Tampere University

Jarno Matarmaa

WEBSOCKET-OHJELMOINTIRAJAPINNAN KÄYTETTÄVYYS SOVELLUSKEHITYKSESSÄ

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Joulukuu 2020

TIIVISTELMÄ

Jarno Matarmaa: *WebSocket-ohjelmointirajapinnan käytettävyys sovelluskehityksessä*
Kandidaattitutkielma
Tampereen yliopisto
Tietojenkäsittelytieteen tutkinto-ohjelma
Joulukuu 2020

Tässä tutkielmassa tarkastellaan WebSocket-protokollan toimintaperiaatteita ja tutustutaan sen ohjelmointirajapintaan. Tavoitteena on tunnistaa tärkeimmät WebSocket API:n (Application Programming Interface) käytettävyysongelmat. Protokollan käytettävyyttä tarkastellaan erityisesti sovelluskehityksen näkökulmasta. Käytettävyydessä otetaan huomioon ensisijaisesti ohjelmointirajapinnan käytön ja kehityksen ketteryys, mutta myös yhteensopivuus-, luotettavuus- ja turvallisuusnäkökulmat.

Perinteisten HTTP-protokollaan perustuvien, *pyyntö-vastaus*-sykliä hyödyntävien protokollien käytettävyys ei vastaa enää uusimpia vaatimuksia monissa sovelluksissa, sillä ne lisäävät tarpeetonta verkon kuormitusta, eikä niitä ole suunniteltu reaaliaikaiseen ja kaksisuuntaiseen verkkoviestintään. Myös WebSocket käyttää HTTP-protokollaa yhteyden avaamisessa asiakkaan (client) ja palvelimen välille, mutta se on ensimmäinen aidosti kaksisuuntaisen kommunikoinnin mahdollistava verkkoteknologia. Kaksisuuntaisuuden tuoman suorituskykyedun ansiosta se on erittäin käyttökelpoinen teknologia moniin sellaisiin absoluuttista reaaliaikaisuutta vaativiin sovelluskohteisiin, joissa aiemmat protokollat ovat osoittautuneet riittämättömiksi.

Selaimen tarjoama JavaScript-kielinen natiivi WebSocket API on suppea ja yksinkertainen. Koska WebSocket-protokollaa voidaan käyttää myös muissa kuin HTTP-perusteisissa selainsovelluksissa, sen ohjelmointirajapinta ei sisällä ratkaisuja palomuurien, virustorjuntaohjelmien, verkon kuormituksentasaajien tai yhteyskatkoksien käsittelyyn. Tämä muodostaa protokollan käytettävyysongelmia sovelluskehityksessä, joiden ratkaisuksi tässä tutkielmassa esitetään rajapintalaajennoksia.

Tutkielmassa osoitetaan, miten Socket.io API:n avulla voidaan saavuttaa merkittäviä WebSocket-protokollan käytettävyysparannuksia säilyttämällä ohjelmointirajapinnan johdonmukaisuus ja selkeys sovelluskehityksessä. Vaikka web-sovelluskehittäjän ratkaistavaksi jää vielä paljon tietoturvahaavoittuvuuksia, kuten *palvelunestohyökkäysten*, *WebSocket-sivustokaappausten* ja *väärennettyjen pyyntöjen* käsittelyä, muodostavat Socket.io ja sen sisältämät moduulit hyvin käyttökelpoisen rajapinnan web-sovellusten tiedonsiirto-ongelmien ratkaisemiseksi. Lisäksi aiemmat tutkimukset osoittavat, että käytettävyysparannukset eivät merkittävästi heikennä yhteyden tiedonsiirtoviivettä verrattuna WebSocket-ohjelmointirajapintaan. Tästä huolimatta suorituskykykriittisissä sovelluksissa tulisi käyttää aina WebSocket API:a.

Avainsanat:
WebSocket, WebSocket API, Socket.io, Engine.io, reaaliaikainen verkkoviestintä, kaksisuuntainen verkkoviestintä

Sisällysluettelo

1 Johdanto

Web-selain on laajimmalle levinnyt sovelluskehittäjien käytössä oleva kehitysalusta, joka on asennettuna lähes jokaiseen älypuhelimeen, tablettiin ja tietokoneeseen. Web-selainten toiminnallisuudet ja ominaisuudet laajenevat jatkuvasti. Grigorik (2015) toteaakin *High Performance Browser Networking* -kirjansa esipuheessa, että selainten verkkoinfrastruktuurin kehittyminen on paras esimerkki teknologian nopeasta edistyksestä ja innovaatioista viime vuosikymmenellä. HTTP:n *pyyntö-vastaus*-vuorovaikutus web-selaimissa rajoitti aiemmin voimakkaasti web-kehitystä, mutta nykyisin käytössä oleva WebSocket-teknologia mahdollistaa tehokkaan *suoratoiston* (streaming), *kaksisuuntaisen* (bi-directional, full-duplex) ja *reaaliaikaisen* (real-time) kommunikoinnin, sekä mukautettujen sovellusprotokollien käytön.

Esineiden Internet (IoT, Internet of Things) -järjestelmä on yksi merkittävimmistä web-kehityksen vaikuttimista ja sitä käytetään monilla eri aloilla, kuten maataloudessa, turvallisuudessa ja teollisuudessa. Se asettaa erityiset vaatimuksensa web-selainten tietoliikenteen nopeudelle ja luotettavuudelle. Web-teknologiaa ei käytetä enää vain tavanomaisessa datan prosessoinnissa, vaan myös prosessien valvonnassa, seurantajärjestelmissä ja älykotien turvajärjestelmissä (Soewito, Christian, Gunawan, Diana & Kusuma, 2019). Esineiden Internet on järjestelmä, joka perustuu monesti anturien mittamaan tiedon prosessointiin ja sen perusteella tapahtuvaan toimintaan. Sekä monissa teollisuuden tuotantolinjojen prosesseissa että esimerkiksi itse kulkevissa maatalouden työkoneissa on kriittistä, että reaktioajoissa ei ole liian suurta viivettä. Soewiton ym. (2019) mukaan datan siirto anturilta Internetin kautta valvontalaitteelle voi olla enintään noin 300 ms, johon useimmat web-teknologiat eivät kykene. Tämä onkin ollut keskeinen ongelma käytettäessä HTTP-protokollia, ja toisaalta lisännyt web-sovellusten monimutkaisuutta, kun tarkoitukseen sopimatonta teknologiaa on yritetty hyödyntää huonolla menestyksellä.

Tässä tutkielmassa ei käsitellä suorituskykyä, mutta parempien valintojen tekemiseksi myös sovelluskehittäjän on ymmärrettävä käyttämänsä teknologian perusteet. Siksi pohjustan aihetta ensin WebSocketin taustatiedoilla. Sen sijaan tutkielma käsittelee WebSocket API:a (Application Programming Interface) ja pyrkimyksenä on nostaa esille tärkeimmät teknologian käytettävyyteen ohjelmistokehityksessä vaikuttavat seikat. Tavoitteena on löytää näkökulmia siihen, minkälaisia käytettävyysongelmia on syytä tiedostaa ja voidaanko niihin varautua ja voidaanko niitä ennaltaehkäistä. Syvällisen tutkimuksen sijaan tarkastelen asioita osana isompaa kokonaiskuvaa ja huomioin laajasti eri käytettävyyden osa-alueita. Tässä tutkielmassa käytettävyydellä tarkoitetaan sovellusohjelmoijan kannalta yleisen tason asioita, joka merkitsee myös ohjelmointirajapintojen käytön tietoturvahaavoittuvuuksien huomiointia, sillä ne vaikuttavat olennaisesti sovelluskehityksessä käytettäviin algoritmiratkaisuihin ja siten käytettävyyteen.

Mitä WebSocket oikeastaan tarkoittaa ja miksi se kehitettiin? Luvussa *2 WebSocket tiedonsiirtoprotokollana* pyrin avaamaan protokollan taustoja lyhyesti web-kehityksen historian ja muiden protokollien avulla. Määrittelen myös termit reaaliaikaisuus ja kaksisuuntaisuus niiden kontekstissaan. Avaan lyhyesti myös termin *socket*, joka puolestaan vaatii verkkokerrosten tuntemista. Taustatietojen jälkeen, luvussa *3 Sovellusohjelmointirajapinta ja algoritmit* käydään läpi WebSocket API:n funktioita ja koodiesimerkkien avulla protokollan käyttöä kaksisuuntaisen ja reaaliaikaisen yhteyskanavan muodostuksessa. Tässä tutkielmassa WebSocket API:lla tarkoitetaan vaihtelevasti hieman eri asioita, riippuen asiayhteydestä, jossa se mainitaan. Yleisesti sillä viitataan vain selaimen JavaScript API:in, mutta toisinaan sillä tarkoitetaan myös palvelimen *Nodejs*-sovelluksen funktiokirjastoa, vaikka ne ovatkin täsmällisesti katsoen eri asioita. Tässä tutkielmassa WebSocket API tarkoittaa vain protokollaan liittyvää JavaScript API:a, mutta luonnollisesti myös muille ohjelmointikielille se on saatavilla. Luvussa *4 Käytettävyys ja ominaisuudet* arvioidaan protokollan käytettävyyttä 3. luvun koodiesimerkkien perusteella, sekä tuodaan esille muutamia haavoittuvuuksia ja uhkia, jotka voidaan tulkita potentiaalisina käytettävyyttä heikentävinä seikkoina. Sovelluskehityksen kannalta jonkin teknologian käytettävyys perustuu pääasiassa olemassa oleviin API-kirjastoihin sekä siihen, miten ne on toteutettu ja miten eri ominaisuudet, kuten tietoturvallisuus ja vikasietoisuus on huomioitu. Jos ohjelmoijalle jää paljon omien ratkaisujen luomista yleisen tason ongelmiin, se heikentää käytettävyyttä.

Käytettävyys koostuu siis monista eri osa-alueista, kuten vikasietoisuus, luotettavuus ja tietoturvallisuus, jotka voivat vaikuttaa protokollan käyttöön. Lisäksi tarkastelen tässä tutkielmassa tiedon konkreettista ohjelmallista käsittelyä ja siirtoa yhteyskanavan läpi, eli missä muodossa tietoa voidaan siirtää ja miten siitä huolehditaan. Käytettävyydessä emme kuitenkaan ota kantaa siihen, onko selaimen JavaScript-sovellus optimoitu käytettävissä olevien laitteistoresurssien suhteen, vaan tutkimus rajataan JavaScript-kieliseen selainsovellukseen, koska käsittelemme web-teknologioita. On silti hyvä muistaa, että WebSocket ei edellytä selainsovelluksen käyttöä, sillä protokollaa voidaan laajasti hyödyntää erityyppisissä muissakin sovelluksissa.

2 WebSocket tiedonsiirtoprotokollana

WebSocket on hyvin usein keskeinen käsite etenkin tiedonsiirron suorituskykykeskusteluissa. Lähes kaikissa aihetta käsittelevissä artikkeleissa viitataan jollain tavalla tiedonsiirron tehokkuuteen ja latenssiin. Tämä ei sinänsä ole ihme, sillä juuri näiden ongelmien ratkaisemiseksi – vastaamaan muuttuneen web-sovellusympäristön luonteen tarpeisiin – WebSocket kehitettiin. Toisaalta sen kuvaamiseen on luontevaa käyttää muita tiedonsiirtoprotokollia, jotta sen tuomat edut havainnollistuisivat paremmin. WebSocket ei kuitenkaan sinänsä ole kovin ihmeellinen tai monimutkainen protokollana ja ennemminkin nousee mieleen retorinen kysymys: miksi sitä edeltäviä protokollia kehitettiin, koska niiden toimintaperiaatteet ovat ilmeisen

mahdottomia, kun tavoitteena on saavuttaa tehokas, reaaliaikainen, kahteen suuntaan saumattomasti toimiva kommunikointiyhteys. Lubbers ja Greco (2013) ilmaisevat asian paremmin:

> *"WebSocket is great, but what you can do once you have a full-duplex socket connection available in your browser is even greater."*

WebSocketin luomat mahdollisuudet web-kehitykselle ovatkin paljon mittavammat ja suuremmoisemmat, kuin itse protokolla, joka ne mahdollistaa.

2.1 Toimintaperiaate

WebSocket-protokolla on suunniteltu siten, että se on yhteensopiva nykyisen verkkoinfrastruktuurin kanssa. Osana sen suunnitteluperiaatteita määritellään, että WebSocket-yhteys alkaa ensin HTTP-yhteytenä, jonka tarkoituksena on taata täydellinen *taaksepäin yhteensopivuus* (backwards compatibility) edeltävien protokollien kanssa. Yhteensopivuus saavutetaan käyttämällä TCP portteja 80 ja 443. Yhteyden muodostuksen jälkeen sekä palvelin että selain voivat lähettää ja vastaanottaa dataa itsenäisesti saman kaksisuuntaisen TCP-yhteyden kautta (Hribernik & Kos, 2020). Yhteyden muodostus suoritetaan siten, että selain lähettää ensin HTTP-pyynnön palvelimelle ja ilmoittaa haluavansa vaihtaa protokollaa HTTP:stä WebSocketiin. Protokollanvaihtoa HTTP:stä WebSocketiin kutsutaan WebSocket-kättelyksi (Lubbers & Greco, 2013). Palvelin saa tiedon tästä *päivitysotsakkeen* (upgrade header) kautta, joka lähetetään yhteyspyynnön mukana (Kulshrestha, 2013):

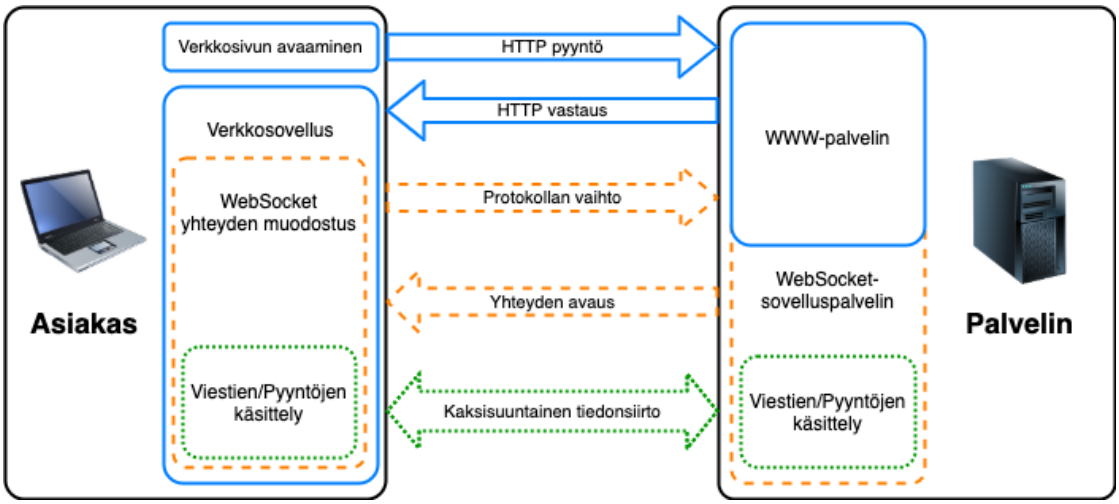
```
GET ws://example.com/?encoding=text HTTP/1.1
Origin: http://example.com
Cookie: __ASWE1241
Connection: Upgrade
Host: example.com
Sec-Websocket-Key: KJsdnkjwqel12ee454==
Upgrade: Websocket Sec-Websocket-Version: 13
```

Jos palvelin tukee WebSocket-protokollaa, se hyväksyy pyynnön ja vaihtaa yhteystyypin HTTP:stä WebSocket-yhteyteen (Kulshrestha, 2013):

```
HTTP/1.1 101 Websocket Protocol Handshake
Connection: Upgrade
Server: Websocket-Server
Upgrade: Websocket
Access-Control-Allow-Origin: http://Websocket.org
Access-Control-Allow-Credentials: true
Sec-Websocket-Accept: hAKDSL/WEKAldmQWmasdsAS=
```

```
Access-Control-Allow-Headers: content-type
```

Kun kättely on valmis, HTTP-yhteys korvataan WebSocket-yhteydellä samalla TCP-kanavalla. Kun yhteys on muodostettu, asiakkaan ja palvelimen välille luodaan kaksisuuntainen kommunikointikanava, jossa *datakehysten* (frames) vaihto jatkuu (Kulshrestha, 2013). *Kuvio 2-1* havainnollistaa WebSocket-yhteyden muodostamisen vaiheita.



Kuvio 2-1: WebSocket yhteyden muodostaminen. (Karla & Tarnawski, 2019)

WebSocket-protokollan toimintaperiaatteiden ja sen tuomien etujen kuvaamiseksi tutkielmassa mainitaan myös muutama vaihtoehtoinen HTTP-protokollaa hyödyntävä yhteystapa asiakkaan ja palvelimen välillä. Tiedonsiirrossa käytetyillä WebSocketia edeltävillä protokollilla, joista käytän tässä *XHR-polling* tai *XHR* (XMLHttpRequest) -nimitystä, voidaan muodostaa reaaliaikaisilta ja kaksisuuntaisilta tuntuvia yhteyksiä, mutta koska niitä ei ole suunniteltu tämänkaltaiseen muuttuneeseen web-sovellusten käyttötavan ympäristöön, ne eivät sovellu tähän käyttötarkoitukseen (Srinivasan, Scharnagl & Schilling, 2013). Seuraavassa luvussa tarkastellaankin reaaliaikaisuutta ja kaksisuuntaisuutta yleisellä tasolla ja pyritään vertailun avulla selventämään näitä WebSocketiin liittyviä käsitteitä.

2.2 Reaaliaikaisuus ja kaksisuuntaisuus

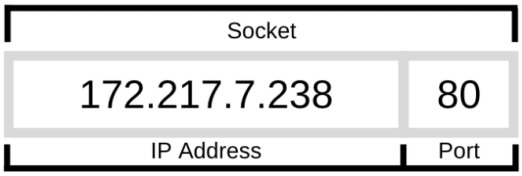
Tässä yhteydessä ei ole tarkoitus muodostaa tarkkaa määritelmää sille, mitä *reaaliaikaisuudella* (real-time communication) tarkoitetaan. Todetaan yleisesti, että sillä tarkoitetaan jonkin operaation suorittamista tietyn ennaltamäärätyn aikavälin kuluessa. Tavallisesti sopiva mittayksikkö on millisekunti. Esimerkiksi jos HTTP-kyselyn pyyntö/vastaus syklin kesto eli latenssi on keskimäärin 500 ms, niin sen voidaan sanoa olevan reaaliaikainen. WebSocketin vastaava on vain noin kymmenesosan tästä, eli 57 ms (Srinivasan ym., 2013). Reaaliaikaisuus ei siis tässä yhteydessä ole tarkoin määritelty, vaan sillä viitataan reaaliaikaisuuden tuntuun esimerkiksi jonkin web-sovelluksen toimintaperiaatteessa.

Ennen WebSocketia käytössä olleiden HTTP-protokollien, kuten *XHR-pollingin* avulla kyettiin muodostamaan reaaliaikaisia verkkoyhteyksiä, jotka perustuivat käyttäjän *selaimen* (client) tekemiin automaattisiin pyyntöihin (Srinivasan ym., 2013) ja (Pimentel & Nickerson, 2012). Ne olivat ensimmäinen askel manuaalisesta sivun päivittämisestä kohti reaaliaikaisuutta. Niissä reaaliaikaisuus saavutetaan kuitenkin suorituskyvyn kustannuksella, sillä ne lähettävät säännöllisesti tarpeettomia kyselyitä, jotka sisältävät HTTP-otsakkeita, koska todellista pysyvää yhteyttä ei ole, kuten WebSocketissa. Näissä siis käytännössä yhteys muodostetaan joka kerta uudelleen, mutta koska se tapahtuu taustalla, se näkyy käyttäjälle esimerkiksi jatkuvasti ajan tasaisena sisältönä. Jos palvelimella olevassa datassa tapahtuu muutoksia, ne päivittyvät toistuvien ja säännöllisten HTTP-kyselyiden ansiosta asiakkaalle. Tällä tavoin saavutetaan kyllä reaaliaikaisuus, mutta näemme selvästi, etteivät perinteiset *XHR-polling*-protokollat ole lainkaan suunniteltu tai alun perin tarkoitettu tämänkaltaiseen viestintään, kuten myös Srinivasan ym. (2013) toteavat. Samoin Pimentelin ja Nickersonin (2012) mukaan HTTP:tä ei ole suunniteltu reaaliaikaiseen, kaksisuuntaiseen viestintään reaaliaikaisten HTTP-verkkosovellusten monimutkaisuuden vuoksi. Tämän takia HTTP voi simuloida reaaliaikaisuutta vain korkealla hinnalla – lisääntyneellä viiveellä ja suurella verkkoliikenteellä. HTTP:tä ei siis suunniteltu ylläpidettäväksi yhteysprotokollaksi (Kulshrestha, 2013).

WebSocket mahdollistaa myös *kaksisuuntaisuuden* (bi-directional, full-duplex) verkkoviestinnässä. Perinteiset HTTP-kyselyt perustuvat yksinomaan selaimen tekemiin kyselyihin, mutta WebSocket-protokollan avulla myös palvelin voi lähettää viestejä erityisen *yhteyskanavan* (socket) kautta (*luku 3.1*). Se onkin ainoa tiedonsiirtoprotokolla, joka sallii kaksisuuntaisen viestinnän samaa TCP-yhteyttä käyttämällä (Grigorik, 2013, 285). Tosin muun muassa Agarval (2012) kutsuu myös *XHR-polling*-kyselyitä kaksisuuntaisiksi, vaikka todellisuudessa ne perustuvat yksipuolisiin selaimen palvelimelle lähettämiin pyyntöihin. Kaksisuuntaisuus voidaankin määritellä siten, että asiakkaan tekemien muutosten palvelimelle lähettämisen lisäksi, myös muiden asiakkaiden muutokset lähetetään palvelimelta asiakkaalle, mutta siinä ei oteta kantaa siihen, miten se on toteutettu algoritmien tasolla. Myöskään kaksisuuntaisuus ei siis tässä yhteydessä ole tarkka määritelmä, vaan sillä tarkoitetaan pikemminkin – samaan tapaan kuin reaaliaikaisuudessa – kaksisuuntaisuuden tuntua.

2.3 Sijoittuminen verkkokerroksilla

WebSocket on sovelluskerroksen eli OSI-mallin ylimmän seitsemännen kerroksen protokolla ja osa *HTML5* (Hypertext Markup Language) -standardisointia. Nimen loppuosa *socket* viittaa sen käyttämään TCP-yhteystyyppiin eli IP-osoitteen ja verkkoportin yhdistelmään (*kuva 2-2*). OSI-mallissa *socketit* eli niin sanotut



Kuva 2-2: Socket eli yhteyskanava

yhteyskanavat sijoittuvat kuitenkin viidennelle verkkokerrokselle TCP/IP- eli siirtokerroksen päälle. WebSocketia ei tule siis sekoittaa tässä mielessä istuntokerroksen protokollaksi, sillä se on osa selaimen toiminnallisuutta ja se toimii niin *välityspalvelinten* (proxy), palomuurien, suodattimien kuin todennusprosessienkin kanssa (Hribernik & Kos, 2019), jotka sijoittuvat puolestaan esitystapakerrokselle

3 Sovellusohjelmointirajapinta ja algoritmit

Yleisemmällä tasolla WebSocketilla viitataan tavallisesti luvussa kaksi läpi käytyihin asioihin, mutta web-ohjelmoinnissa sillä tarkoitetaan täsmällisemmin WebSocket-ohjelmointirajapintaa eli *API*-kirjastoa (Application Programming Interface). WebSocket API:a kehittää *W3C* (World Wide Web Consortium), mutta WebSocket-protokolla (RFC 6455) ja sen laajennokset ovat puolestaan *Internet Engineering Task Forcen* (IETF) määrittelemiä ja HyBi-työryhmän kehityksessä (Grigorik, 2013, 285). WebSocket API mielletään usein myös osaksi HTML5-teknologiaa, koska se määriteltiin alun perin sen osana. Myöhemmin se siirrettiin kuitenkin erikseen omaan standardiasiakirjaansa, jotta määrittely voitaisiin pitää kohdennettuna (Lubbers & Greco, 2013). Ohjelmointirajapinta on saatavilla useille eri ohjelmointikielille, mutta tässä luvussa tutustumme tarkemmin JavaScript-kieliseen WebSocket API:in selaimissa.

3.1 WebSocket API

Selaimen tarjoama WebSocket API on funktioiden lukumäärältään huomattavan pieni ja yksinkertainen. Myöskään sen käyttötavat eivät eroa merkittävästi muiden protokollien ohjelmointirajapinnoista. *Taulukossa 3-1* WebSocketin JavaScript API-kirjasto on jaettu viiden eri kategorian funktioihin, joiden avulla huolehditaan kokonaisuudessaan verkkosovelluksen WebSocket-perusteisesta tietoliikenteestä (MDN web docs, 2020).

Konstruktori	
WebSocket(url [, protocols])	Palauttaa uuden WebSocket-olion.
Vakiot	
WebSocket.CONNECTING	value: 0
WebSocket.OPEN	value: 1
WebSocket.CLOSING	value: 2
WebSocket.CLOSED	value: 3
Ominaisuudet	
WebSocket.**binaryType**	Yhteyden käyttämä binääridatatyyppi.
WebSocket.**bufferedAmount**	Puskuroidun datan määrä.
WebSocket.**extensions**	Palvelimen valitsemat laajennokset.
WebSocket.**onclose**	Tapahtumankäsittelijä yhteyden sulkemiselle.
WebSocket.**onerror**	Tapahtumankäsittelijä, jota kutsutaan virhetilanteissa.
WebSocket.**onmessage**	Tapahtumankäsittelijä viestin vastaanottamiselle.
WebSocket.**onopen**	Tapahtumankäsittelijä yhteyden avaamiselle.

WebSocket.**protocol**	Serverin valitsema aliprotokolla.
WebSocket.**readyState**	Yhteyden nykyinen tila.
WebSocket.**url**	WebSocketin absoluuttinen osoite.
Metodit	
WebSocket.**close([code [, reason]])**	Sulkee yhteyden.
WebSocket.**send(data)**	Asettaa datan lähetysjonoon.
Tapahtumat	
close – *onclose*	Suoritetaan, kun yhteys WebSocketiin suljetaan.
error – *onerror*	Suoritetaan, kun WebSocket-yhteys joudutaan sulkemaan virheen vuoksi, esimerkiksi jos datan lähetys epäonnistuu.
message – *onmessage*	Suoritetaan, kun vastaanotetaan viesti WebSocketin kautta.
open – *onopen*	Suoritetaan, kun uusi WebSocket-yhteys on avattu.

Taulukko 3-1: WebSocket API funktiot. (MDN Web Docs)

WebSocket-yhteyttä hyödyntävässä sovelluksessa on aina toteutettava sekä *web-selaimella* (client) toimiva käyttöliittymä eli varsinainen web-sovellus että palvelinpuolen sovellus. Yksinkertaisimmassa mallissa tämä on toteutettu integroimalla asiakassovelluksen WebSocket-koodi suoraan *HTML*-tiedostoon *client.html* ja palvelimen koodi omaan erilliseen JavaScript-tiedostoon *server.js*, joka suoritetaan esimerkiksi *Nodejs*-ohjelmalla. Tällöin sekä asiakaskoneena että palvelinkoneena toimii sama tietokone (*localhost eli 127.0.0.1*). *Luvussa 3.1.1* tutustutaan *taulukossa 3-1* esitettyyn muutamaan yhteydenmuodostuksessa keskeisimpään funktioon.

3.1.1 WebSocket-palvelinsovellus ja Nodejs

Tässä luvussa käsitellään JavaScript-kieliset toteutukset WebSocket-yhteyden muodostamiseksi sekä viestien lähettämiseen ja vastaanottamiseen. Palvelinsovelluksessa käytetään kuitenkin *Nodejs*:n *ws*-API-kirjastoa, joka on sisällöltään hyvin samankaltainen kuin WebSocket API. Yksinkertaisessa WebSocket-sovelluksessa määritellään tavallisesti viisi eri vaihetta ja neljä tapahtumaa (*Taulukko 3-1*). Ensimmäisessä vaiheessa luodaan WebSocket-olio. Toisessa vaiheessa määritellään mitkä funktiot suoritetaan, kun uusi asiakas avaa yhteyden (*open*). Tällöin yleensä palvelin lähettää asiakkaalle jonkinlaisen vastauksen yhteyden muodostamisesta. Kolmannessa vaiheessa määritetään viestitapahtumat (*message*), eli varsinainen asiakkaan ja palvelimen välinen tiedonsiirto, joka voi olla joko tekstimuotoista tai binäärimuotoista. Neljännessä luodaan virheisiin varautumisen toiminnallisuus ja kuudennessa yhteyden sulkemisen periaatteet eli miten toimitaan, kun asiakkaalta tulee yhteyden sulkemisen pyyntö, tai yhteys joudutaan sulkemaan virheen seurauksena. WebSocket-yhteyden muodostamisen ja ylläpidon vaiheet palvelimella ovat:

1. Yhteysolion luominen ja määrittely
2. Yhteyden avaaminen (*open*)
3. Viestien lähetys ja vastaanottaminen (*message*)
4. Virheiden käsittely (*error*)
5. Yhteyden sulkeminen (*close*)

Koodi 3-2 kuvaa pelkistettyä palvelimella suoritettavaa WebSocket-sovellusta. Aluksi JavaScriptin *require('module_name')*-metodilla sisällytetään sovelluksessa käytettävät moduulit *http* ja *ws*. *Http*-moduulin metodilla *createServer* luodaan ensin HTTP-palvelin WebSocket-yhteensopivuutta varten. Kohdassa kolme niin sanottu *callback*-funktio eli funktio, jota kutsutaan *server*-olion luomisen jälkeisenä toimenpiteenä, lähettää vastauksena *asiakkaalle* (client) *"I am connected"*-viestin. Kohdassa neljä *ws*-moduulin *websocket*-viitteellä suoritetaan metodikutsu *Server({server})*, joka saa parametrinaan edellä määritellyn *http*-serverin objektina, ja luodaan WebSocket-*server*-olio *wss*. Kohdissa 5 ja 6 kutsut *wss.on(event, callback)* asettavat käsittelijät tapahtumille *headers* ja *connection*. Tässä otsakkeet vain tulostetaan näytölle, mutta *headers*-tapahtumankäsittelijän avulla niitä voitaisiin esimerkiksi muokata ennen vastausotsakkeen lähettämistä osana kättelyä. Tapahtuma *connection* vastaanotetaan asiakkaalta, kun kättely on valmis. Parametri *request* on asiakkaan lähettämä HTTP GET-pyyntö. Kohdassa 7 metodi *send("...")* lähettää asiakkaalle viestin WebSocket-yhteyden avautumisesta. Kohdassa 8 asetetaan lisäksi yhteyskohtainen käsittelijä tapahtumalle *message*, joka tulostaa asiakkaan lähettämät viestit konsoliin. Kohdassa 9 *http*-palvelimen käyttämä verkkoportti määritetään funktiolla *server.listen(port_number)*, jossa "*server*" on vapaasti valittu tarkoitukseen sopiva muuttujanimi luodulle *http*-palvelinoliolle. Tämä voitaisiin toki tehdä myös serverin luonnin yhteydessä.

```
const http = require('http'); ❶
const websocket = require('ws');
const server = http.createServer( (request, response) => { ❷
     response.end("I am connected!"); ❸
)};
const wss = new websocket.Server({server}); ❹
wss.on('headers', (headers, request) => { ❺
     console.log(headers);
)};
wss.on('connection', (ws, request) => { ❻
    ws.send("Welcome to the websocket server!!"); ❼
    ws.on('message', (msg) => { ❽
          console.log(msg);
    });
)};
server.listen(8181); ❾
```

Koodi 3-2: Palvelimen Nodejs-sovellus, server.js (Bunch, 2020)

Koodi 3-2 ajetaan palvelimelle asennetulla *Nodejs*-ohjelmalla. *Nodejs* on JavaScript pohjainen käyttöympäristö, jonka avulla voi luoda verkkopalvelimia ja verkotettuja sovelluksia. *Nodejs* sisältää useita sisäänrakennettuja kirjastoja ja se on kevyt ja tehokas käyttää (Zhangling & Mao, 2012). *Nodejs*-sovellus ajetaan esimerkiksi tietokoneen komentoriviltä komennolla *node server.js*, jonka jälkeen se jää odottamaan yhteyspyyntöjä: tässä esimerkissä osoitteeseen *ws://localhost:8181*.

3.1.2 WebSocket-selainsovellus

Tässä yksinkertaisuuden vuoksi WebSocket-yhteyden avaava koodi on upotettu HTML-sivulle. Normaalisti tällaista ohjelmointitapaa ei käytännöllisistä syistä suositella, vaan JavaScript-koodi tulisi aina sijoittaa omaan tiedostoonsa. Web-sovelluksissa käyttöliittymänä toimii kuitenkin aina selain ja JavaScript-tiedostot suoritetaan jollekin tietylle web-sivulle eli käytännössä HTML tiedostoon navigoimalla. Siksi tässä esimerkissä käytetään tiedostopäätettä *html*. Ensimmäisenä luodaan WebSocket-olio, jonka konstruktorille annetaan parametrina sen palvelimen URL-osoite, johon yhteys muodostetaan. Kun käytetään paikallista WebSocket-palvelinta eikä yhteyden välityspalvelinta tarvita, yhteysolio voidaan muodostaa seuraavaa selaimen natiivia WebSocket-oliota käyttämällä:

```
const ws = new WebSocket(“ws://localhost:8181”);
```

Konstruktorille voidaan antaa toisena parametrina myös aliprotokollia, kuten käytettävän dataformaatin (*luku 4.1*). Tätä luotua uutta *ws*-oliota voidaan käyttää nyt *taulukossa 3-1* esiteltyjen tapahtumien kuunteluun. JavaScriptissä tapahtumankuuntelija voidaan lisätä joko käyttämällä *addEventListener()*-metodia tai sitten viittaamalla suoraan oliomuuttujaan tapaan *ws.onopen.* (Lompardi, 2015) *Koodissa 3-2* käytettiin JavaScriptin vaihtoehtoista syntaksia, mikä toimisi yhtä hyvin tässä, eli tapahtumien kuuntelijat voidaan asettaa myös muodossa:

```
ws.on(’open’, (event) => { … } );
```

Koodista 3-3 voidaan havaita, että selaimen WebSocket API on periaatteiltaan hyvin samankaltainen kuin palvelimen *koodi 3-2*, vaikka syntaksi ja erityisesti tapahtumien nimet hieman eroavatkin.

```
const ws = new WebSocket("http://localhost:8181");
ws.onerror = (error) => {
     console.log("An error occurred: " + error);
};
ws.onclose = () => {
     console.log("Connection closed!");
};
```

```
ws.onopen = () => {
     ws.send("I’m so excited I’m connected! It is like a Christmas!");
});
ws.onmessage = (msg) => {
     console.log(msg);
};
```

Koodi 3-3: Selaimella suoritettava JavaScript-koodi. (Bunch, 2020; Grigorik, 2013, s. 286; Lompardi, 2015)

Kun käyttäjä ajaa *koodin 3-3* selaimellaan, tulostuu palvelimen konsoliin *tuloste 3-4*. Tämä on ohjelmoitu *koodin 3-2* kohdassa 5, jonka funktiomäärittelyssä on asetettu *headers*-tapahtumankäsittelijä tulostamaan otsaketiedot metodikutsulla *console.log(headers)*. Nämä otsaketiedot lähetetään vastauksena asiakkaalta saatuun WebSocket-yhteyspyyntöön. Alimmainen rivi tulostuu, kun palvelinsovelluksen *message*-tapahtumankäsittelijä (*koodi 3-2, kohta 8)* vastaanottaa selainsovelluksen metodin *send(...)* lähettämän viestin. Tämä on määritelty yhteyden avaamisessa suoritettavassa *onopen*-tapahtuman *callback*-funktiossa (*koodi 3-3*).

```
[
  'HTTP/1.1 101 Switching Protocols',
  'Upgrade: websocket',
  'Connection: Upgrade',
  'Sec-WebSocket-Accept: XGQ51V6wK6Rj7EzPojY+0pVf8Pc='
]
I’m so excited I’m connected! It is like a Christmas!
```

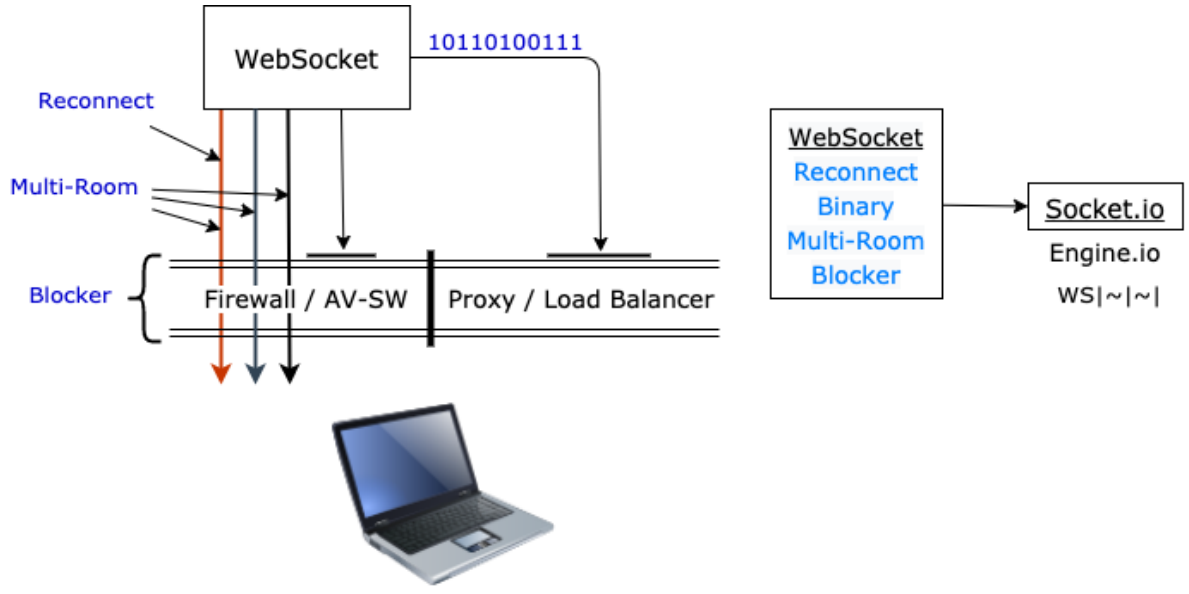
Tuloste 3-4: Palvelimen tuloste. (Bunch, 2020)

Koodikatkelmissa luotiin toimiva kaksisuuntainen ja reaaliaikainen yhteyskanava selaimen ja paikallisen palvelimen välille, mutta se ei sisällä vielä muuta toiminnallisuutta. Yhteenvetona voidaan silti todeta, että WebSocket API on periaatteiltaan huomattavan yksinkertainen ja sen avulla kaksisuuntainen web-sovellus melko helposti toteutettavissa. Sen JavaScript API on myös lähes täysin itsenäinen, eikä sisällä riippuvuuksia muihin kirjastoihin tai moduuleihin. Sovellusten monimutkaistuessa tarvitaan kuitenkin monipuolisempia ohjelmointirajapintoja, jotta säilytetään mielekkään ja ymmärrettävän koodin kirjoittaminen. Joissakin tapauksissa sama sovellus voi vaatia useita rinnakkaisia *yhteyskanavia* (sockets). Käytännössä kaikki yleisimmät selaimet ovat asettaneet samasta alkuperästä muodostettavien yhteyksien maksimilukumäärän kuuteen (Grigorik 2013, 253). WebSocket API ei sisällä myöskään toiminnallisuutta yhteyden ylläpidon kontrollointiin ja uudelleen muodostamiseen. Käyttäjä voi siis menettää yhteyden palvelimelle tietämättään (Bunch, 2020). Lisäksi palomuurit, virustorjuntaohjelmistot ja välityspalvelimet voivat aiheuttaa ongelmia WebSocket-yhteyden muodostamiselle.

3.1.3 JavaScript-kirjastot Socket.io ja Engine.io

Siinä missä WebSocket API on itsenäinen ja riippumaton ohjelmointirajapinta, *Socket.io* on erilaisten rajapintojen eli moduulien kokoelma. Ohjelmoinnissa moduuleiksi kutsutaan apukirjastoja, joita sisällytetään ohjelmistoprojektiin. Myös *Socket.io* tarjoaa reaaliaikaisen kaksisuuntaisen tapahtumapohjaisen viestinnän koska se käyttää tiedonsiirrossa WebSocketia (Bunch, 2020). Se ei ole kuitenkaan WebSocket-toteutus ja vaikka se käyttää tiedonsiirrossa WebSocketia aina kun mahdollista, ne eivät ole keskenään yhteensopivia. Lisäksi *Socket.io* lisää jokaiseen siirrettyyn pakettiin metadataa. Samoin kuin WebSocket API, myös *Socket.io* sisältää erikseen palvelinpuolen ja selaimen kirjastot (Guillermo, 2014).

Socket.io on suunniteltu lisäämään yhteyden luotettavuutta ja toimintavarmuutta, sillä se huolehtii myös välityspalvelimista, kuormituksen tasaajista, palomuureista ja viruksentorjuntaohjelmistoista. Tätä tarkoitusta varten *Socket.io* tukeutuu kuitenkin *Engine.io* alikirjastoonsa, jonka kautta se käyttää WebSocketia (*luku 3.1.4*). Tämän lisäksi *Socket.io*:n avulla saavutetaan monia muita etuja ja lisäominaisuuksia, kuten yhteyskatkoksien havaitseminen ja automaattinen yhteyden uudelleen avaaminen, binärituki, yhteyksien kanavointi ja suora selainten välinen kommunikointi. (Npm, 2019) *Kaavio 3-5* kuvaa *Socket.io*-kirjaston reaaliaikaiseen ja kaksisuuntaiseen kommunikointiyhteyteen tuomia lisäominaisuuksia WebSocket-rajapinnan päälle.



Kaavio 3-5: Socket.io. (Bunch, 2020)

Engine.io on siis *Socket.io* kirjaston alaisuuteen toteutettu ja sen ytimenä toimiva moduuli, mutta sitä voidaan käyttää myös erikseen. *Socket.io*:n luojan, Guillermon (2014) mukaan *Socket.io*:n elinkaaren aikana on löydetty lukemattomia haittoja luotettaessa HTML5 WebSocket -protokollaan ensisijaisena yhteysmekanismina. Siksi *Engine.io* kehitettiin siten, että se muodostaa yhteyden käyttämällä aina ensin *XHR-polling*-protokollaa ja vaihtaa

yhteystapaa vasta saatavuustestien jälkeen. Tätä kutsutaan WebSocket-kättelyksi, kuten aiemmin todettiin. *Engine.io* suunniteltiin varmistamaan kaksisuuntaisen ja reaaliaikaisen yhteyden maksimaalinen luotettavuus käsittelemällä yhteyden muodostamisen *esteet* (blockers), kuten välityspalvelimet ja palomuurit, sekä skaalautuvuus huomioimalla kuormituksen tasaajat.

4 Käytettävyys ja ominaisuudet

Käytettävyydessä huomioidaan sekä web-ohjelmoinnissa merkittävät ohjelmointirajapintaan liittyvät käytettävyysseikat että myös luotettavuuteen ja vikasietoisuuteen eli toimintavarmuuteen ja tietoturvallisuuteen liittyvät näkökulmat. *Luvussa 3* havaittiin, että erityisesti WebSocket API tarjoaa melko selkeän *tapahtumavetoisen* (event-driven) ohjelmointirajapinnan yhteyden muodostamiseksi selaimen ja palvelimen välille. Luotettavan ja turvallisen web-sovellusohjelmoinnin kannalta on tärkeää, että ohjelmoijilla on käytettävissään riittävän ymmärrettäviä ja selkeitä ohjelmointirajapintoja. Tällöin sovelluskehittäjällä on paremmat mahdollisuudet onnistua varsinaisen sovelluslogiikan toteuttamisessa. Totesimme jo *Socket.io* API:n tuomat edut käytettävyyden kannalta ohjelmoinnissa, sillä se lisää web-sovelluksen luotettavuutta ja toimintavarmuutta muun muassa huolehtimalla välityspalvelimista ja palomuureista ja tarjoaa monia muita lisäominaisuuksia. Järjestelmän kompleksisuuden lisääntymisellä on kuitenkin aina hintansa. Grigorikin (2013, 287) mukaan WebSocket API tarjoaakin parhaan suorituskyvyn ja suosittelee käyttämään erityisesti suorituskykykriittisissä sovelluksissa aina ensisijaisesti natiivia WebSocket API:a jäljennösten sijaan. *Socket.io* JavaScript API mahdollistaa kuitenkin monimutkaisten WebSocket-yhteyttä käyttävien sovellusten kehityksen siten, että ohjelmoijan ei tarvitse kirjoittaa massiivisia koodimääriä. Mitä enemmän ja monimutkaisempia rakenteita koodi sisältää, sitä herkemmin syntyy haavoittuvuuksia ja epävakaita ohjelmistoja, ja virheistä toipuminen kestää huomattavasti kauemmin, koska usein myös bugien eli ohjelmointivirheiden löytäminen hidastuu.

Tässä luvussa tutustumme vielä datan siirtämiseen *kaksisuuntaisessa ja reaaliaikaisessa yhteyskanavassa* (socket), eli käsittelemme *Socket.io*-kirjaston *emit()*-metodin toimintaa datan siirrossa. Sen lisäksi tulemme tietoiseksi yleisimmistä ja tunnetuimmista WebSocket-protokollaan liittyvistä haavoittuvuuksista

4.1 Tiedonsiirto – teksti- ja binääridata

Luvun 3.1.2 koodiesimerkeissä esiteltiin WebSocket API:n tiedonsiirtofunktio *send()* ja sitä kuunteleva selainsovelluksen *onmessage()*-funktio. Vastaava toiminnallisuus voidaan suorittaa *Socket.io* API-kirjaston funktioilla *emit('event', callback)* ja *on('event', callback)*. Tärkein *emit*- ja *send*-funktioiden käytännön ero on siinä, että *emit* saa parametrina lisäksi tapahtuman nimen, jonka avulla se voidaan osoittaa jonkin ennalta määrätyn tyypin kategoriaan kuuluvaksi. Samaan tapaan WebSocket API:n *onmessage(callback)* muuntuu *Socket.io* muotoon *on('message', callback)*. *Socket*-kanavassa tiedonsiirto voidaan toteuttaa joko teksti- tai

binäärimuotoisena. (Socket.io, 2020) Tämän luvun koodikatkelmissa käytetään tietomuotona UTF-8-koodattua tekstidataa. Tyypillisessä web-sovelluksen käyttötapauksessa tietoa käsitellään JSON-formaatissa, siksi esimerkeissä käytettävyyttä on havainnollistettu JavaScript-objektien ja JSON-luokan avulla.

Samoin kuin WebSocket API, *Socket.io* sisältää joukon esimääriteltyjä tapahtumia, mutta sovellusohjelmoija voi määritellä myös omia tapahtumia ja käyttää mielivaltaisia tapahtumanimiä (Socket.io, 2020). *Koodi 4-1* kuvaa palvelimen *Nodejs*-sovelluksen toteutusta, jossa on ohjelmoitu tapahtumalle *connection* datan lähettämisen ja vastaanottamisen toiminnallisuutta. *Connection* on yksi *Socket.io*:n esimääritellyistä tapahtumista, joka kuvaa palvelimelle yhdistettyä asiakasta eli selainta. Sen *callback*-funktio saa parametrinaan *socket*-olion, joka kuvaa luotua *socket*- eli TCP/IP-yhteyttä. Jokaiselle yhteydelle on määritelty, että yhteydenmuodostuksen jälkeen lähetetään tapahtuma *”messageFromServer”* asiakkaalle ja lisäksi asetettu tapahtumankuuntelija tapahtumalle *”messageToServer”*, jota odotetaan saapuvaksi asiakkaalta. Näistä jälkimmäisen tapahtuman *callback*-funktio suoritetaan vain, jos selainsovellukseen on määritelty tapahtuma *”messageToServer”*. (Bunch, 2020)

```
socketio.on('connection', (socket) => {
     socket.emit('messageFromServer', JSON.stringify({data:"payload"}));
     socket.on('messageToServer', (dataFromClient) => {
          var jsonObject = JSON.parse(dataFromClient);
          console.log(jsonObject.data);
     });
)};
```

Koodi 4-1: Palvelimen Socket.IO toteutus. (Bunch, 2020)

JSON-luokan funktiota *stringify()* käytetään JavaScript-objektien muuntamiseksi tekstiformaattiin ja *parse()*-funktiolla vastaavasti tekstimuotoinen JSON-data takaisin JavaScript-objektiksi: {data: ”payload”} => ”{data: ”payload”}”.

```
const socket = io("http://example.com/socket");

socket.on('messageFromServer', (dataFromServer) => {
     var jsonObject = JSON.parse(dataFromServer);
     console.log(jsonObject.data);
     socket.emit('messageToServer', JSON.stringify({data:"Data received"}));
)};
```

Koodi 4-2: Selaimen Socket.IO toteutus. Funktio ”io” on sisällytettävä. (Bunch, 2020)

Koodi 4-2 kuvaa selainsovelluksen *Socket.io* toteutusta, joka kuuntelee tapahtumaa *”messageFromServer”*. Kun kyseinen tapahtuma ja siihen liittyvä datapaketti vastaanotetaan palvelimelta, suoritetaan datan muunnos tekstistä JSON-objektiksi ja tulostetaan sitten selaimen konsoliin. Lopuksi lähetetään funktiolla *emit()* vahvistusviesti palvelimelle tekstiksi muunnettuna JSON-objektina.

WebSocket-yhteyden ei tarvitse huolehtia vastaanotetun datan puskuroinnista ja jäsentelystä tai rekonstruoinnista. Esimerkiksi jos palvelin lähettää 1MB datapaketin, asiakassovelluksen tapahtumankäsittelijän callback-funktio suoritetaan vasta, kun koko viesti on asiakkaan saatavilla. WebSocket-protokolla ei myöskään tee oletuksia tai aseta rajoituksia datapaketeille ja se tarkkailee ainoastaan datapaketin kokoa ja datatyyppiä erottaakseen UTF-8-koodatun tekstin ja binääridatan. (Grigorik, 2013, 288)

Tässä luvussa osoitettiin, miten konkreettista dataa lähetetään ja vastaanotetaan WebSocket-protokollaa hyödyntävää ohjelmointirajapintaa käyttämällä. Huomioitavaa kuitenkin on, että *Socket.io* ei ole WebSocket-toteutus, minkä vuoksi ne eivät ole yhteensopivia. Tämä tarkoittaa sitä, että selain- ja palvelinsovellus on molemmat toteutettava *Socket.io* API:lla. WebSocket API -asiakas ei siis voi yhdistää *Socket.io* palvelimelle, eikä päinvastoin. (Socket.io, 2020)

4.2 Luotettavuus- ja turvallisuusnäkökulmia

WebSocket voi tarjota ratkaisun yhteysongelmiin monissa tapauksissa, mutta se ei poista olemassa olevia haavoittuvuuksia. Tämän vuoksi sovelluskehittäjän on edelleen tiedostettava uhkatyypit, joita protokollan käytössä ilmenee ja huomioitava ne toteuttamalla omat tietoturvaratkaisut sovellukseen. Tässä luvussa mainitut tietoturvauhat eivät siis liity pelkästään WebSocket-protokollaan, vaan ovat yleisiä web-sovellusten tietoturvakysymyksiä. WebSocket ei siis ole ratkaisu web-sovellusten tietoturvaongelmiin, kuten Erkkilä (2012) ja Kulshrestha (2013) toteavat. Erkkilä kuitenkin jatkaa, että vaikka on useita avoimia kysymyksiä, niin hyvällä suunnittelulla ja verkkoselainten ja verkkopalvelujen asianmukaisella toteutuksella riskitasoa voidaan pienentää.

4.2.1 Haavoittuvuudet

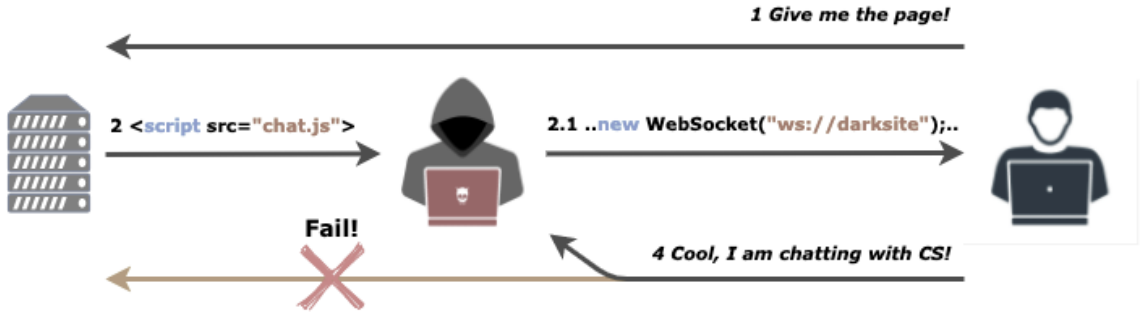
Monissa aiemmissa tutkimuksissa on nimetty viisi yleistä haavoittuvuutta, jotka web-sovelluskehittäjän on huomioitava. *Palvelunestohyökkäys* (Denial of Service) on näistä yksi tunnetuimmista. Tässä hyökkäyksessä tarkoituksena on ruuhkauttaa palvelin siten, että se joko kaatuu kokonaan tai palvelun käyttö estyy muodostamalla mielivaltainen määrä yhteyksiä palvelimelle. Sovelluskehittäjä voi ottaa tämän huomioon rajoittamalla samasta alkuperästä muodostettavien yhteyksien määrää. Tämä jättää kuitenkin yhä mahdollisuuden edistyneemmille *hajautetuille palvelunestohyökkäyksille* (DDoS, Distributed DoS). WebSocket ei tarjoa ratkaisua tämän haavoittuvuuden poistamiseksi.

WebSocket-protokolla käyttää *todennetun alkuperän* (verified-origin) mekanismia, jonka avulla kohdepalvelin voi päättää mistä lähteestä se sallii yhteydet (Erkkilä, 2012). Koska WebSocket-kehykset eivät sisällä otsakkeita, lähetetään yhteyspyynnön alkuperä eli *origin*-otsake jo WebSocket-kättelyssä. WebSocket-palvelimen vastuulla on siis määritellä, että yhteydenmuodostukset sallitaan vain luotetuiksi katsottavista lähteistä. Erkkilän mukaan

alkuperän todennuksesta huolimatta hyökkääjän on silti aina mahdollista väärentää otsaketta ja yhdistää palvelimelle.

Pyyntöjen väärentämishyökkäyksessä (Cross-Site Request Forgery, CSRF) hyödynnetään kohdepalvelimella olevan sivuston luottamusta käyttäjän selaimeen. WebSocket-palvelimella voidaan kuitenkin estää yhteyspyynnöt, jotka eivät ole *samasta alkulähteestä* (same-origin). Tämä suojaa käyttäjää CSRF-hyökkäyksiltä estämällä kolmansia osapuolia muodostamasta yhteyttä uhrin identiteetillä.

WebSocket-kaappaus (Cross-Site WebSocket Hijacking, CSWSH) -haavoittuvuus on hyvin samankaltainen kuin CSRF, mutta Mei ja Long (2020) luonnehtivat sitä vakavammaksi siitä syystä, että sen avulla hyökkääjällä on kaksisuuntainen *luku-kirjoitus*-kanava eikä sitä voida estää *saman alkuperän todentamismenettelyllä* (same-origin policy). Hyökkäyksessä käytetään hyväksi selaimeen tallennettuja evästeitä. Kun käyttäjä kirjautuu sisään luotetulle web-sivustolle, selain tallentaa todennustiedot evästeisiin ja avaa WebSocket-yhteyden palvelimelle. Jos käyttäjä ei kirjaudu ulos, ennen kuin avaa yhteyden *vahingolliselle sivustolle* (malicious service), avautuu hyökkääjälle laillista yhteyttä vastaava pääsy samalle palvelimelle naamioitumalla uhrin identiteetin taakse evästeiden avulla (Mei & Long, 2020). Nämä niin sanotut vahingollisten palvelujen haavoittuvuudet vaativat kuitenkin aina sitä, että käyttäjä saadaan huijattua hyökkääjän sivustolle, tai että hyökkääjä onnistuu injektoimaan hyökkäyksessä käytettävän skriptin jollekin toiselle luotetulle sivustolle (Erkkilä, 2012). Tämänkaltaisia haavoittuvuuksia kutsutaan myös *cross-site scripting* (XSS) -haavoittuvuuksiksi. Mishra ja Jeysree (2015) huomauttavat, että *XSS*-haavoittuvuudet avaavat monia uusia tietoturva-aukkoja. Sitä hyödyntämällä voidaan esimerkiksi korvata WebSocket-yhteyden *callback*-funktiot, eli funktiot, jotka suoritetaan määriteltyjen tapahtumien yhteydessä (*taulukko 3-1* ja *koodi 3-2)*. Tätä hyödyntämällä, hyökkääjä voi suorittaa lähes mitä tahansa hyökkäyksiä, kuten salakuuntelua ja datan manipulointia (Mishra & Jeysree, 2015). *Kaavio 4-3* kuvaa luotetulle sivustolle ujutettua hyökkääjän vahingollista sivua, jonka seurauksena käyttäjä avaakin yhteyden hyökkääjän sivustolle, mutta kuvittelee muodostaneensa yhteyden aitoon haluamaansa palveluun.



Kaavio 4-3: Yhteyskaappaus, XSS-haavoittuvuus. (Shema, Shekyan & Toukharian, 2012)

WebSocket ei tarjoa mekanismia datan salaukseen ja validointiin tai todennuksiin kuten Erkkilä (2012), Mishra ja Jeysree (2015) osoittavat, joten niiden käsittely jää sovelluskehittäjän

vastuulle. Koska tiedonsiirto WebSocket-yhteyskanavan kautta tapahtuu puhtaana tekstinä tai binäärimuotoisena, tulisi käytännössä kaikissa web-sovelluksissa käyttää aina TLS-suojattua wss://-toteutusta. Sen lisäksi sovelluskehittäjän on otettava huomioon, että *wss* (websocket secure) tarkoittaa ainoastaan turvallista tiedonsiirtoa, mutta ei takaa itse sovelluksen turvallisuutta. Tästä lisää seuraavassa luvussa.

4.2.2 URL-skeemat

Selainsovelluksesta muodostetaan yhteys WebSocket-palvelimelle käyttämällä joko *ws* tai *wss* URL-skeemaa. *Wss* tulee siis sanoista *websocket secure,* mikä tarkoittaa, että se käyttää salattua TCP+TLS yhteyskanavaa (Grigorik, 2013, 287). Käytännössä tämä tarkoittaa sitä, että salaamattoman puhtaan tekstipohjaisen *ws*-skeeman käyttämän oletusportin 80 sijaan *wss*-yhteys käyttää porttia 443 (Lompardi, 2015). WebSocket-yhteys käyttää siis samoja portteja kuin HTTP ja HTTPS, jonka ansiosta se läpäisee palomuurit ja välityspalvelimet ongelmitta (Kulshrestha, 2013). Mutta miksi näitä URL-skeemoja *ws:// ja wss://* tarvitaan? Grigorikin (2013, 287) mukaan tyypillisin käyttötapaus WebSocketille on luoda optimoitu, kaksisuuntainen yhteyskanava selaimella ja palvelimella suoritettavien sovellusten välille, mutta WebSocket-yhteyttä voidaan käyttää myös muualla kuin HTTP-perusteisissa selainsovelluksissa. Tämän vuoksi WebSocket-laajennoksia kehittävä HyBi-työryhmä päätti ottaa käyttöön URL-skeemat.

4.2.3 Vikasietoisuus

Vikasietoisuus on tärkeä osa web-sovelluksen käytettävyyteen ja luotettavuuteen liittyvässä keskustelussa. Kuten olemme todenneet, WebSocket-yhteydessä tiedonsiirto tapahtuu puhtaassa tekstiformaatissa tai binäärimuotoisena. Lisäksi selaimen ja palvelimen tulee molemminpuolisesti vahvistaa vastaanotetun datan eheys ja sulkea yhteys, jos sen formaatti ei ole asianmukainen. Palvelimen ei koskaan tulisi sokeasti luottaa asiakkaalta tulevaan dataan. Useimmissa nykyaikaisissa verkkosovelluksissa käytetään *JavaScript Object Notation* (JSON) -datamuotoa, joka on turvallisinta jäsentää käyttämällä JavaScript-kirjaston *JSON.parse* metodia. (Kulshrestha, 2013)

Vaikka turvallisuus on tärkeä osa luotettavaa ja käyttökelpoista web-sovellusta, olisi muistettava, että se ei saa vaikuttaa oleellisesti palvelujen saatavuuteen ja käytettävyyteen. WebSocket-yhteys voi sulkeutua myös monista tahattomista syistä, kuten sähkökatkoksissa tai reitittimen uudelleen käynnistyksessä, järjestelmien päivityksissä ja monissa vastaavissa tapauksissa. Yhteyskatkoksista tulisi kuitenkin toipua ja pyrkiä palauttamaan yhteys asiakkaan ja palvelimen välille heti kun mahdollista. *Luvussa 3.1.3* totesimme, että *Socket.io* API:a käyttävä sovellus pyrkii aktiivisesti palauttamaan katkenneen yhteyden selaimen ja palvelimen välille. Tällainen ominaisuus lisää järjestelmän vikasietoisuutta merkittävästi silloin, kun yhteyskatkokset johtuvat järjestelmän ulkopuolisista syistä.

5 Johtopäätökset

WebSocket API tarjoaa yksinkertaisen ja luotettavan suorituskyvylle optimoidun ohjelmointirajapinnan kaksisuuntaista ja reaaliaikaista teknologiaa käyttäville verkkosovelluksille. WebSocket-teknologiaa hyödyntävien ohjelmointirajapintalaajennosten, kuten *Socket.io*:n avulla voidaan parantaa yhteyden luotettavuutta ja turvallisuutta ilman, että käytettävyys ja lähdekoodin luettavuus hankaloituvat. Lisäksi tiedonsiirto on helppo toteuttaa luettavassa tekstimuodossa JSON-objektina, joka on erittäin käytännöllinen JavaScript-tietorakenne. Tämän tyyppinen tiedonsiirto on kuitenkin haavoittuva, koska se on luettavassa muodossa, mutta käyttämällä TLS-suojattua yhteyttä voidaan rajoittaa tietoliikenteen vakoilua.

Aiemmat tutkimukset osoittavat, että WebSocket ei luo merkittäviä uusia haavoittuvuuksia, mutta ei myöskään poista olemassa olevia. Tämä johtuu siitä, että se käyttää edelleen samoja mekanismeja kuin HTTP, koska se suunniteltiin yhteensopivaksi protokollaksi. WebSocket-kättely on kriittinen, sillä itse WebSocket-yhteys ei sisällä otsakkeita tai muuta metadataa, jolla käyttäjä ja selain todennetaan turvalliseksi. *XSS*-haavoittuvuuksiin luettava *Cross-Site WebSocket Hijacking* muodostaa uudenlaisen erityisesti WebSocket-yhteyksiä koskevan tietoturvaongelman, johon tulee varautua protokollaa hyödyntävissä web-sovelluksissa. Sovelluskehittäjän on siis edelleen huomioitava turvallisuusnäkökulmat sovelluksissa. Koska WebSocket API:n ja sen päälle rakennettujen rajapintojen toteutuksessa ei ole huomioitu lähes lainkaan tietoturvaongelmia, se jättää vielä paljon vastuuta tiedonsiirron turvallisuudesta ohjelmistokehittäjille ja lopulta ohjelmien käyttäjille eli asiakkaille.

Tämän tutkielman tavoitteena oli nostaa esille tärkeimmät WebSocket API:in liittyvät käytettävyysongelmat ja löytää niihin ratkaisuja. Havaitsimme monia käytettävyyteen liittyviä epäkohtia, joita pelkistetyssä WebSocket-ohjelmointirajapinnassa ei ole huomioitu. Näitä ovat muun muassa yhteensopivuusongelmat, välityspalvelimet, useiden samanaikaisten yhteyksien hallinta, tietoturvahaavoittuvuudet, palomuurit, kolmansien osapuolten virustorjuntaohjelmistot ja yhteyden tilan tarkkailu sekä katkenneen yhteyden palautus. Osoitimme myös, että kyseistä teknologiaa hyödyntävää *Socket.io*-rajapintalaajennosta käyttämällä saavutetaan lukuisia käytettävyysparannuksia ja voidaan ratkaista suurin osa edellä mainituista ongelmista säilyttäen lähes identtinen rajapinta sovelluskehittäjille. Kaikki mainitut käytettävyyttä parantavat ominaisuudet lisäävät kuitenkin verkon kuormitusta ja vaativat lisäresursseja laitteistolta. Siksi erityisesti suorituskykykriittisissä sovelluksissa tulisi aina käyttää puhdasta WebSocket-ohjelmointirajapintaa.

WebSocket-protokollan suorituskykyä on tutkittu lukuisissa eri tutkimuksissa, mutta miten paljon erilaisten käytettävyyttä parantavien ominaisuuksien käyttöönotto vaikuttaa verkkoliikenteen määrään eli verkon kuormitukseen ja latenssiin? Jatkotutkimuksia tulisikin tehdä siitä, mikä hinta käytettävyyden parantamisesta joudutaan maksamaan.

Lähdeluettelo

Agarwal, S. (2012). Real-time web application roadblock: Performance penalty of HTML sockets. *Paper presented at the 2012 IEEE International Conference on Communications (ICC),* 1225-1229. doi:10.1109/ICC.2012.6364271

Bunch, R. (2020). Socket.io with websockets. *Udemy*. Retrieved from https://www.udemy.com/share/101XA8AksbdldTTXg=/. Viitattu 26.10.2020

Erkkilä, J. (2012). Websocket security analysis. *Aalto University School of Science*, 2-3.

Grigorik, I. (2013). High performance browser networking. *North Sebastopol*, *California: O'Reilly*. 285-305

Guillermo, R. (2014). Socket.io. *GitHub*. Retrieved from https://github.com/socketio/socket.io. Viitattu 27.10.2020

Hribernik, M. & Kos, A. (2020). Secure WebSocket based broker and architecture for connecting IoT devices and web-based applications. *IPSI Bgd Transactions.* Retrieved from: http://ipsitransactions.org/journals/papers/tar/2020jan/p2.pdf

Karla, T. & Tarnawski, J. (2019). Soft real-time communication with WebSocket and WebRTC protocols performance analysis for web-based control loops. Paper presented at the *2019 24th International Conference on Methods and Models in Automation and Robotics (MMAR),* 634-639. doi:10.1109/MMAR.2019.8864680

Kulshrestha, A. (2013). An empirical study of HTML5 websockets and their cross-browser behavior for mixed content and untrusted certificates. *International Journal of Computer Applications, 82(6)*.

Lombardi, A. (2015). WebSocket. *Sebastopol, CA: O'Reilly*.

Lubbers, P. & Greco, F. (2020). HTML5 WebSocket: A Quantum Leap in Scalability for the Web. *Kaazing Corporation*. Retrieved from http://websocket.org/quantum.html

MDN Web Docs. WebSocket. Retrieved from https://developer.mozilla.org/en-US/docs/Web/API/WebSocket. Viitattu 25.10.2020

Mei, W. & Long, Z. (2020). Research and Defense of Cross-Site WebSocket Hijacking Vulnerability. Paper presented at the *2020 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA), Dalian, China, 2020*, pp. 591-594. doi:10.1109/ICAICA50127.2020.9182458.

Mishra, P. & Jeysree, J. (2015). Application Research and Penetration Testing on WebSocket Technology. *International Journal of Science and Research (IJSR),* ISSN, pp. 1362-1364

Npm (2019). Socket.io. Retrieved from https://www.npmjs.com/package/socket.io. Viitattu 27.10.2020

Pimentel, V. & Nickerson, B. G. (2012). Communicating and displaying real-time data with WebSocket. *IEEE Internet Computing*, vol. 16, no. 4, pp. 45-53, July-Aug. 2012, doi: 10.1109/MIC.2012.64.

Shema, M., Shekyan, S., & Toukharian, V. (2012). Hacking with webSockets. Retrieved from https://bit.ly/38SmPxt.

Socket.io. Server and Client API. Retrieved from https://socket.io/docs/. Viitattu 28.10.2020

Soweto, B., Christian, Gunawan, F. E., Diana, & Kusuma, I. G. P. (2019). Websocket to support real time smart home applications. *Procedia Computer Science, 157*, 560-566. doi:https://doi.org/10.1016/j.procs.2019.09.014

Srinivasan, L., Scharnagl, J., & Schilling, K. (2013). Analysis of WebSockets as the new age protocol for remote robot tele-operation. *IFAC Proceedings Volumes, 46(29)*, 83-88. doi:https://doi.org/10.3182/20131111-3-KR-2043.00032

Zhangling, Y. & Mao, D. (2012). A real-time group communication architecture based on websocket. *International Journal of Computer and Communication Engineering, 1*(4), 408.