnasta ja yhteyksiin kirjoittamisesta / lukemisesta.
HttpServer	HttpServer parsii HTTP-pyyntöjä. Sen päätehtävä on päivittää HTTP-protokolla WebSocket-protokollaan.
Router	Router mahdollistaa usean WebSocket-sovelluksen toimimisen samalla palvelimella.
OriginCheck	OriginCheck lisää sovelluksen turvallisuutta. Se sallii yhteyden muodostamisen vain määritellyiltä sivustoilta.
FlashPolicy	FlashPolicy lisää tuen Flash Socket -tekniikalle, jonka avulla myös selaimet, jotka eivät tue WebSocket-teknologiaa, voivat yhdisää Ratchet-sovellukseen. Flash Socket ei toimi mobiiliselaimissa.
IpBlackList	IpBlackList estää yhteydet määritetyistä IP-osoitteista.

Ratchet käyttää kolmea muuta kirjastoa, minkä vuoksi asentaminen kannattaa tehdä Composer-riippuvuuksienhallintasovelluksen avulla. Composer on asennettavissa sekä Unix- että Windows-ympäristöissä. Composer luo riippuvuuksille oman kansion ja yhden PHP-skriptin, joka liittää kaikki tarvittavat kirjastot sovellukseen. Composer tarvitsee ke-

hitettävää sovellusta varten asetustiedoston, jonka tietojen perusteella se lisää riippuvuudet projektiin. Koodiesimerkissä 3 JSON-muotoisen asetustiedoston malli. Tällaista asetustiedostoa käyttämällä Composer asentaa Ratchet-kirjaston vendor-nimiseen kansioon, jonka se luo projektin juureen, ja sovelluksen omat skriptit sijoitetaan src-kansion sisällä olevaan MyApp-kansioon. Riippuvuudet on määritelty ”require”-osassa. [37; 38.]

```
{
    "autoload": {
        "psr-0": {
            "MyApp": "src"
        }
    },
    "require": {
        "cboden/ratchet": "0.3.*"
    }
}
```

Koodiesimerkki 3. Esimerkki JSON-asennustiedostosta Composer-riippuvuuksienhallintasovellusta varten [37].

Yksinkertaisissa WebSocket-sovelluksissa voi käyttää Ratchetin verkkosivuilta löytyvää esimerkkiä [38] hyvänä pohjana palvelinpuolen koodille. Tärkein kohta sovelluskehittäjän kannalta ovat tapahtumafunktiot, jotka käsittelevät viestien vastaanottamisen ja välittämisen. Koodiesimerkissä 4 on esitetty yksinkertaisen tekstipohjaisen keskustelusovelluksen tapahtumafunktiot. Esimerkissä sovellus kerää kaikki yhdistäneet asiakasohjelmat taulukkomuuttujaan. Kun palvelin vastaanottaa asiakasohjelmalta viestin, se välitetään kaikille muille paitsi lähettäjälle itselleen.

```
public function onOpen(ConnectionInterface $conn) {
  $this->clients->attach($conn);
}
public function onMessage(ConnectionInterface $from, $msg)
{
  foreach ($this->clients as $client) {
      if ($from !== $client) {
          $client->send($msg);
      }
  }
}

public function onClose(ConnectionInterface $conn) {
  $this->clients->detach($conn);
}
```

Koodiesimerkki 4. Chat-sovelluksen tapahtumafunktiot [36].

Suurin osa sovelluksen toiminnallisuuden kannalta tärkeistä ominaisuuksista tapahtuu tapahtumafunktioissa. Funktion sisällä voi tapahtua vaikkapa tietokannasta lukemista tai tietokantaan tallentamista. Samalla voidaan esimerkiksi validoida saapuneen viestin sisältö ja määritellä mille kaikille asiakasohjelmille viesti välitetään eteenpäin tai tallentaa validointivirheet lokitiedostoon.

4.2 Node.js-palvelin

Node.js on avoimen lähdekoodin asynkroninen ja tapahtumapainotteinen JavaScript-suoritusympäristö. Sen sydämenä on Googlen V8-JavaScript-moottori. Node.js pystyy käsittelemään useita yhteyksiä samanaikaisesti, mutta se tekee sen kuitenkin eri tavalla kuin suurin osa muista verkkosovelluksista. Tavanomaisesti rinnakkaisten yhteyksien hallinta on toteutettu avaamalla jokaiselle yhteydelle oma säie käyttöjärjestelmässä. Säikeisiin perustuvat verkkosovellukset ovat melko tehottomia ja vaikeita kehittää. Säikeiden kanssa ajautuu helposti ongelmiin, koska eri säikeet saattavat esimerkiksi muokata samaa resurssia samanaikaisesti. Tämä saattaa aiheuttaa hyvin omituisia virheitä sovelluksen toiminnassa. [39.]

Node.js käyttää säikeiden sijasta tapahtumasilmukkaa. Kun sovellukseen avataan uusi yhteys, Node.js kutsuu tapahtumalle määritettyä tapahtumakuuntelijaa. Tapahtumakuuntelija voi esimerkiksi tarjota asiakkaalle HTML-tiedoston ja olla näin osa HTTP-palvelinta. Kun tapahtumakuuntelija on suorittanut tehtävänsä ja muuta suoritettavaa ei ole jäljellä, Node.js vaipuu lepotilaan ja jää odottamaan uusia tapahtumia. Node.js ei sisällä lähes yhtään sellaista funktiota, joka suorittaisi suoraan kirjoitus- tai lukuoperaatioita, joten prosessi ei koskaan keskeydy tällaisen operaation ajaksi. Esimerkiksi tietokantakyselylle luodaan oma tapahtumakuuntelija, joka suoritetaan vasta, kun tietokantapalvelin vastaa kyselyyn. Node.js voi suorittaa muita toimintoja sillä aikaa, kun tietokantapalvelin prosessoi kyselyä. Asynkronisen ja tapahtumapainotteisen lähestymistavan vuoksi Node.js on tehokas vaihtoehto sovelluksiin, joissa nopea vasteaika ja suuri määrä samanaikaisia yhteyksiä ovat keskeisessä roolissa. [39.]

Modulaarisuus on yksi Node.js:n merkittävimmistä ominaisuuksista. Oma sovellus on helppo jakaa moduuleihin, jotka ovat mahdollisimman pieniä ja itsenäisesti toimivia kokonaisuuksia. Moduuleiden käyttäminen auttaa pitämään projektin koodikannan selkeänä ja helpottaa koodin uudelleenkäyttöä. Npm-paketinhallintajärjestelmän avulla voi

helposti ladata ja ottaa käyttöön muiden tekemiä paketteja. Myös omien pakettien julkaiseminen on mahdollista. Maaliskuussa 2016 ladattavien pakettien määrä rikkoi 250 000 rajan [42]. Npm:n helppokäyttöisyyden katsotaan olevan suuri syy Node.js:n kasvavalle suosiolle. [40; 41.]

4.3 Socket.io-kirjasto

Socket.io tarjoaa kaksisuuntaisen ja reaaliaikaisen kommunikointiyhteyden web-sovelluksille. Sen sekä asiakas- että palvelinpuolen ohjelmointirajapinnat ovat lähes identtisiä WebSocket-teknologian rajapintojen kanssa. Socket.io yhdistää monta eri protokollaa ja tekniikkaa yhden rajapinnan alle. Tuetut tekniikat ovat WebSocket, XHR-polling, flash-socket, jsonp-polling ja htmlfile [44]. Se osaa myös valita parhaan mahdollisen siirtotavan, joka on tuettu asiakasohjelmassa. Myös yhteyden ylläpito ja uudelleen yhdistäminen ovat automaattisia toimintoja. Socket.io on laajasti tuettu eri selaimissa, koska se yhdistää niin monta eri siirtotekniikkaa. [43.]

Monimutkaiset toiminnot on piilotettu hyvin ohjelmointirajapintojen taakse, joiden avulla reaaliaikaisen web-sovelluksen kehittäminen on helppoa. Koodiesimerkistä 5 voidaan havaita, miten yksinkertainen pienen chat-sovelluksen palvelinpuolen koodi voi olla. Esimerkissä näkyy myös JavaScript-ohjelmoinnille ja WebSocket-teknologialle tyypillisiä tapahtumakuuntelijoita.

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

io.on('connection', function(socket){

  socket.on('message', function(msg) {
    socket.broadcast.emit('message', msg);
  });

  socket.on('disconnect', function() {
    console.log('user disconnected');
  });
});

http.listen(3000, function(){
  console.log('listening on *:3000');
});
```

Koodiesimerkki 5. Yksinkertainen pohja chat-sovellukselle Node.js:n ja Socket.io:n avulla [45].

Koodiesimerkin 5 sovellus pystyy vastaanottamaan yhteyksiä ja viestejä, jotka välitetään muille asiakkaille. Koodia voisi itse asiassa vieläkin lyhentää, koska kaikki kutsut ”console.log”-funktioon ja ”disconnect”-tapahtumakuuntelija ovat tarpeettomia. Esimerkki on kuitenkin hyvin yksinkertainen, ja todellisessa chat-sovelluksessa olisi varmasti enemmän toimintoja.

4.4 AutobahnJS-WAMP-rajapinta

AutobahnJS on alaprojekti Autobahn-projektille, joka tarjoaa avoimen lähdekoodin toteutuksen WAMP-protokollasta (Web Application Messaging Protocol). Autobahn on julkaistu useille eri ohjelmointikielille, ja AutobahnJS on sen JavaScript-versio. WAMP mahdollistaa asynkroniset kutsut etäproseduureihin ja tilaus-julkaisumallin käytön WebSocket-teknologian kanssa. Tilaus-julkaisumallissa asiakasohjelma tilaa jonkin aiheen, ja palvelin julkaisee viestin kaikille tilauksen tehneille asiakkaille heti, kun uusi viesti on saatavissa. Etäproseduuri on jonkin toisen laitteen sovelluksessa oleva toiminto, jonka voi suorittaa etänä WebSocket-yhteyden kautta. [28.]

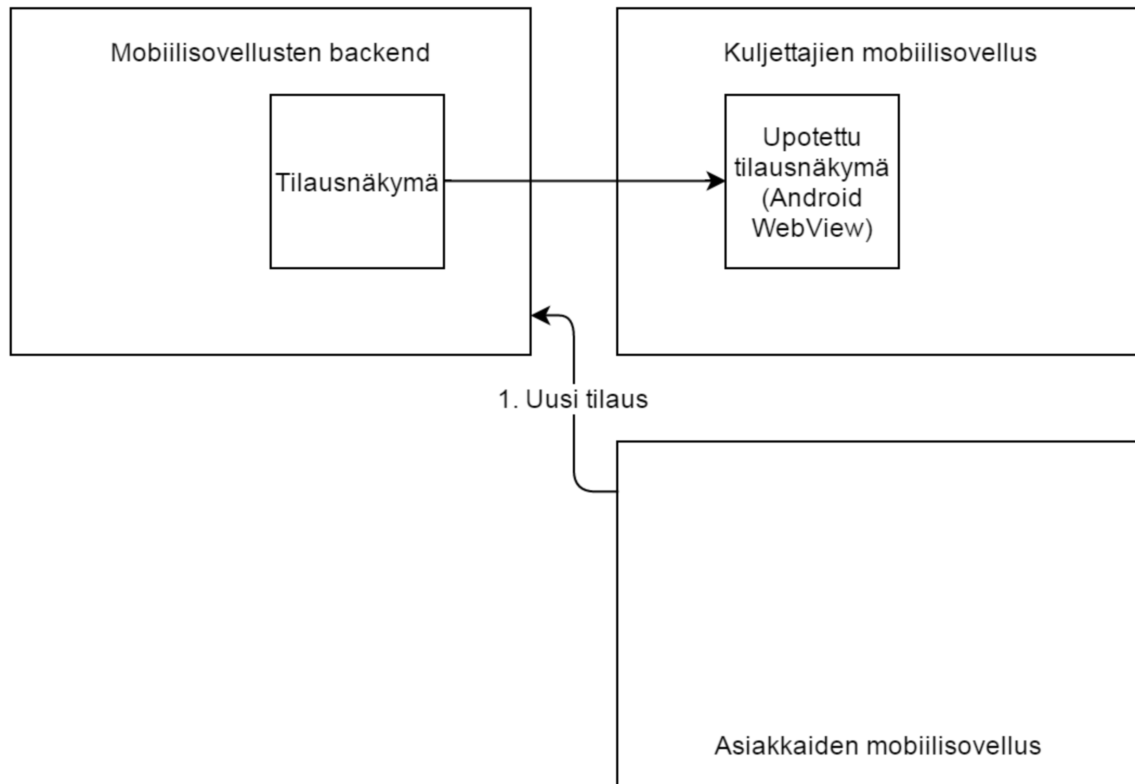
WAMP on kätevä sellaisissa sovelluksissa, joissa palvelin työntää asiakasohjelmille uuden informaation heti, kun se saapuu palvelimelle. Esimerkkinä tällaisesta sovelluksesta voi olla blogisivusto, jossa ilmoitus uudesta blogikirjoituksesta välitetään kaikille sivustolla vieraileville asiakkaille välittömästi, kun kirjoitus julkaistaan. Etäproseduurien avulla sovelluksen koodi voidaan jakaa useille eri prosesseille ja laitteille. [28.]

5 Tilausnäkymän suunnittelu ja toteutus

5.1 Suunnittelu

Taksintilausjärjestelmä koostuu useasta eri osasta. Kuvassa 7 on esitetty järjestelmän eri osat lähtötilanteessa. Kuva keskittyy tilausnäkymän kannalta merkittäviin seikkoihin, joten siinä ei ole kuvattu täydellisesti kaikkia eri osien sisältämiä toimintoja. Tilausnäkymä on verkkosivu, joka on itse asiassa osa mobiilisovellusten palvelinpuolta. Palvelinpuolen päätehtävä on tallentaa ja toimittaa mobiilisovellusten käyttämä data rajapintojen avulla, mutta se sisältää myös muutamia verkkosivumuotoisia näkymiä. Palvelinpuoli

suorittaa näiden näkymien muodossa osittain sellaisia tehtäviä, jotka kuuluisivat tavallisesti mobiilisovelluksille. Näkymät ovat suorassa käytössä mobiilisovelluksissa upotettuna käyttöliittymään. Tämä onnistuu esimerkiksi Android-sovelluksissa käyttämällä WebView-komponenttia, joka on sovellukseen upotettu Android-käyttöjärjestelmän verkkoselain. Tällaiseen komponenttiin voidaan avata palvelinpuolen tarjoama verkkosivu.



Kuva 7. Taksintilausjärjestelmän eri osat lähtötilanteessa.

Tilausnäkymän lisäksi palvelinpuoli tarjoaa myös esimerkiksi asiakkaiden mobiilisovellukselle maksukorttientallennusnäkymän, joka on upotettu samalla tavalla mobiilisovellukseen. Asiakkaiden sovellusta käytetään luonnollisesti uusien tilausten tekemiseen. Tämän lisäksi asiakkaat voivat hallita sovelluksella omia tietojaan ja tallentaa maksukortteja, jotta maksaminen onnistuu jatkossa vaivattomasti. Tilaukset tehdään joko valitsemalla osoite automaattisesti GPS:n avulla tai syöttämällä se manuaalisesti.

Kuljettajien mobiilisovellus on hieman monipuolisempi kuin asiakkaiden sovellus. Ensimmäiseksi kuljettajat aloittavat työvuoronsa sovelluksessa. Tämän jälkeen he avaavat tilausnäkymän ja odottavat uusia tilauksia tai hyväksyvät itselleen jo jonkin hyväksymistä odottavan tilauksen. Kun asiakas otetaan kyytiin, tilaus otetaan ajoon ja sovellus siirtyy taksimittarinäkymään. Mittari laskee matkalle hinnan, ja ajon päätyttyä sovellus avaa

upotetun maksusivun, jossa voidaan syöttää uusi maksukortti, valita jokin asiakkaiden sovelluksessa tallennettu maksukortti tai valita esimerkiksi käteismaksu.

Lähtötilanteessa tilausnäkymä käytti polling-tekniikkaa tilauslistan päivittämiseen. Polling-tekniikka oli toteutettu yksinkertaisella AJAX-kyselyllä, joka suoritettiin 15 sekunnin väliajoin JavaScriptin setInterval-funktiolla. Näin näkymään saatiin jossain määrin reaaliaikainen vaikutelma. Toteutuksessa oli kuitenkin useita eri ongelmia, joista osa vaikutti skaalautuvuuteen ja osa suoraan näkymän tarjoamaan käyttökokemukseen.

Polling-tekniikan vaikutus käyttökokemukseen oli selkeä. Ensinnäkin AJAX-kyselyllä noudettiin kaikki tilaukset, jotka odottivat kuljettajien hyväksyntää. Tämän jälkeen tilauslista tyhjennettiin kokonaan ja uusi lista muodostettiin AJAX-kyselyn tuloksista. Uuden listan muodostaminen ei ollut tarpeeksi nopeaa, ja tyhjeneminen oli selkeästi havaittavissa. Jos listassa oli enemmän tilauksia kuin näytölle mahtuu, oli hankala arvioida, kuinka monta uutta tilausta päivityksessä tuli. Toinen käytettävyyteen vaikuttava ongelma oli päivitysten välinen pitkä intervalli. Kuljettajien hyväksymät tilaukset katosivat listalta vasta päivityksen yhteydessä, joten 15 sekunnin pituinen intervalli oli erittäin epäkäytännöllinen. Kuljettajat yrittäisivät jatkuvasti hyväksyä itselleen tilausta, jonka joku muu on jo ehtinyt hyväksyä.

Uusilla tilauksilla on erilaisia prioriteettiarvoja. Tilauksen prioriteetti vaikuttaa siihen, minkälaista työvuoroa ajaville kuljettajille tilaus näytetään. Näistä prioriteeteista ensimmäinen näkyy vain tietynlaisille työvuoroille. Tämä prioriteetti myös vaihtuu jo 30 sekunnin kuluttua tilauksen vastaanottamisesta. Päivitysintervallin takia tilauksen vastaanottohetkellä oli erittäin suuri vaikutus siihen, kuinka kauan tilaus näkyi tälle tietylle työvuororyhmälle, ennen kuin prioriteetti vaihtui. Jos tilaus vastaanotettiin heti edellisen AJAX-kyselyn jälkeen, tilaus näkyi kuljettajalle vain puolet ideaalisesta ajasta. Jos tilaus sen sijaan vastaanotettiin juuri ennen AJAX-kyselyä, tilaus näkyi kuljettajalle lähes koko 30 sekunnin ajan, ennen kuin se näytettiin myös muita työvuoroja ajaville kuljettajille.

Lähtötilanteen ratkaisu oli hyvin huonosti skaalautuva, koska AJAX-kyselyjä tehtiin toistuvasti, vaikka uusia tilauksia ei olisikaan ollut saatavilla. Lisäksi jokainen AJAX-haku haki kaikki hyväksyntää odottavat tilaukset mukaan lukien ne, jotka oli jo näytetty kuljettajalle. Todennäköisesti päivitysintervallia olisi täytynyt myös lyhentää, jotta tilausnäkymästä olisi saatu paremmin käytettävä. Tämä olisi heikentänyt skaalautuvuutta entisestään. Toki polling-tekniikalla toimivaa ratkaisua olisi voinut myös parannella niin, ettei se

hakisi aina kaikkia hyväksyntää odottavia tilauksia. Tällöin listalta poistettavat tilaukset olisi pitänyt toimittaa vastauksen mukana erikseen. Parannuksen olisi voinut toteuttaa esimerkiksi lähettämällä AJAX-kyselyn vastauksen mukana palvelimen senhetkisen ajan. Tätä aikaa käytettäisiin seuraavan kyselyn parametrina, ja palvelin lähettäisi vastauksena ainoastaan ajan jälkeen vastaanotetut ja hyväksytyt tilaukset.

Parannusten jälkeen myös käyttökokemusta olisi saatu helposti paremmaksi, koska päivityksen yhteydessä listan eteen olisi voitu lisätä uudet tilaukset ja listasta olisi voitu poistaa kaikki jo hyväksytyt tilaukset. Koko listaa ei tarvitsisi enää tyhjentää ja muodostaa uudelleen. Listaan tehtäisiin ainoastaan kaikki tarpeelliset lisäykset ja poistot. Parannukset olisivat tehneet palvelinpuolen toteutuksesta melko monimutkaisen, ja skaalautuvuus ei olisi silti ollut hyvä, koska jokainen tilauslistaa katseleva kuljettaja olisi rasittanut palvelinta toistuvasti päivitysintervallin välein. Jokainen kysely rasittaisi sekä Apache-web-palvelinta että MySQL-tietokantapalvelinta, jota palvelinpuoli käyttää tiedon tallentamiseen.

Koska polling-tekniikalla päivitettävän tilauslistan parantelu ei olisi tuottanut suuria hyötyjä, päätettiin tutkia, saisiko uudemmilla teknologioilla toteutettua tehokkaamman ja käytettävämmän ratkaisun. Parhaat vaihtoehdot olisivat WebSocket- ja Server Sent Events (SSE) -teknologiat, koska ne soveltuvat hyvin malliin, jossa palvelin lähettää verkkosivulle dataa haluamallaan hetkellä. Itse asiassa SSE on suunniteltu juuri tähän tarkoitukseen. WebSocket-teknologialla pystyy tekemään kaiken sen mihin, SSE-teknologia pystyy ja vielä hieman enemmänkin. Päätin valita WebSocketin käytettäväksi teknologiaksi, koska se mahdollistaa kaksisuuntaisen jatkuvan yhteyden palvelimen ja verkkosivun välillä. Ajatuksena oli, että verkkosivulta tarvitsisi lähettää dataa palvelimelle esimerkiksi silloin, kun kuljettaja hyväksyy itselleen tilauksen. Kaksisuuntainen yhteys saattaisi olla täten hyödyllinen. [16, s. 31–37.]

Ainoa huolenaihe WebSocket-teknologiaan liittyen oli se, tukeeko Androidin WebView-elementti WebSocket-teknologiaa. Kokeillakseni teknologian toimivuutta tein pienen Android-sovelluksen, joka avasi WebView-elementtiin internetistä löytämäni WebSocket-teknologian toimivuuden testaamiseen tarkoitetun verkkosivun. Verkkosivu ilmoitti, että selain tukee WebSocket-teknologiaa, joten suunnitelma WebSocketin käyttämisestä jatkui.

Tarjolla oli useita ilmaisia ja avoimen lähdekoodin WebSocket-palvelinkirjastoja useille eri kielille. Järjestelmän palvelinpuoli on toteutettu CakePHP-sovelluskehyksen ympärille, joten halusin valita sellaisen kirjaston, joka käyttää myös PHP-ohjelmointikieltä. Tämän jälkeen oli helppo valita Ratchet, koska se oli ainoa löytämäni hyvin ylläpidetty ja dokumentoitu WebSocket-kirjasto PHP:lle [26].

Ratchetin esimerkeissä suositeltiin käyttämään AutobahnJS-kirjastoa verkkosivun päässä, koska se tukee WAMP-protokollan käyttöä [27]. WAMP-protokolla toteuttaa rakenteen, jossa asiakasohjelma tilaa jonkin aiheen ja palvelin lähettää kaikille aiheen tilanneille asiakasohjelmille uutta dataa heti, kun sitä on saatavissa [28]. Tämä rakenne sopisi hyvin reaaliaikaiseen tilauslistaan. Tilattavia aiheita voisivat olla tilausten eri prioriteetit, joilla kontrolloidaan mille työvuoroille uusi tilaus toimitetaan. Myös kuljettajien hyväksymille tilauksille olisi hyvä tehdä oma aihe, koska tällöin tilausten poistaminen listasta voidaan selkeästi erottaa uusien tilausten lisäämisestä.

Nyt tarvittavat teknologiat ja työkalut oli valittu. Suunnitelmana oli toteuttaa mobiilisovellusten palvelinpuolen rinnalle erillinen WebSocket-palvelin. Palvelinpuoli välittäisi uudet tilaukset heti, kun ne saapuisivat järjestelmään, WebSocket-palvelimelle, joka vuorostaan välittäisi ne kaikille tilausnäkymää parhaillaan seuraaville kuljettajille. Nyt kuljettajat saisivat tietää uusista tilauksista reaaliajassa ja eri osapuolten välillä liikkuva turha data olisi minimoitu.

5.2 Toteutus

5.2.1 Ratchet-kirjasto

Toteutus lähti liikkeelle WebSocket-projektin perustamisella kehitysympäristöön. Kehitysympäristönä oli Windows 7 -käyttöjärjestelmä ja XAMPP, jonka avulla käytettiin Apache-, MySQL- ja PHP-palvelinohjelmistoja. Jotta projektiin saatiin asennettua Ratchet, kehitysympäristöön täytyi asentaa Composer-riippuvuuksienhallintajärjestelmä. Composer oli melko helppo asentaa Windowsiin asennustiedostolla. Polku Composerin exe-tiedostoon täytyi lisätä erikseen Windowsin ympäristömuuttujiin, jotta tiedoston pystyi suorittamaan komentoriviltä ilman tiedostopolun lisäämistä komennon eteen.

Riippuvuuksien asentamiseksi Composerille tehtiin projektin juureen JSON-muotoinen asetustiedosto. Tiedoston sisältö on esitetty koodiesimerkissä 6. Kun tiedosto oli lisätty

projektin juureen, riippuvuudet asennettiin navigoimalla komentorivillä projektikansioon ja suorittamalla komento *composer install*. Composer latasi Ratchet- ja ZeroMQ-kirjaston ja asetti ne ”vendors”-kansioon. ZeroMQ on kirjasto, jolla voi hallita sokettiyhteyksiä, ja sitä käytetään viestien välityksessä CakePHP:n ja WebSocket-palvelimen välillä. Composer loi samalla myös skriptin, jonka avulla asennetut kirjastot voi helposti linkittää omaan PHP-skriptiin.

```
{
  "autoload": {
      "psr-0": {
          "TaxiaWebSocket": "src"
      }
  },
  "require": {
      "cboden/Ratchet": "0.3.*",
      "react/zmq": "0.2.*|0.3.*"
  }
}
```

Koodiesimerkki 6. Asetustiedosto, jossa määritellään projektissa käytettävät riippuvuudet Ratchet ja ZeroMQ.

ZeroMQ:n asentaminen Windows-ympäristöön oli melko hankalaa. Kuten Composer, myös ZeroMQ asennettiin asennustiedostolla. Jotta kirjastoa pystyi käyttämään, PHP:lle piti asentaa kuitenkin ZeroMQ-lisäosa, jonka täytyi olla käännetty samalla Visual C++ -versiolla kuin käytössä oleva PHP. Käytössä ollut PHP-versio oli vanha, joten tällaisen version löytäminen oli haastavaa. Kun sopiva versio lopulta löytyi, se kopioitiin PHP:n ext-kansioon ja viittaus tiedostoon lisättiin PHP:n asetustiedostoon.

Kaikki tarpeelliset työkalut oli asennettu, joten WebSocket-palvelimen ohjelmointi aloitettiin. Koodin pohjana käytettiin Ratchetin tilaus-julkaisumalliin perustuvaa esimerkkiä [27]. Asiakasohjelmat eivät itse lähetä viestejä ollenkaan, vaan ne ainoastaan vastaanottavat palvelimen työntämiä viestejä. Koodiesimerkissä 7 on esitelty WAMP-protokollaa käyttävän WebSocket-palvelimen tärkeimmät tapahtumakuuntelijat. Asiakasohjelmien tilaukset otetaan vastaan ”onSubscribe”-funktiolla, ja aiheet kerätään taulukkomuuttujaan. Kun CakePHP:ltä saapuu uusi viesti, se välitetään ensin ”onIntraMessage”-funktiolla, joka välittää viestin edelleen aiheen tilanneille asiakkaille.

```
protected $subscribedTopics = array();

public function onIntraMessage($message) {

  $messageData = json_decode($message, true);

  // If the lookup topic object isn't set
  // there is no one to publish to
  if(!array_key_exists($messageData['topic'],
      $this->subscribedTopics)) {
      return;
  }

  $topic = $this->subscribedTopics[$messageData['topic']];

  echo "Topic: " . $topic . "\n";
  echo $message . "\n";

  // Send the data to all clients subscribed to that
  // category.
  $topic->broadcast($messageData);
}

public  function  onSubscribe(ConnectionInterface  $conn,
$topic) {
  $this->subscribedTopics[$topic->getId()] = $topic;
  echo "new subscription\n";
}
```

Koodiesimerkki 7. WAMP-palvelimen tärkeimmät tapahtumakuuntelijat.

Sovelluksen asiakaspuoli toteutettiin WAMP-yhteensopivalla AutobahnJS-kirjastolla, jonka käyttöä esitellään koodiesimerkissä 8. AutobahnJS:lle annetaan parametrina WebSocket-palvelimen ip-osoite. Heti yhdistämisen jälkeen asiakkaalle tilataan ”fare_inquiry_canceled”-aihe ja sille määritellään tapahtumakuuntelija. Tapahtumakuuntelija näyttää asiakkaalle ponnahdusikkunan, jossa kerrotaan, että tilaus on peruttu.

```
var conn = new ab.Session('127.0.0.1:5555',
  function() {
    conn.subscribe('fare_inquiry_canceled',
      function(topic, data) {
        if(fi_id == data.msg.id) {
          $("#popupOtsikko").html("Tilaus on peruttu.");
          $(".popupMenu").popup( "open" );
        }
    });
});
```

Koodiesimerkki 8. Yhdistäminen WebSocket-palvelimeen ja aiheen tilaaminen asiakaspuolella.

Kun kaikki tarvittavat aiheet ja niiden toiminnallisuus oli määritelty ja sovellus toimi oikein, tuli aika testata sovellusta oikealla palvelimella. Tarvittavien sovellusten asentaminen Unix-ympäristössä oli helpompaa kuin Windowsissa, koska ne kaikki voitiin asentaa yhdellä yum-komennolla. Yum etsii sovellusta internetin kautta tietolähteistä, joista löytyy yleisimmin käytetyt sovellukset. Riippuvuuksien asentaminen Unix-ympäristössä onnistui samalla tavalla kuin Windowsissakin.

WebSocket-palvelin toimi myös Unix-ympäristössä ilman erityisempiä toimenpiteitä. Se otettiin käyttöön järjestelmän testiversiossa, jota myös kuljettajien mobiilisovellus käytti. Tässä vaiheessa WebSocket-yhteyttä päästiin kokeilemaan ensimmäisen kerran mobiilisovelluksessa. Pettymykseksi yhteys ei toiminutkaan Androidin WebView-elementissä, vaikka sen toimintaa oli kokeiltu tarkoitukseen laaditulla verkkosivulla. Kävi ilmi, että ensimmäinen Android-versio, jonka WebView-elementissä WebSocket toimii, oli 4.4.4. Sovelluksen tuli kuitenkin toimia myös vanhemmilla laitteilla. Ratchet tarjoaa vaihtoehtoisena siirtotekniikkana ainoastaan flashsocketin, mutta Flash ei myöskään toimi mobiililaitteissa.

5.2.2 Socket.io-kirjasto

Valitsin vaihtoehtoiseksi teknologiaksi Socket.io:n, koska se tukee WebSocketin lisäksi long polling -tekniikkaa. Näin uudemmat laitteet voivat hyötyä WebSocket-teknologian tehokkuudesta, mutta sovellus toimii myös vanhemmilla laitteilla. Olisin halunnut käyttää PHP:tä, koska järjestelmän rajapinnat ja muu palvelinpuoli oli ohjelmoitu PHP:llä. Node.js ja Socket.io eivät olleet missään nimessä huono vaihtoehto, koska Node.js on kevyt ja JavaScript on myös tuttu kieli.

Käytin samanlaista lähestymistapaa Socket.io:n opetteluun kuin Ratchetin kanssa. Aloitin seuraamalla chat-esimerkkiä [45]. Kun sain tehtyä esimerkin kaltaisen sovelluksen, siirsin sen testipalvelimelle. Seuraavaksi ohjelmoin pienen Android-sovelluksen, joka avasi esimerkin WebView-elementtiin. Kokeilin sovellusta Androidin versiolla 4.3, ja se toimi moitteettomasti.

Kehittäminen Socket.io:lla oli helppoa ja nopeaa, koska siihen löytyi runsaasti esimerkkejä ja keskusteluja. Lisäksi se oli melko samanlainen kuin Ratchet, joten oli helppo ymmärtää, minkälaisia toimintoja ja osia tarvittiin. Socket.io ei tue WAMP-protokollaa, joten Ratchetin kanssa käytetylle tilaus-julkaisumallille täytyi keksiä jokin korvike. Socket.io

tarjoaa toiminnon, jossa asiakasohjelmat voivat liittyä huoneisiin. Päätiin käyttää huoneita samaan tapaan kuin aiheiden tilaamista Ratchetilla toteutetussa sovelluksessa.

Jotta asiakasohjelmat pystyivät liittymään huoneisiin, Socket.io-palvelin tarvitsee tapahtumakuuntelijan, joka rekisteröi yhteyden pyydettyyn huoneeseen. Huoneista ja yhdistäneistä asiakasohjelmista ei tarvitse itse pitää kirjaa, koska Socket.io hoitaa sen taustalla. Huoneeseen liittyminen asiakasohjelmalla ja liittymispyynnöt käsittelevä tapahtumakuuntelija ovat hyvin yksinkertaisia, kuten koodiesimerkeistä 9 ja 10 voidaan havaita. Koodiesimerkissä 9 luodaan ensin yhteys Socket.io-palvelimeen luomalla io-objekti, jolle annetaan parametrina palvelimen ip-osoite ja portti.

```
var socket = io("127.0.0.1:9000");
socket.emit('subscribe', 'new_fare_inquiry');
```

Koodiesimerkki 9. Asiakasohjelman liittyminen Socket.io-huoneeseen.

```
socket.on('subscribe', function(room) {
  socket.join(room);
});
```

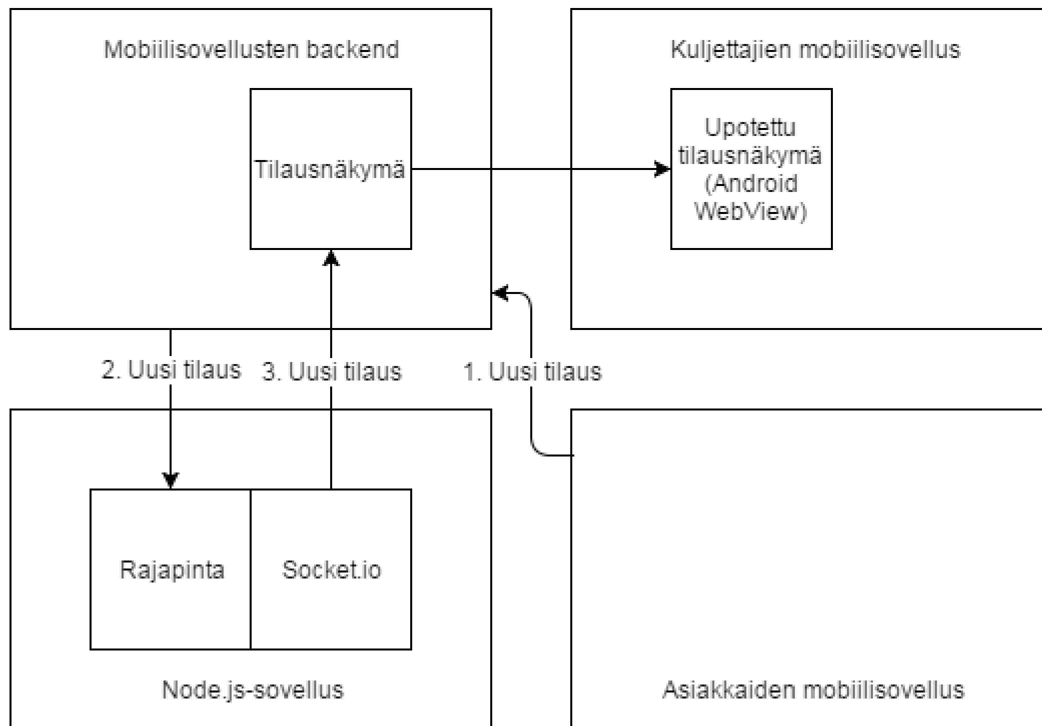
Koodiesimerkki 10. Socket.io-palvelimen tapahtumakuuntelija huoneisiin liittymistä varten.

Yhteyden muodostamisen jälkeen palvelimelle lähetetään viesti, joka sisältää huoneen nimen. Molemmissa esimerkeissä socket-muuttujan voidaan ajatella kuvastavan yhteyttä toiseen osapuoleen. Koodiesimerkissä 10 yhteys liitetään huoneeseen, jonka nimen asiakasohjelma lähetti palvelimelle.

Kun huoneet oli saatu toimiviksi, ainoa puuttuva osa oli viestien välittäminen CakePHP-sovelluksesta Socket.io-palvelimelle. Tätä varten Socket.io-sovellukseen lisättiin toinen palvelin, joka on yksinkertainen rajapintana toimiva HTTP-palvelin. Rajapinta ottaa vastaan JSON-muotoisia viestejä, jotka sisältävät huoneen nimen ja viestin. Rajapinnalle lähetetään viestinä uuden tilauksen tiedot heti sen jälkeen, kun tilaus on tallennettu tietokantaan. Rajapinta vastaanottaa viestin ja pyytää Socket.io:ta välittämään viestin kaikille asiakasohjelmille, jotka ovat liittyneet viestissä määriteltyyn huoneeseen.

Kuvassa 8 voidaan nähdä, miten Node.js-sovellus istuu koko järjestelmän muiden osien kanssa. Nyt mobiilisovellus tekee tilauksen, joka vastaanotetaan palvelinpuolen rajapinnassa. Uusi tilaus tallennetaan tietokantaan, minkä jälkeen se lähetetään myös Node.js-sovelluksen rajapinnalle. Rajapinta pyytää Socke.io:ta lähettämään tilauksen jälleen

eteenpäin tilausnäkymään, joka näytetään upotettuna WebView-elementissä kuljettajien mobiilisovelluksessa.



Kuva 8. Järjestelmän eri osat Node.js-sovelluksen lisäämisen jälkeen.

Tilauksilla voi olla eri prioriteetteja, ja jokaista prioriteettia varten luotiin oma huone. Lisäksi tilauslistasta poistettavia tilauksia varten luotiin vielä yksi huone lisää. Huoneiden avulla erilaiset tapahtumat voidaan helposti erotella eri tapahtumakuuntelijoihin ja sitä kautta reagoida tapahtumiin eri tavalla. Eri prioriteeteille tehdyt huoneet kontrolloivat, mille kuljettajille uusi tilaus näytetään.

5.2.3 Yhteyden salaaminen

Tuotantopalvelin käyttää salattua yhteyttä, joten tilauslistauksen ja Node.js-sovelluksen välisestä yhteydestäkin täytyi tehdä salattu. Tämän toteuttamiseen oli kaksi eri vaihtoehtoa. Saapuvat salatut yhteydet olisi voinut vastaanottaa Apachella, joka olisi purkanut salauksen ja ohjannut yhteyden edelleen salaamattomana Node.js-sovellukselle. Toinen vaihtoehto oli käyttää SSL-sertifikaattia suoraan Node.js-sovelluksessa. Päätin käyttää jälkimmäistä vaihtoehtoa, koska sen toteuttaminen oli paljon yksinkertaisempaa.

Salattu yhteys saatiin käyttöön käyttämällä HTTP-moduulin sijasta HTTPS-moduulia ja antamalla palvelimelle parametrina sertifikaattitiedostojen sisältö. Koodiesimerkissä 11 on havainnollistettu salattua yhteyttä käyttävän HTTP-palvelimen luominen.

```
var options = {
  key: fs.readFileSync('/path/to/cert/cert.key'),
  cert: fs.readFileSync('/path/to/cert/cert.crt'),
  requestCert: true
};

var socketServer = https.createServer(options, app);
```

Koodiesimerkki 11. Salattua yhteyttä käyttävän HTTP-palvelimen luominen Node.js-sovelluksessa.

Node.js palvelin toimii samalla palvelimella kuin järjestelmän palvelinpuolen muut osiot, joten salausta varten käytettiin samoja sertifikaattitiedostoja. Salauksen lisäämisen jälkeen Node.js palvelimeen täytyy yhdistää käyttämällä sertifikaatissa määriteltyä verkkotunnusta. Nyt tilauslista ja Node.js-sovellus käyttävät samaa verkkotunnusta, mutta portit eivät ole samat.

5.2.4 Viimeistely

Viimeistelynä tilauslistaan tehtiin lopulliset tapahtumakuuntelijat tilausten lisäämiseksi ja poistamiseksi listasta. Uusien tilausten tiedot asetettiin HTML-elementteihin, jotka lisättiin uusina riveinä tilauslistaan JavaScriptin avulla. Lisäämisessä ja poistamisessa hyödynnettiin jQuery-kirjaston animaatioita, jotta muutokset olivat helpommin havaittavissa.

Palvelinpuolella viimeinen tehtävä oli luoda Node.js-sovellukselle alustusskripti, joka käynnistää sovelluksen automaattisesti, kun palvelin käynnistyy. Tämä on tärkeää, koska muuten sovellus täytyisi käydä käynnistämässä erikseen, jos palvelin kaatuisi tai pitäisi käynnistää uudelleen jostain syystä. Skripti suorittaa komennon, joka käynnistää Node.js-sovelluksen Forever-pakettia hyödyntäen. Forever on npm-paketti, jonka avulla Node.js-sovelluksia voi suorittaa helposti taustalla.

6 Yhteenveto

Insinöörityössä tutkittiin reaaliaikaisen webin teknologioita ja tutkimustulosten perusteella laadittiin suunnitelma ja toteutus taksikuljettajien tilausnäkymän nykyaikaistamiseksi. Tilausnäkymä on verkkosivu, ja se kärsi asiakas-palvelinmalliin perustuvan HTTP:n asettamista rajoitteista. Tavoitteena oli tehdä tilausnäkymästä käytettävämpi ja reaaliaikainen sekä samalla kerätä tietoa ja osaamista uusista teknologioista. Työn tuloksia voitaisiin hyödyntää tulevissa projekteissa.

Reaaliaikaisuutta jäljitteleviä tekniikoita on ollut jo pitkään. Nämä vanhat tekniikat kiertävät HTTP:n rajoitteita kekseliäin keinoin, mutta ne kärsivät skaalautuvuusongelmista, jos sovelluksella on runsaasti käyttäjiä. Uudet teknologiat on suunniteltu varta vasten reaaliaikaisen webin tarpeisiin. Teknologiat ja niiden protokollat ovat standardoituja ja toimivat tehokkaasti myös suurilla käyttäjämäärillä. Uudet teknologiat toimivat ainoastaan uusilla selaimilla. Vanhat tekniikat puolestaan toimivat hyvinkin vanhoilla selaimilla, koska ne perustuvat HTTP-protokollaan.

Tutkimustulosten perusteella tilausnäkymää lähdettiin uudistamaan kehittämällä näkymän palvelinpuolen rinnalle erillinen WebSocket-palvelin. Ensimmäinen versio toteutettiin Ratchet PHP-kirjastolla. Lopputulos oli muuten hyvin toimiva, mutta WebSocket-yhteys ei toiminut vanhemmissa Android-versioissa sovellukseen upotetussa näkymässä.

Socket.io on hyvin tunnettu sen saavuttamasta laajasta selaintuesta. Socket.io yhdistää WebSocket-teknologian ja useita vanhoja tekniikoita yhden rajapinnan taakse. Tämän vuoksi toinen versio toteutettiin Node.js:llä ja Socket.io-kirjastolla. Socket.io toimi erinomaisesti niin vanhoilla kuin uusillakin Android-versioilla. Kirjaston avulla WebSocket-teknologiaa tukevat uudet selaimet hyötyvät uuden teknologian tehokkuudesta, mutta samalla sovellus toimii myös vanhoissa selaimissa.

Lopputuloksena syntyi reaaliaikainen tilausnäkymä, joka näyttää mobiilisovelluksella tehdyt uudet tilaukset lähes välittömästi, kun ne on tallennettu tietokantaan. Reaaliaikaisuus toi jo itsessään huomattavasti paremman käyttökokemuksen tilausnäkymälle, mutta sitä saatiin paranneltua vielä lisäämällä tilauksiin tapahtuviin muutoksiin animaatioita.

Server Sent Events olisi ollut toinen teknologia, jolla olisi voinut toteuttaa tilausnäkymän reaaliaikaisuuden, mutta sen käyttö jäi kokonaan kokeilematta. SSE tarjoaisi yksisuuntaisen yhteyden palvelimelta asiakkaalle, joka olisi ollut riittävä, koska asiakasohjelman ei tarvitse lähettää mitään palvelimelle.

Järjestelmä ei ole tämän raportin kirjoittamisen hetkellä vielä virallisessa käytössä. On hyvin todennäköistä, että Node.js-palvelimeen tulee vielä jatkokehitystä. Todennäköisesti siihen toteutetaan ainakin tunnistautumistoiminto niin, että Socket.io tietää, mitä kuljettajaa kukin yhteys vastaa.

Lähteet

1 Legetter Phil. 2013. The top 10 realtime web apps. Verkkodokumentti. Creativebloq. <http://www.creativebloq.com/app-design/top-10-realtime-web-apps-5133752>. Luettu 15.2.2016.

2 Legetter Phil. 2013. History, Background, Benefits & Use Cases of Realtime. 2013. Verkkodokumentti. Legetter.co.uk. <http://www.leggetter.co.uk/2013/10/28/history-background-benefits-use-cases-realtime.html>. Luettu 15.2.2016.

3 MacManus Richard. 2010. 5 Use Cases for The Real-Time Web. Verkkodokumentti. Readwrite. <http://readwrite.com/2010/05/25/5_use_cases_for_the_real-time_web>. Luettu 15.2.2016.

4 Wähner Kai. 2014. Real-Time Stream Processing as Game Changer in a Big Data World with Hadoop and Data Warehouse. Verkkodokumentti. InfoQ. <http://www.infoq.com/articles/stream-processing-hadoop>. Luettu 16.2.2016.

5 Hugg John. 2014. Fast data: The next step after big data. Verkkodokumentti. InfoWorld. <http://www.infoworld.com/article/2608040/big-data/fast-data--the-next-step-after-big-data.html>. Luettu 16.2.2016.

6 Pusher Use Cases. Verkkodokumentti. Pusher. <https://pusher-community.github.io/real-time-laravel/introduction/pusher-use-cases.html>. Luettu 16.2.2016.

7 Cloud 9 IDE. Verkkodokumentti. Cloud9. <https://c9.io/>. Luettu 17.2.2016.

8 Make remote design work. Verkkodokumentti. Mural. <https://mural.ly/>. Luettu 17.2.2016.

9 Bergström Sven. 2012. Real Time Multiplayer in HTML5. Verkkodokumentti. Build New Games. <http://buildnewgames.com/real-time-multiplayer/>. Luettu 17.2.2016.

10 Rai Rohit. 2013. Socket.io Real-time Web Application Development. Mumbai: Pact Publishing.

11 Community-built projects, powered by Arduino. Verkkodokumentti. Hackster.io. <https://www.hackster.io/arduino/projects?page=4&sort=trending>. Luettu 17.2.2016.

12 Internet of things. Verkkodokumentti. Raspberry Pi. <https://www.raspberrypi.org/blog/tag/internet-of-things/>. Luettu 18.2.2016.

13 HTTP Operational Model and Client/Server Communication. Verkkodokumentti. The TCP/IP Guide. <http://www.tcpipguide.com/free/t_HTTPOperationalModelandClientServerCommunication.htm>. Luettu 18.2.2016.

14 The Client/Server Model. Verkkodokumentti. IBM Knowledge Center. <https://www-01.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.ieak500/ieak511.htm>. Luettu 18.2.2016.

15 Hoffman Jason. 2013. Building for the Internet of things. Verkkodokumentti. VentureBeat. <http://venturebeat.com/2013/01/31/building-for-the-internet-of-things-and-the-demise-of-the-client-server-model/>. Luettu 18.2.2016.

16 Kalali Masoud, Mehta Bhakti. 2013. Developing RESTful Services with JAX-RS 2.0, WebSockets, and JSON. Mumbai: Packt Publishing.

17 Gechev Minko. 2008. How do I implement basic long polling? Verkkodokumentti. Stackoverflow. <http://stackoverflow.com/a/333884>. Luettu 23.2.2016.

18 How to optimize apache web server for maximum concurrent connections or increase max clients in apache. 2012. Verkkodokumentti. Generic Articles. <http://www.genericarticles.com/mediawiki/index.php?title=How_to_optimize_apache_web_server_for_maximum_concurrent_connections_or_increase_max_clients_in_apache.> Luettu 23.2.2016.

19 Configuring Apache for Maximum Performance. 2006. Verkkodokumentti. Linux Gazette. <http://linuxgazette.net/123/vishnu.html>. Luettu 23.2.2016.

20 Robison Arch. 2007. Why too many threads hurts performance, and what to do about it. Verkkodokumentti. Code Guru. <http://www.codeguru.com/cpp/sample_chapter/article.php/c13533/Why-Too-Many-Threads-Hurts-Performance-and-What-to-do-About-It.htm>. Luettu 23.2.2016.

21 The WebSocket Protocol. 2011. Verkkodokumentti. IETF. <https://tools.ietf.org/html/rfc6455>. Luettu 6.3.2016.

22 The WebSocket API. 2011. Verkkodokumentti. W3C. <https://www.w3.org/TR/2011/WD-websockets-20110929/>. Luettu 6.3.2016.

23 Upgrading HTTP to WebSocket. 2014. Verkkodokumentti. Enterprise Web Book. <http://enterprisewebbook.com/ch8_websockets.html>. Updated 20 December 2014. Luettu 8.3.2016.

24 Jaitla Jasdeep. 2015. REST vs. WebSockets. Verkkodokumentti. PubNub.. <https://www.pubnub.com/blog/2015-01-05-websockets-vs-rest-api-understanding-the-difference/>. Luettu 8.3.2016.

25 WebSocket. Verkkodokumentti. Oreilly. <http://chimera.labs.oreilly.com/books/1230000000545/ch17.html>. Luettu 8.3.2016.

26 WebSockets for PHP. Verkkodokumentti. Ratchet. <http://socketo.me/>. Luettu 19.3.2016.

27 Push to an Existing Site. Verkkodokumentti. Ratchet. <http://socketo.me/docs/push>. Luettu 19.3.2016.

28 AutobahnJS. Verkkodokumentti. Autobahn. <ttp://autobahn.ws/js/>. Luettu 19.3.2016.

29 Bidelman Eric. 2010. Stream Updates with Server-Sent Events. Verkkodokumentti. HTML5 Rocks. <http://www.html5rocks.com/en/tutorials/eventsource/basics/>. Luettu 19.3.2016.

30 EventSource polyfill. Verkkodokumentti. GitHub. <https://github.com/Yaffle/EventSource>. Luettu 19.3.2016.

31 HTTP access control (CORS). 2016. Verkkodokumentti. Mozilla Developer Network. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS>. Updated 1 April 2016. Luettu 20.3.2016.

32 Using server-sent events. 2015. Verkkodokumentti. Mozilla Developer Network. <https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events>. Updated 17 September 2015. Luettu 20.3.2016.

33 Frequent questions. Verkkodokumentti. WebRTC. <https://webrtc.org/faq/>. Luettu 20.3.2016.

34 WebRTC code samples. Verkkodokumentti. GitHub. <https://github.com/webrtc/samples>. Luettu 20.3.2016.

35 Dutton Sam. 2014. Getting Started with WebRTC. Verkkodokumentti. HTML5 Rocks. <http://www.html5rocks.com/en/tutorials/webrtc/basics/>. Luettu 20.3.2016.

36 Introduction to WebSockets. Verkkodokumentti. Ratchet. Verkkodokumentti. <http://socketo.me/docs/>. Luettu 20.3.2016.

37 Creating Your First Application. Verkkodokumentti. Ratchet. <http://socketo.me/docs/hello-world>. Luettu 20.3.2016.

38 Getting Started. Verkkodokumentti. Composer. <https://getcomposer.org/doc/00-intro.md>. Luettu 20.3.2016.

39 About Node.js. Verkkodokumentti. Node.js. <https://nodejs.org/en/about/>. Luettu 20.3.2016.

40 Harter Marc. 2014. Writing Modular Node.js Projects for Express and Beyond. Verkkodokumentti. Strongloop. <https://strongloop.com/strongblog/modular-node-js-express/>. Luettu 20.3.2016.

41 Robbins Charlie. 2013. Npm: innovation through modularity. Verkkodokumentti. Nodejitsu. <http://blog.nodejitsu.com/npm-innovation-through-modularity/>. Luettu 22.3.2016.

42 Module Counts. Verkkodokumentti. Modulecounts. <http://www.modulecounts.com/>. Luettu 22.3.2016.

43 Kelleher Fionn. 2014. Understanding Socket.IO. Verkkodokumentti. Nodesource. <https://nodesource.com/blog/understanding-socketio/>. Luettu 22.3.2016.

44 An Wei. 2011. Define transport types on the client side. Verkkodokumentti. Stackoverflow. <http://stackoverflow.com/questions/7016144/define-transport-types-on-the-client-side>. Luettu 22.3.2016.

45 Get Started: Chat Application. Verkkodokumentti. Socket.io. <http://socket.io/get-started/chat/>. Luettu 22.3.2016.