TAMPEREEN TEKNILLINEN YLIOPISTO

VILLE KUOPPALA

TAPAHTUMAPOHJAINEN JAVASCRIPT-OHJELMOINTI NODE.JS:LLÄ

Kandidaatintyö

Tarkastaja: lehtori Tero Ahtee

TIIVISTELMÄ

VILLE KUOPPALA: Tapahtumapohjainen JavaScript-ohjelmointi Node.js:llä
Tampereen teknillinen yliopisto
Kandidaatintyö, 25 sivua
Marraskuu 2014
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotekniikka
Työn tarkastaja: lehtori Tero Ahtee

Avainsanat: Node.js, tapahtumapohjainen JavaScript-ohjelmointi, web-ohjelmointi

Node.js on yksisäikeinen, tapahtumapohjainen ja lukkiutumaton sovellusalusta, jolla voidaan kehittää nopeita ja skaalautuvia web-sovelluksia. Node.js ratkaisee monisäikeisten sovellusalustojen ongelmia, joita ovat esimerkiksi suorituskontekstin säilyttäminen ja rinnakkaisuuden hallinta.

Node.js:n tapahtumapohjaisuus perustuu yksisäikeiseen tapahtumasilmukkaan, jonka suorituskyvyn takaa tehokas Googlen V8 JavaScript -moottori. Tapahtumasilmukan päätehtävä on skeduloida tehtäviä tapahtumajonoon ja laittaa tapahtumia suoritukseen tapahtumajonosta. Node.js:llä ohjelmointi tapahtuu päästä päähän JavaScriptillä, jonka klosuurit ja first-class-funktiot helpottavat tapahtumapohjaisuuden toteutusta.

Node.js:ssä on mahdollista emittoida ja kuunnella eri tyyppisiä tapahtumia. Sovelluskehittäjä voi emittoida haluamansa tyyppisen tapahtuman EventEmitter-olion emit-metodilla, ja eri tyyppisiin tapahtumiin voidaan liittää tapahtumankuuntelijoita Node.js:n on-, addListener- tai once-metodeilla. Myös sovellus itse voi emittoida tapahtumia, joihin voi tavallisesti liittää tapahtumankuuntelijoita. Yleisiä tapahtumia ovat esimerkiksi tietokantakyselyn valmistuminen tai käyttäjän suorittama pyyntö web-sovellukseen.

Node.js on hyvin skaalautuva ja pystyy hallitsemaan useita yhtäaikaisia pyyntöjä. Node.js käyttää JavaScriptiä sekä palvelin- että asiakaspäässä, jolloin sovelluskehittäjien ei tarvitse osata useita ohjelmointikieliä, mikä helpottaa työskentelyä. Node.js:lle on tarjolla runsaasti kolmannen osapuolen kirjastoja ja niiden hallintaan Node.js:n mukana tulee npm-paketinhallintajärjestelmä. Haasteena Node.js-sovelluksissa on suorituskyky staattisia tiedostoja tarjottaessa ja ohjelmakoodiin continuation-passing-tyylin takia syntyvät callback-pyramidit, jotka heikentävät sovelluskoodin luettavuutta.

SISÄLLYS

1 Johdanto....1
2 Tapahtumapohjainen ohjelmointi yleisesti....2
3 Node.js....4
3.1 Yleisesti....4
3.2 Käyttötarve....4
4 Tapahtumapohjaisuuden toteutus Node.js:ssä....6
4.1 JavaScript....6
4.2 Callback-funktio....6
4.3 Asynkroninen suoritustapa....7
4.4 Tapahtumasilmukka....9
4.5 Tapahtuman emittointi....11
4.6 Tapahtumankuuntelija....12
5 Node.js:n edut....16
5.1 Suorituskyky ja skaalautuvuus....16
5.2 Edut JavaScriptistä....18
5.3 Muut edut....18
6 Tapahtumapohjaisen ohjelmoinnin haasteet Node.js:ssä....20
7 Yhteenveto....24
Lähteet....25

TERMIT JA NIIDEN MÄÄRITELMÄT

Asynkroninen suoritustapa	Sovelluskoodin suoritustapa, jossa suoritusjärjestys ei aina ole lineaarinen.
Callback	Funktio, joka annetaan argumenttina jollekin toiselle funktiolle ja se aktivoituu sopivan ajan kuluttua.
Continuation-passing-tyyli	Ohjelmointityyli, jossa funktioille annetaan toinen funktio ylimääräisenä argumenttina. Argumenttina annettu funktio aktivoituu, kun alkuperäinen funktio on saanut suorituksensa valmiiksi.
First-class-funktio	Funktio, joka voidaan välittää argumenttina toiselle funktiolle.
I/O	Input/output.
Klosuuri	JavaScriptin tapauksessa klosuurilla voidaan muodostaa olion kaltainen funktio, jolla voi olla yksityisiä muuttujia.
Lukkiutumaton sovellus	Lukkiutumaton sovellus suorittaa sovelluskoodia koko ajan, eli sovellus voi jatkaa suoritustaan esimerkiksi pitkien tietokantakyselyjen aikana.
Node.js:n konsoli	Olio, jonka avulla voidaan tulostaa stdout- ja stderr -virtoihin. Konsolin käyttötarkoitus on yleensä sovelluksen testaaminen.
npm	Node Packaged Modules eli npm on Node.js:n paketinhallintajärjestelmä, jolla voidaan asentaa moduuleita Node.js-sovellukseen.
Palvelin	Palvelinsovellusta suorittava tietokone, joka tarjoaa palveluja asiakkaille.
Rikas web-sovellus	Työpöytäsovellusta muistuttava web-sovellus, jonka suoritus tapahtuu suurimmaksi osaksi selaimessa.
Skedulointi	Seuraavan suoritettavan tehtävän valitseminen.
Tapahtumanemittoija	Olio, joka synnyttää uusia tapahtumia.
Tapahtumankuuntelija	Tapahtumankuuntelija liittää tapahtumankäsittelijäfunktion halutun tyyppiseen tapahtumaan.
Tapahtumankäsittelijä-funktio	Funktio, joka suoritetaan, jos tietyn tyyppinen tapahtuma emittoituu, ja tapahtumakuuntelija on liitetty tapahtumaan.
Tapahtumasilmukka	Ikuinen silmukka, joka skeduloi syntyviä tapahtumia tapahtumajonoon, josta tapahtumat aikanaan suoritetaan.

1 JOHDANTO

Web-sovelluskehittäjät joutuvat käsittelemään usein monia yhtäaikaisia pyyntöjä palvelinta ohjelmoidessaan. Perinteinen ja intuitiivinen lähestymistapa rinnakkaisuuden toteuttamiseen on käyttää monisäikeisiä ohjelmointitekniikoita. Yleisesti monisäikeisiä toteutuksia pidetään helposti ymmärrettävinä, toteutettavina ja hallittavina, mutta myös nopeampina ja tehokkaampina kuin muut ratkaisut. [1]

Monisäikeisyydestä seuraa kuitenkin ongelmia, jotka voivat olla hyvin vaikeasti paikannettavissa ja korjattavissa. Näitä ovat esimerkiksi lukkiutuminen tai jaettujen resurssien suojaaminen ja käyttäminen useasta eri säikeestä. Sovelluskehittäjät menettävät myös kontrollia ohjelmakoodin suoritukseen, koska yleensä monisäikeisissä järjestelmissä käyttöjärjestelmä päättää, mitä säiettä suoritetaan ja kuinka pitkään. [1] Monisäikeiset toteutukset web-sovelluksissa eivät myöskään kaikissa tilanteissa pysty hallitsemaan hyvin suuria määriä yhtäaikaisia pyyntöjä, vaan tietyn kuormituksen jälkeen ne alkavat epäonnistua pyyntöihin vastaamisessa, ellei resurssien määrää kasvateta huomattavasti [2].

Vuonna 2009 European JSConf -konferensissa esiteltiin yksisäikeinen tapahtumapohjainen sovellusalusta Node.js ja se havaittiin jo siellä potentiaaliseksi tulevaisuuden tekniikaksi [3]. Node.js:lle ominaista on lukkiutumaton I/O ja tapahtumapohjaisuus, minkä seurauksena se on hyvin tehokas ja kevyt ratkaisu palvelin- ja asiakaspään ohjelmointiin yksisäikeisyydestään huolimatta [4].

Node.js tarjoaa tehokkaamman ja skaalautuvamman vaihtoehdon toteuttaa web-sovelluksia ja antaa kehittäjille mahdollisuuden hallita paremmin I/O:ta. Node.js:llä toteutettu dynaaminen web-sovellus pystyy myös käsittelemään suurempia määriä yhtäaikaisia pyyntöjä kuin monisäikeiset tekniikat. [4] Useat suuret yritykset, kuten Microsoft, LinkedIn, Walmart, eBay ja Yahoo, ovat jo ottaneet Node.js:n käyttöön sovelluksissaan, mikä osoittaa sen potentiaalia ja asemaa tulevaisuudessa [2].

Tässä työssä esittellään aluksi tapahtumapohjaisuutta yleisesti, jonka jälkeen keskitytään Node.js:n tapahtumapohjaisuuteen ja siihen, kuinka tapahtumapohjaisuus on Node.js:ssä toteutettu. Työssä kerrotaan myös, kuinka Node.js:llä voidaan emittoida tapahtumia ja miten tapahtumiin voidaan lisätä tapahtumankuuntelijoita. Lopuksi esitellään Node.js:n edut ja haasteet web-sovellusten kehittämisessä.

2 TAPAHTUMAPOHJAINEN OHJELMOINTI YLEISESTI

Tapahtumapohjaisessa ohjelmoinnissa suoritusjärjestyksen määräävät tapahtumat. Perinteiseen ohjelmointitapaan verrattuna suoritusjärjestystä ei pystytä määrittämään ennalta, mikä vaikeuttaa sovellusten testaamista. Tapahtumapohjainen ohjelmointi kuitenkin mahdollistaa esimerkiksi graafisten käyttöliittymien toimimisen, mitä voidaan nykypäivänä pitää tärkeänä erilaisille sovelluksille. Yleisimpiä tapahtumia ovatkin käyttäjän synnyttämät tapahtumat, kuten painikkeiden painaminen graafisissa käyttöliittymissä. Myös muut käyttäjästä riippumattomat tapahtumat ovat mahdollisia, esimerkiksi verkkoyhteyden katkeaminen tai tietokantakyselyn valmistuminen. [5]

```
1       alkuluvut = laskeAlkulukuja(10000);
2       tulosta(alkuluvut);
```

Koodilistaus 1: Alkulukujen laskenta perinteisellä sovelluksella.

Koodilistauksessa 1 on esitetty alkulukujen laskenta tavallisesti. Funktio laskee alkuluvut ja tallentaa ne muuttujaan, jonka jälkeen alkuluvut voidaan tulostaa. Ongelma tällaisessa suoritustavassa on alkulukujen laskennan hitaus, jolloin ohjelma lukkiutuu hetkeksi alkulukuja laskiessaan ja tulostaa vasta sen jälkeen syötteensä. Tapahtumapohjaisuudella voidaan ratkaista tämä ongelma. [5]

```
1       laskeAlkulukuja(10000, function(alkuluvut) {
2          tulosta(alkuluvut);
3       });
```

Koodilistaus 2: Alkulukujen laskenta tapahtumapohjaisesti.

Koodilistauksessa 2 on laskettu alkuluvut tapahtumapohjaisesti. Alkulukuja laskevalle funktiolle on annettu parametriksi laskettavien alkulukujen lisäksi tapahtumankäsittelijäfunktio, joka aktivoituu laskeAlkulukuja-funktion suorituksen valmistuttua. Tämän seurauksena alkulukuja voidaan laskea sillä välin, kun ohjelma

suorittaa jotain muuta sovelluskoodia. Tämä tarkoittaa siis sitä, että sovellus on lukkiutumaton. [5]

Tyypillisesti tapahtumapohjaisuus toteutetaan tapahtumasilmukalla, joka on yksisäikeinen ikuinen silmukka. Tapahtumasilmukan tärkeimpänä ja yleensä ainoana tehtävänä on havaita syntyneet tapahtumat ja laukaista tapahtumiin liittyvät tapahtumankäsittelijät. Yksisäikeisen tapahtumasilmukan etuna on se, että ajossa on vain yksi tapahtumankäsittelijä kerrallaan, joka saa suorittaa itsensä loppuun asti ilman keskeytyksiä. Ohjelmoijan ei siis tarvitse huolehtia sovelluksen synkronisoinnista ja mahdollisista rinnakkaisista säikeistä. [5]

3 NODE.JS

3.1 Yleisesti

Node.js on tapahtumapohjainen sovellusalusta lukkiutumattomalla I/O:lla, joka suorittaa ohjelmakoodiaan Googlen V8 JavaScript -moottorilla. Node.js:n sisäänrakennettu tapahtumapohjainen malli ja lukkiutumaton I/O tekevät siitä kevyen ja hyvin tehokkaan työkalun sellaisille reaaliaikasovelluksille, joiden täytyy käsitellä paljon kuormaa. [1] Node.js on toteutettu pääasiassa C:llä, C++:lla ja JavaScriptillä [1] [6], jotta se olisi tehokas ja käyttäisi mahdollisimman vähän muistia [1].

Node.js:llä ohjelmoidaan käyttäen JavaScriptiä sekä palvelin- että asiakaspäässä. Palvelinpuolella vaaditaan JavaScriptiltä paljon suorituskykyä websovelluksen toimivuuden takaamiseksi. Kilpailukykyisen suorituskyvyn mahdollistaa Node.js:n käyttämä Googlen V8 JavaScript -moottori, joka on optimoitu suorittamaan hyvin tehokkaasti JavaScript-ohjelmakoodia. Vaikka Googlen V8 JavaScript -moottori tukeekin pääasiassa JavaScriptin suoritusta selaimessa, pyrkii Node.js sitä käyttämällä prosesseihin, jotka pysyvät ajossa pitkään. JavaScript on luonteva valinta Node.js:n ohjelmointikieleksi, sillä JavaScriptin first-class-funktioiden ja klosuurien takia tapahtumapohjaisuus on helppo toteuttaa. Node.js:n suosion seurauksena JavaScriptistä tuli varteenotettava palvelinpuolen ohjelmointikieli C:n ja Javan rinnalle. [1]

3.2 Käyttötarve

Nykyään verkossa on kiinni hyvin paljon erilaisia laitteita, ja jo pelkästään mobiililaitteissa on suuria eroja tehoissa, ominaisuuksissa tai käyttöjärjestelmissä. Siksi on hyvin tärkeää tehdä web-sovelluksia, jotka toimivat erilaisilla ja eritehoisilla laitteilla yhtä sujuvasti. Tähän tarkoitukseen Node.js on erittäin hyvä työkalu, koska se on kevyt ja koska kaikki ohjelmointi tapahtuu JavaScriptillä, jota suurin osa moderneista selaimista tukee. [2]

Reaaliaikaisuus ja suuri rinnakkaisuus ovat tärkeitä ominaisuuksia nykyaikaisille web-sovelluksille, sillä niiden kuorma voi olla aina yksittäisistä käyttäjistä jopa miljooniin yhtäaikaisiin käyttäjiin asti. Tehokkaan Googlen V8 JavaScript -moottorin, asynkronisen lukkiutumattoman I/O:n ja tapahtumapohjaisuutensa ansiosta Node.js pystyykin palvelemaan hyvin suuria määriä yhtäaikaisia käyttäjiä vain yhtä säiettä käyttämällä. Yksisäikeisyytensä ja vähäisen muistinkäyttönsä vuoksi Node.js:llä

voidaan toteuttaa paljon liikennettä kestävä palvelin vähemmällä palvelinteholla. Vähemmästä palvelintehosta huolimatta Node.js on myös skaalautuvampi ja tehokkaampi verrattuna monisäikeiseen ratkaisuun. [2]

4 TAPAHTUMAPOHJAISUUDEN TOTEUTUS NODE.JS:SSÄ

4.1 JavaScript

JavaScript on prototyyppipohjainen dynaamisesti tyypitetty skriptikieli. Sillä on myös jotain ominaisuuksia funktionaalisista ohjelmointikielistä kuten klosuurit ja first-class-funktiot, jotka ovat keskeisiä Node.js:n tapahtumapohjaisuuden toteutuksen kannalta. [7]

JavaScriptin first-class-funktiot ovat Node.js:lle tärkeitä, koska niitä käytetään tapahtumankäsittelijäfunktiona. Käytännössä first-class-funktio on tavallinen funktio, mutta se voidaan tallentaa muuttujaan ja välittää parametrina kuten mikä tahansa muukin olio tai muuttuja. [7]

Toinen huomattava etu JavaScriptin first-class-funktioista on se, että ne voivat viitata näkyvyysalueensa ulkopuolella oleviin muuttujiin. Kun funktio välitetään argumenttina eteenpäin, se kapseloi suorituskontekstinsa muodostamalla klosuurin, jolloin päästään edelleen käsiksi vapaisiin muuttujiin. [2] Tämä helpottaa Node.js:n tapahtumapohjaisuuden toteutusta, koska ei ole tarvetta välittää informaatiota senhetkisistä tiloista tapahtumankäsittelijäfunktioille, vaan tapahtumankäsittelijäfunktioilla on koko ajan pääsy suorituskontekstiensa muuttujiin. Myös ohjelmointi selkeytyy huomattavasti, koska ei tarvitse huolehtia pääsystä suorituskontekstiin kaikkialla. [1]

4.2 Callback-funktio

Callback-funktio on Node.js:n tapauksessa parametrina välitettävä anonyymi- tai first-class-funktio. Node.js:ssä callback-funktioita käytetään usein myös tapahtumankäsittelijäfunktioina. Esimerkiksi I/O-tehtäviin on mahdollista liittää callback-funktio, jolloin tehtävän suorituksen päättyessä kyseistä callback-funktiota kutsutaan. [6]

Node.js:n käyttämää syntaksia callback-funktioiden välittämiseen kutsutaan continuation-passing-ohjelmointityyliksi ja JavaScript itsessäänkin jo johdattelee tällaiseen ohjelmointitapaan. Continuation-passing-ohjelmointityylissä annetaan contiunation-passing-funktiolle parametrina jokin funktio, ja kun continuation-passing-funktion suoritus valmistuu, kutsutaan argumenttina annettua funktiota. [5]

Koodilistauksen 3 ohjelmakoodi on Node.js:n contiunation-passing-syntaksia ja siinä luetaan tiedoston sisältö, jonka jälkeen tiedoston sisältö tulostetaan konsoliin. Koodilistauksessa 3 funktio function(err, tiedosto) on anonyymi callback-funktio, joka aktivoituu, kun rivin kaksi continuation-passing-funktiona toimiva readFile-metodi on saanut tiedoston luettua.

```
1       var fs = require('fs')
2       fs.readFile('/polku/tiedosto', function(err, tiedosto) {
3       if(err) {
4             throw err;
5          }
6          console.log('tiedoston sisältö', tiedosto.toString());
7       });
```

Koodilistaus 3: Sovellus continuation-passing-syntaksilla, joka lukee tiedoston sisällön ja tulostaa sen Node.js:n konsoliin. Perustuu lähteeseen [5].

Asynkroninen ohjelmointi Node.js:llä on continuation-passing-tyyliä ja siksi se onkin tärkeä osa Node.js:n tapahtumapohjaisuutta. Node.js:n asynkronisuuden vuoksi koodilistauksen 3 sovellus jatkaa suoritustaan sillä välin, kun readFile()-metodi lukee tiedostoa. [5] Palvelin siis voi aloittaa seuraavan pyynnön käsittelyn heti, ja mikäli jokin callbackeista sillä välin aktivoituu, voidaan sen suoritukseen siirtyä pyynnön prosessoinnin jälkeen. Tällöin palvelin ei lukkiudu koskaan toisista pyynnöistä. [3]

4.3 Asynkroninen suoritustapa

Node.js:llä suurin osa ohjelmoinnista on asynkronista [3]. Muista asynkronisista palvelintoteutuksista Node.js eroaa sillä, että se pakottaa sovelluskehittäjät käyttämään asynkronista mallia alusta asti. Asynkroninen suoritustapa on olennaista Node.js:n tapahtumapohjaisuuden toteutuksen kannalta, koska se estää sovelluksen lukkiutumisen I/O-operaatioiden tuloksia odotellessa. Lukkiutumattomuus on ensiarvoisen tärkeää Node.js:n yksisäikeisen luonteen ja tapahtumasilmukan toiminnan kannalta, sillä suoritusta voidaan jatkaa vaikka jonkin I/O-operaation suoritus kestäisikin kauan. [1] Node.js:ään on kuitenkin olemassa joitakin synkronisia funktioita ja kolmannen osapuolen kirjastoja, mutta ”sync”-merkkijono funktioiden nimessä kertoo kehittäjille, että kyseessä on synkronisesti suoritettava funktio. Tällaisia funktioita tai kirjastoja käyttämällä Node.js-sovellus voi lukkiutua, mutta esimerkiksi sovelluksen testaamista synkronisuus saattaa helpottaa. [6]

Asynkronisessa suoritustavassa ohjelmakoodia ei välttämättä suoriteta lineaarisessa järjestyksessä. Suoritusjärjestystä on siis hyvin vaikea ennustaa, sillä yksikään koodirivi ei estä seuraavan rivin suorittamista ja callback-funktiot saattavat aktivoitua milloin

tahansa. Esimerkiksi hidasta tietokantakyselyä suorittavaa riviä ei jäädä odottelemaan, vaan suoritus jatkuu eteenpäin ja tietokantakyselyn callback-funktio aktivoituu heti, kun kysely valmistuu. [3] Koodilistauksessa 4 on esimerkki synkronisesta ohjelmakoodista ja koodilistauksessa 5 asynkronisesta ohjelmakoodista. Molemmissa esimerkeissä sovellus hakee käyttäjät tietokannasta ja tulostaa Node.js:n konsoliin haetut käyttäjät, sekä merkkijonon ”suoritus jatkuu” suorituksen jatkumisen merkiksi.

```
1       var kayttajat = tietokanta.haeKayttajat();
2       console.log(kayttajat);
3       console.log(”suoritus jatkuu”);
```

Koodilistaus 4: Synkroninen esimerkkisovellus.

```
1       tietokanta.haeKayttajat(function(err, kayttajat) {
2          if(err) {
3             throw err;
4          }
5          console.log(kayttajat);
6       )};
7       console.log(”suoritus jatkuu”);
```

Koodilistaus 5: Asynkroninen esimerkkisovellus.

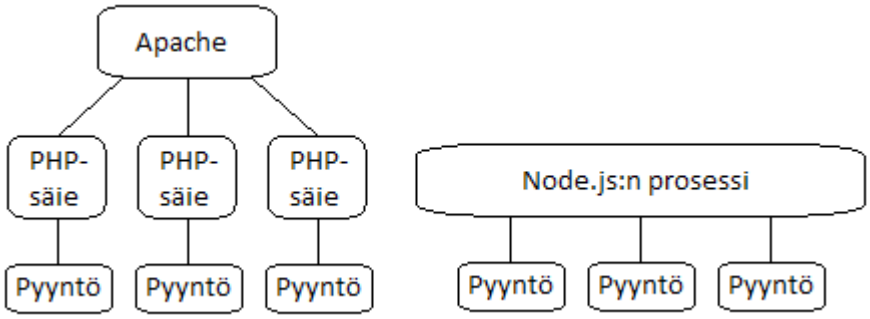
Koodilistauksessa 4 on esitetty synkroninen lukkiutuva ohjelmakoodi, jossa aluksi kayttajat-muuttujaan haetaan tietokannasta lista käyttäjistä. Kyseinen tietokantatapahtuma on hidas suorittaa, minkä seurauksena synkroninen ohjelma lukkiutuu siksi aikaa, että tietokantakysely valmistuu ja käyttäjälista on muuttujassa. Kun sovellus voi jatkaa suorittamistaan, se tulostaa käyttäjälistan ja sen perään merkkijonon ”suoritus jatkuu” suorituksen jatkumisen merkiksi. Tällaisen sovelluskoodin asynkroninen suorittaminen ei olisi järkevää, koska tietokantatapahtuma ei ehtisi määritellä kayttajat-muuttujaan mitään sisältöä ennen kuin sovelluksen suoritus olisi jo seuraavalla rivillä tulostamassa käyttäjälistausta.

Koodilistauksen 5 ohjelmakoodi on asynkroninen. Tässäkin tapauksessa tietokantatapahtuma on hidas, mutta koska sovellus on asynkroninen, sen ei tarvitse jäädä odottelemaan tietokantatapahtuman valmistumista. Tietokannan funktiolle haeKayttajat annetaan parametrina callback-funktio, joka aktivoituu kun käyttäjälistan hakeminen tietokannasta valmistuu. Tällä callback-funktion rekisteröinnillä varmistetaan se, että kayttajat-muuttujassa on käyttäjälista varmasti siinä vaiheessa, kun käyttäjiä aletaan tulostaa. Tietokantatapahtuman valmistumista odotellessa

asynkroninen sovellus jatkaa suoritustaan tavallisesti seuraavasta rivistä, ja konsoliin tulostuu ”suoritus jatkuu” ennen käyttäjälistan tulostumista.

Node.js:n asynkronisen luonteen takia virheiden käsittelyssä täytyy olla erityisen tarkkana. Mikäli poikkeuksia ei käsitellä oikein, koko sovellus voi olla käyttökelvoton. Tavallisesti synkronisessa koodissa virheet käsitellään käyttämällä try, catch ja finally -rakennetta, mutta asynkronisessa tapauksessa virheet käsitellään callback-funktioiden avulla. Esimerkki asynkronisesta virheiden käsittelystä Node.js:llä on koodilistauksessa 3, jossa callback-funktion parametrina annetaan ”err” ja mahdollisen virheen syntyminen tarkastellaan rivillä kaksi. Mikäli virhe tapahtuu ja sitä ei ole käsitelty oikein, syntyy ”uncaught exception” -poikkeus ja sovellus kaatuu. [7]

Asynkronisyys aiheuttaa myös sen, että Node.js:ssä on hallittava jaetun tilan samanaikaisuutta toisin kuin esimerkiksi PHP:n Apachessa. Kuvassa 1 vertaillaan Apachen ja Node.js:n tapaa käsitellä pyyntöjä ja siitä näkyy, kuinka Apache luo jokaiselle pyynnölle uuden PHP-säikeen, kun Node.js sen sijaan ajaa vain yhtä prosessia ja käsittelee tässä yhdessä prosessissa kaikki pyyntönsä. Tästä aiheutuu Node.js:n vaatimus jaetun tilan samanaikaisuuden hallinnalle. [7]



Kuva 1: Apachen ja Node.js:n tavat käsitellä pyyntöjä. Perustuu lähteeseen [7].

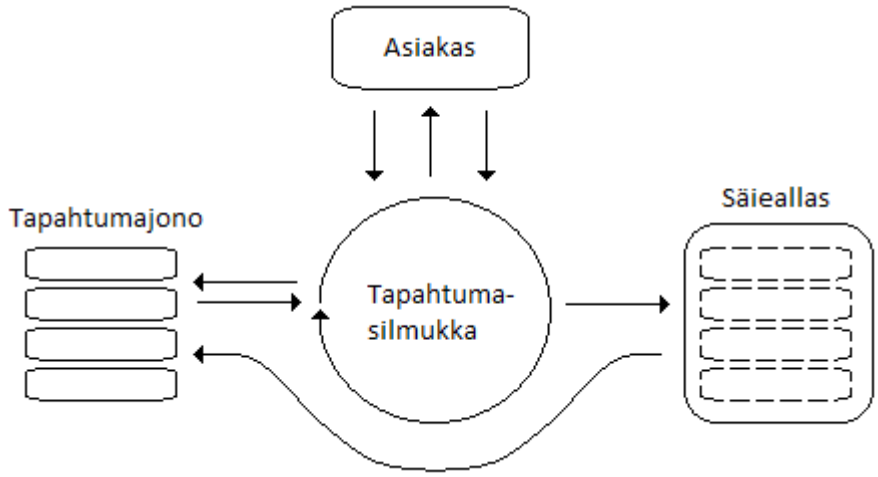
4.4 Tapahtumasilmukka

Node.js:n toteutus on yksisäikeinen, eli se voi suorittaa vain yhtä asiaa kerrallaan, mikä saattaa kuulostaa korkeaa rinnakkaisuutta vaativalle web-sovellukselle hyvin huonolta ratkaisulta. [4] Vaikka ratkaisu onkin ristiriidassa yleisemmälle tavalle käyttää käyttöjärjestelmän useita säikeitä rinnakkaisuuden hallintaan, on se kuitenkin parhaimmillaan hyvinkin tehokas. Tehokkuus on saatu aikaan Node.js:n tapahtumasilmukalla. [6]

Node.js:n tapahtumasilmukka on ikuinen silmukka, jonka avulla saadaan aikaan illuusio siitä, että säikeitä olisi käytössä useampia. Todellisuudessa tapahtumasilmukan suorituksessa on vain yksi tehtävä kerrallaan [6], mutta Node.js:n tapahtumasilmukan toteutus on kuitenkin saatu niin kevyeksi Googlen V8 JavaScript -moottorin avulla, jolloin tehtävät saadaan suoritettua niin nopeasti, ettei yksisäikeisyyttä pysty havaitsemaan hyvin toteutettua web-sovellusta käytettäessä. [7]

Tapahtumasilmukan toiminta perustuu siihen, että se skeduloi syntyneitä tehtäviä tapahtumajonoon. Kun jokin tapahtuma tapahtuu, se asetetaan tapahtumajonon perälle, ja jokaisella tapahtumasilmukan kierroksella jonossa ensimmäisenä ollut tapahtuma suoritetaan, jonka jälkeen se poistetaan jonosta. Jos tämän tapahtuman suorituksen aikana syntyy uusia tapahtumia, ne yksinkertaisesti laitetaan jonon perälle odottamaan suoritusvuoroaan. Kun yksittäinen tapahtuma on suoritettu loppuun kokonaan, tapahtumasilmukka ottaa jonosta seuraavan tapahtuman ja siirtyy sen käsittelyyn. Tätä jatketaan siihen saakka, kunnes koko Node.js-sovellus suljetaan tai tapahtuu jokin virhe, josta ei voida toipua. [6]

Tapahtumasilmukan tapahtumajonossa olevat tehtävät ovat yleensä callbackeja, joita toiset tapahtumat ovat synnyttäneet. Yhden tapahtumasilmukan käyttäminen takaa, että vain yhtä callbackia kutsutaan kerrallaan. Tämä helpottaa ohjelmointia, koska rinnakkaisuusongelmista ei tarvitse olla huolissaan ja ohjelmakoodistakin tulee selkeämpää. [3] Tapahtumasilmukka ei kuitenkaan takaa sitä, ettei ohjelmaa saa jumiutumaan esimerkiksi tapahtumasilmukan sisällä suoritettavalla ikuisella silmukalla tai kovalla laskennalla. [7]



Kuva 2: Tapahtumasilmukan toiminta. Perustuu lähteeseen [8].

Kuvassa 2 asiakas tekee järjestelmään pyyntöjä, jotka tapahtumasilmukka laittaa tapahtumajonoon tai säiealtaaseen suoritettavaksi. Suoritusajaltaan pitkät tehtävät ajetaan rinnakkain säiealtaassa, jolloin vältetään tapahtumasilmukan lukkiutuminen. Säieallas antaa valmistuneet tehtävänsä odottamaan tapahtumajonoon, josta tapahtumasilmukka aikanaan lähettää valmiin tehtävän joko vastauksena asiakkaalle tai uudelleen säiealtaaseen suoritukseen. Tämä ketju toistuu koko sen ajan, kun Node.js-sovellus on ajossa. [6] [8]

4.5 Tapahtuman emittointi

Tapahtuman emittoijiksi kutsutaan olioita, jotka synnyttävät tapahtumia [6]. Node.js:ssä tapahtuman emittoija voidaan luoda koodilistauksen 6 tapaan lisäämällä events-moduulin sovellukseen ja sen jälkeen luomalla instanssin EventEmitter-oliosta.

```
1       var events = require("events");
2       var tapahtumanEmittoija = new events.EventEmitter();
```

Koodilistaus 6: Tapahtumanemittoijan luominen.

EventEmitter-olio voi emittoida uusia tapahtumia käyttämällä omaa emit-metodiaan seuraavasti: tapahtumanEmittoija.emit(”esimerkkitapahtuma”, arg1, arg2). Ensimmäisenä parametrinaan emit-metodi ottaa minkä tahansa merkkijonon, joka määrittelee tapahtuman tyypin. Esimerkissä arg1- ja arg2 -parametrit ovat vapaaehtoisia parametreja, joita voidaan antaa nolla tai useampia kappaleita. Ylimääräisillä parametreilla saadaan välitettyä tapahtumankäsittelijälle haluttua lisätietoa tapahtumasta. [6] Esimerkiksi nappia painettaessa tapahtumalle voidaan antaa parametreina tieto siitä, mitä nappia painettiin ja millainen painallus siihen suoritettiin.

EventEmitteristä on mahdollista myös periyttää oma toteutus, jolloin oman sovelluslogiikan lisääminen luokkaan on mahdollista. Näin voidaan esimerkiksi emittoida tapahtumia omalla oliolla. [6]

```
1       var EventEmitter = require(”events”).EventEmitter;
2       var util = require(”util”);
3
4       function OmaEventEmitter() {
5          EventEmitter.call(this);
6
7          this.emittoiTapahtuma = function() {
8             this.emit(”esimerkkitapahtuma”);
9          }
10      }
11      util.inherits(OmaEventEmitter, EventEmitter);
```

Koodilistaus 7: Esimerkkitapahtuman emittoiminen tapahtumanemittoijalla. Perustuu lähteeseen [6].

Koodilistauksessa 7 on peritty oma toteutus EventEmitteristä. Esimerkissä on käytetty util-moduulia, joka tarjoaa periyttämisfunktion inherits ja joukon muita funktioita. Inherits-funktiolla on periytetty OmaEventEmitter-oliolle EventEmitterin ominaisuudet, jotta oma tapahtumanemittoijaolio voi emittoida tapahtumia. Tässä

tapauksessa emittoiTapahtuma-funktio emittoi ”esimerkkitapahtuma”-tyyppisen tapahtuman. Emittointi onnistuu luomalla instanssi omasta tapahtuman emittoijan toteutuksesta, jonka jälkeen voidaan kutsua emittoiTapahtuma-metodia. Emittoitunut tapahtuma on tavallinen tapahtuma, johon voidaan kiinnittää omia tapahtumankuuntelijoita.

4.6 Tapahtumankuuntelija

Tapahtumien emittoinnista ei ole juurikaan hyötyä, jos tapahtumia ei kuunnella. Tapahtumankuuntelijan avulla tapahtumiin voidaan kiinnittää tapahtumankäsittelijäfunktioita, jotka aktivoituvat kun tietyntyyppinen tapahtuma emittoituu. Tapahtumankäsittelijäfunktio voi olla mikä tahansa first-class-funktio, joka vain liitetään haluttuun tapahtumaan. Tapahtumaan voidaan kiinnittää nolla tai useampia tapahtumankuuntelijoita. [6]

Node.js:ssä erityyppisiin tapahtumiin voidaan liittää tapahtumankuuntelijoita on- ja addListener-metodeilla [6]. Näiden kahden metodin toiminnallisuus on täysin sama ja on-metodi onkin vain oikotie addListener-metodiin lyhyemmän kirjoitusasunsa vuoksi [5]. Kumpikin metodeista ottavat argumentteinaan tapahtuman nimen ja tapahtumankäsittelyfunktion [6].

```
1       var emittoija = new EventEmitter();
2       emittoija.on(”esimerkkitapahtuma”, function() {
3          console log(”esimerkkitapahtuma emittoitiin”);
4       });
```

Koodilistaus 8: Tapahtumankuuntelijan liittäminen tapahtumaan. Perustuu lähteeseen [6].

Koodilistauksessa 8 liitetään tapahtumankuuntelija ”esimerkkitapahtuma”-tyyppiseen tapahtumaan. On-metodille toisena parametrina annettu funktio aktivoituu aina, kun jokin tapahtuman emittoija emittoi ”esimerkkitapahtuma”-tyyppisen tapahtuman. Tässä tapauksessa parametrina annettu callback-funktio tulostaa merkkijonon ”esimerkkitapahtuma emittoitiin” Node.js:n konsoliin aina, kun kiinnitetyn tyyppinen tapahtuma emittoituu.

Mikäli tarvitaan ainoastaan kerran kutsuttavaa tapahtumankäsittelijäfunktiota, voidaan käyttää tapahtumankuuntelijan lisäämiseen once-metodia vastaavalla tavalla kuin on- ja addListener-metodeja. Metodi once toimii siten, että se lisää ensin tapahtumalle tapahtumankuuntelijan, ja kun oikean tyyppinen tapahtuma emittoituu, se suorittaa tapahtumankäsittelijäfunktion ja tapahtumankäsittelijäfunktion suorituksen

jälkeen poistaa tapahtumalta tapahtumankuuntelijan. Näin voidaan varmistua siitä, ettei tapahtumankäsittelijäfunktiota enää kutsuta ensimmäisen kutsukerran jälkeen. [6]

Tapahtumankuuntelijoita saatetaan haluta poistaa itse käsin. Esimerkiksi tilanteessa, jossa halutaan palauttaa tapahtuman emittoija alkutilaan, jolloin sillä ei ole enää tapahtumankuuntelijoita tai kun halutaan poistaa jokin yksittäinen tapahtumankuuntelija, jolle ei ole enää käyttöä. Jos kaikki tapahtumankuuntelijat halutaan poistaa kerralla, voidaan kutsua EventEmitterin removeAllListeners-metodia. Kun metodia kutsutaan ilman argumenttia, se poistaa jokaisen tapahtumankuuntelijan riippumatta tapahtuman tyypistä. Mikäli metodille annetaan argumenttina tapahtuman nimi, poistetaan vain kyseisen argumentin tyyppiset tapahtumankuuntelijat. [6] RemoveAllListeners-metodia kannattaa kuitenkin käyttää vain, jos sille on todellista tarvetta ja tiedetään varmasti mitä ollaan tekemässä, koska sen avulla voidaan vahingossa poistaa tärkeitä tapahtumankuuntelijoita sovelluksen jostain toisesta osasta [5].

Pelkästään yhden tapahtumankuuntelijan poistamiseen on olemassa metodi removeListener. Sille annetaan argumentteina tapahtuman tyyppi ja siihen liittyvän poistettavan tapahtumankäsittelijäfunktion nimi. Tämä metodi suorittaa käänteisen operaation verrattuna on- ja addListener-metodeihin. Mikäli removeListener-metodia halutaan käyttää, tulee välttää anonyymejä tapahtumankäsittelijäfunktioita, koska niitä ei ole sidottu suorituskontekstiin. Koodilistauksessa 9 on esimerkki, missä tapahtumaan on lisätty anonyymi tapahtumankäsittelijäfunktio ja lisätty tapahtumankuuntelija yritetään poistaa antamalla removeListener-metodille parametrina identtinen anonyymi funktio. Tämä ratkaisu ei kuitenkaan poista tapahtumankäsittelijää, koska kaksi erillistä funktio-oliota eivät osoita samaan paikkaan muistissa ja siksi niitä ei käsitellä ekvivalentteina. [6]

```
1       emittoija.on(”esimerkkitapahtuma”, function() {
2          console log(”käsittelijäfunktiota kutsuttiin”);
3       });
4
5       emittoija.removeListener(”esimerkkitapahtuma”, function() {
6          console log(”käsittelijäfunktiota kutsuttiin”);
7       });
```

Koodilistaus 9: Väärä tapa poistaa lisätty tapahtumankuuntelijafunktio tapahtumalta. Perustuu lähteeseen [6].

```
1       function kasittelija() {
2          console log(”käsittelijäfunktiota kutsuttiin”);
3       }
4
5       emittoija.on(”esimerkkitapahtuma”, kasittelija);
6       emittoija.removeListener(”esimerkkitapahtuma, kasittelija);
```

Koodilistaus 10: Toimiva tapa poistaa lisätty tapahtumankuuntelijafunktio tapahtumalta. Perustuu lähteeseen [6].

Koodilistauksessa 10 esitetään toimiva ja muutenkin selkeämpi tapa poistaa tapahtumankuuntelija käyttäen muuttujaan talletettua first-class-funktiota. Tässä tilanteessa funktiot ovat keskenään ekvivalentteja ja tapahtumankuuntelija poistetaan onnistuneesti. [6]

Joissain tilanteissa voidaan haluta tietää, mitä tapahtumankuuntelijoita tapahtumiin on liitetty. Jos halutaan ainoastaan tieto siitä, kuinka monta tapahtumankuuntelijaa tapahtumalla on, voidaan käyttää EventEmitter-olion metodia listenerCount. Huomioitavaa listenerCount-metodia käytettäessä on se, ettei sitä kutsuta EventEmitterin instanssille vaan suoraan EventEmitter-oliolle seuraavasti: EventEmitter.listenerCount(emittoija, ”tyyppi”). Esimerkissä annetaan parametreina tapahtuman emittoijan instanssi ja tapahtuman tyyppi merkkijonona. Metodi palauttaa parametreina annettuun tapahtumanemittoijaan liittyvien tapahtumankäsittelijäfunktioiden lukumäärän. Jos halutaan tarkastella tapahtumankäsittelijäfunktioita, on siihen tarkoitukseen olemassa EventEmitter-olion listeners-metodi, joka palauttaa taulukon tapahtumankäsittelijäfunktioista. Huomioitavaa on, että tämän taulukon muokkaaminen ei kuitenkaan vaikuta tapahtumankäsittelijöihin. [6]

Tavallisesti tapahtumanemittoijaan liitetään vain muutamia tapahtumankuuntelijoita, mutta esimerkiksi ohjelmointivirhe saattaa aiheuttaa sen, että jollekin tapahtumalle lisätään paljon ylimääräisiä tapahtumankuuntelijoita, mistä seuraa muistivuotoja. Ongelmia syntyy esimerkiksi kun silmukassa yritetään asettaa tapahtumalle useita tapahtumankuuntelijoita, mutta ohjelmointivirheen takia silmukka käydäänkin liian monta kertaa läpi ja muistia kuluu turhaan ylimääräisiin tapahtumankuuntelijoihin. Node.js kuitenkin varoittaa konsolissaan, mikäli yli kymmenen tapahtumankuuntelijaa asetetaan mille tahansa yksittäiselle tapahtumalle. Tätä kynnysarvoa voidaan muuttaa setMaxListeners-metodilla, mikäli jonkin tapahtuman käsittely vaatii yli kymmenen tapahtumankuuntelijaa. SetMaxListeners-metodille annetaan argumenttina kokonaisluku, joka määrittää tapahtumankäsittelijöiden maksimimäärän. Poikkeuksena tähän on kokonaisluku nolla, joka sallii tapahtumalle rajoittamattoman määrän tapahtumankuuntelijoita. Tämän metodin käyttö ei vaikuta ohjelman toimintaan

mitenkään ja siksi se onkin hyvä sovelluksen testaamiseen. Esimerkiksi asettamalla setMaxListeners-metodi haluttuun raja-arvoon, voidaan havaita, onko tapahtumankäsittelijöitä liitetty tiettyyn tapahtumaan oikea määrä. [6]

Aina kun jokin uusi tapahtumankäsittelijä rekisteröidään mille tahansa tapahtumalle, tapahtumanemittoija emittoi ”newListener”-tyyppisen tapahtuman. Tähän tapahtumaan voidaan asettaa tapahtumankuuntelija, kuten muihinkin tapahtumiin, antamalla on- tai addListener-metodille tapahtuman tyypiksi ”newListener”-merkkijono ja toiseksi argumentiksi tapahtumankäsittelijäfunktio. Käyttötarkoituksia tällaiselle tapahtumalle ovat esimerkiksi resurssien varaus tai jos jokaisella uuden tapahtuman rekisteröintikerralla täytyy suorittaa jokin tietty osa sovelluskoodia. Tärkeää on pitää mielessä, että ”newListener”-tyyppinen tapahtuma on jo olemassa, kun omia tapahtumia luodaan. Tapahtuman tapahtumankäsittelijäfunktio odottaa parametreikseen tapahtuman tyyppiä merkkijonona ja tapahtumankuuntelijafunktiota. Jos tästä poiketaan antamalla ”newListener”-tapahtuman tapahtumankäsittelijäfunktioille väärän tyyppinen parametri tapahtumankuuntelijan tilalle, saatetaan aiheuttaa virhetilanteita tulkitsemalla esimerkiksi jokin olio tapahtumankuuntelijaksi. [6]

```
1       var events = require(”events”);
2       var emittoija = new events.EventEmitter();
3
4       emittoija.on(”newListener”, function(tapahtumanNimi, kuuntelija){
5          console.log(”uusi kuuntelija lisättiin”);
6       });
7       emittoija.on(”esimerkkitapahtuma”, function() {});
```

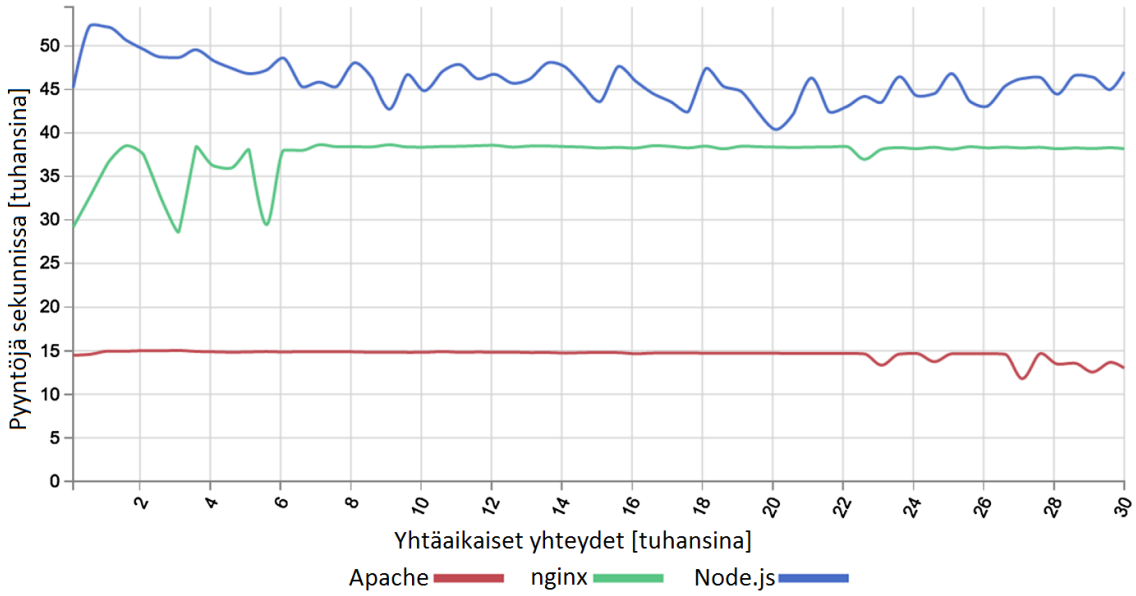
Koodilistaus 11: Tapahtumankuuntelijan lisääminen newListener-tapahtumalle. Perustuu lähteeseen [6].

Koodilistauksessa 11 ”newListener”-tapahtumaan liitetään oikealla tavalla tapahtumankuuntelijafunktio, joka tulostaa ”uusi kuuntelija lisättiin” -merkkijonon konsoliin jokaisella kerralla, kun uusi kuuntelija liitetään johonkin tapahtumaan. Esimerkin tapauksessa merkkijono tulostetaan konsoliin vain kerran, koska viimeisellä rivillä ”esimerkkitapahtuma”-tyyppiseen tapahtumaan liitetään yksi tapahtumankäsittelijäfunktio. [6]

5 NODE.JS:N EDUT

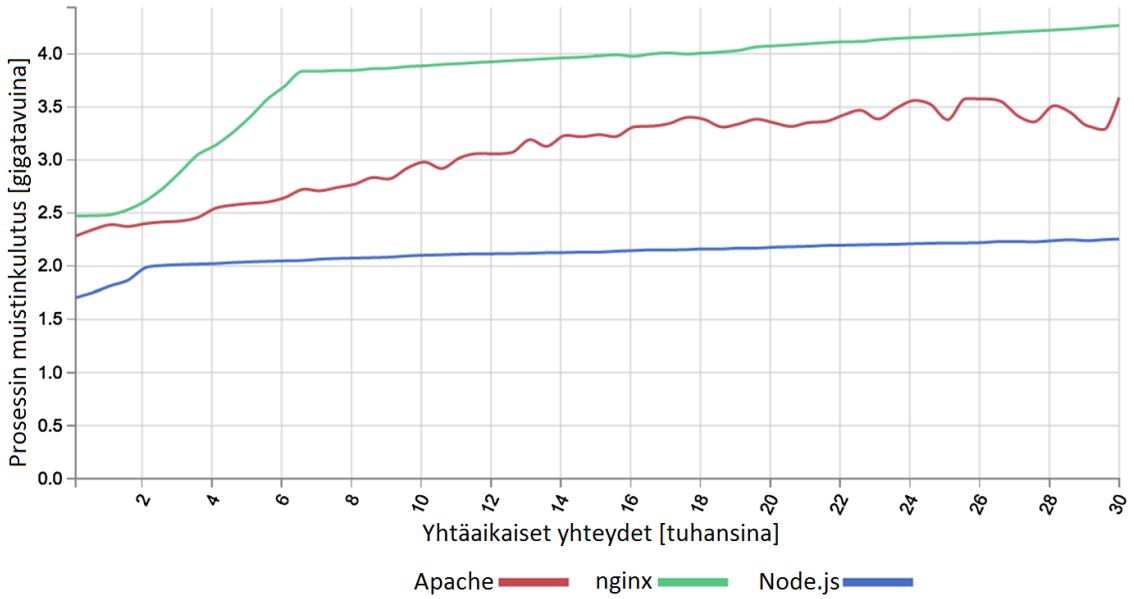
5.1 Suorituskyky ja skaalautuvuus

Node.js:n suurimmat vahvuudet ovat sen suorituskyky suuren kuorman käsittelyssä sekä skaalautuvuus. Koska tapahtumapohjainen ja asynkroninen ohjelmointi on Node.js:ssä hyvin sisäänrakennettua, se pystyy hallitsemaan useita yhtäaikaisia pyyntöjä, vaikka käyttääkin vain yhtä säiettä. [3] Esimerkiksi PHP:n Apache on tehoton, kun liikennettä on paljon, ja vaikka Nginx-palvelin on I/O-operaatioissa 2,5 kertaa PHP:n Apachea nopeampi, suoriutuu Node.js testistä silti molempia paremmin [2].



Kuva 3: Suorituskykytesti, jossa kukin palvelin pyrkii vastaamaan ”hello world” -merkkijonolla yhtäaikaisten käyttäjien kasvaessa. Perustuu lähteeseen [2].

Kuvassa 3 näkyy pyyntöjen määrä yhtäaikaisiin yhteyksiin verrattuna, kun jokainen palvelin tässä suorituskykytestissä vastaa pyyntöihin ”hello world” -merkkijonolla [2]. Kuvasta 3 nähdään, että Node.js pystyy hallitsemaan useampia yhtäaikaisia yhteyksiä kuin Apache tai nginx.



Kuva 4: Eri palvelinten muistinkäyttö yhtäaikaisten yhteyksien määrän kasvaessa. Perustuu lähteeseen [2].

Kuvassa 4 on esitetty eri palvelinten muistinkäyttö, kun yhtäaikaisia yhteyksiä lisätään. Kuvasta 4 näkyy selkeästi, kuinka Node.js:n muistinkulutus pysyy suurilla käyttäjämäärillä noin 2 Gt:n suuruisena. Vastaavasti nginxin muistinkulutus nousee noin 4 Gt:uun, joten myös muistinkäytössä Node.js on Apachea ja nginxiä parempi.

Artikkelin ”Is Node.js a viable option for building modern web applications” [2] I/O-testissä Apache epäonnistuu pyyntöihin vastaamisessa jo noin 19 000 yhtäaikaisen yhteyden aikana, kun nginx ja Node.js vastaavat onnistuneesti 30 000 yhtäaikaiseen käyttäjään saakka, mikä on myös tässä testissä suurin testattu määrä yhtäaikaisia käyttäjiä. Node.js on Apachea ja nginxiä tehokkaampi entistä selkeämmin artikkelin ”Is Node.js a viable option for building modern web applications” [2] toisessa laskennallisessa testissä, jossa palvelimet koodaavat maantieteelliset koordinaatit geohashing-algoritmilla jokaisen pyynnön yhteydessä. [2]

Node.js:n yksisäikeisyys voi olla rajoite, mutta näihin tilanteisiin on olemassa cluster core -moduuli, jonka avulla yksi sovellus voi käynnistää useampia prosesseja ja näin jakaa resurssit sekä kuorman useammalle ytimelle. Näin moniytimisestä suoritusympäristöstä voidaan käyttää kaikki suorituskyky ja sovellukset saadaan skaalautumaan tehokkaasti. Nykyisin sovellusten skaalautumisessa tärkeänä apuna ovat myös pilvitoteutukset, ja Node.js:lle onkin olemassa hyviä pilvilaskenta-alustoja, kuten Heroku ja Nodejitsu, joiden avulla Node.js-sovellukset voidaan sijoittaa ajoon pilviympäristöön. [6]

5.2 Edut JavaScriptistä

Node.js:llä ohjelmoidaan päästä-päähän JavaScriptillä, eli sekä palvelimella että asiakaspäässä ohjelmoitaessa käytetään JavaScriptiä ohjelmointikielenä. JavaScriptin käyttäminen kaikkialla helpottaa rikkaiden web-sovellusten kirjoittamista [3], sillä lähes kaikki selaimet tukevat JavaScriptiä ja se on tehokas työkalu asiakaspäässä esimerkiksi käyttöliittymien tekemisessä [1]. Koska JavaScriptillä ohjelmoidaan kaikki Node.js:llä, kehittäjien ei tarvitse välttämättä osata useita kieliä, jolloin ohjelmointi on yksinkertaisempaa. Myös ohjelmoija, joka osaa ohjelmoida palvelinpäätä hyvin, voi toimia samalla myös asiakaspään kehittäjänä, minkä seurauksena työskentelystä tulee tehokkaampaa. [3]

JavaScript ratkaisee myös asynkronisesta suoritustavasta syntyviä ongelmia, kuten tapahtumien suorituskontekstin säilyttämisen. Koska JavaScriptissä on käytössä anonyymit funktiot ja klosuurit, on mahdollista luoda callbackeja, jotka säilyttävät suorituskontekstinsa. Callbackien avulla yleensä päästään eroon huonosti ylläpidettävästä ohjelmakoodista, jossa esimerkiksi julkisilla muuttujilla tai muilla tavoin varmistetaan se, että funktioilla on pääsy suorituskontekstiinsa. [1]

Suorituskontekstin säilyttäminen onnistuu klosuurilla, joka muodostuu anonyymin funktion ympärille. Klosuuri muistaa ympäristönsä tilan ja kapseloi oman nimiavaruutensa, jolloin julkiseen nimiavaruuteen ei tarvitse luoda muuttujia ja hallita niiden mahdollisia törmäyksiä. Esimerkiksi, kun jokin yksittäinen callback-funktio käsittelee useita yhtäaikaisia pyyntöjä, joilla jokaisella on oma sisääntulodata, klosuuri huolehtii siitä, ettei törmäyksiä tapahdu. First-class-funktiot ja klosuurit JavaScriptissä mahdollistavat tapahtumapohjaisen ohjelmoinnin ja lukkiutumattoman asynkronisen I/O:n, mikä tekee Node.js:stä hyvin tehokkaan ja samalla vähentää sovelluksien monimutkaisuutta huomattavasti. [3]

Node.js:n suorituskykyä palvelimena parantaa Googlen V8 JavaScript -moottori, joka on optimoitu suorittamaan JavaScript-ohjelmakoodia hyvin nopeasti. Se on huomattavasti nopeampi kuin muut skriptikielet, esimerkiksi Ruby tai Python. Googlen V8 JavaScript -moottorin mediaanisuorituskyky Python 3:a vastaan on 13 kertaa parempi, ja Ruby 1.9:ää vastaan V8:n mediaanisuorituskyky on kahdeksan kertaa parempi, joten JavaScript-ohjelmakoodin suorituskyky ei ole enää nykyään ongelma. [9] Googlen V8 JavaScript -moottori pysyy suosionsa vuoksi myös päivitettynä ja näin tukee ECMAScriptin viimeisimpiä ominaisuuksia [7].

5.3 Muut edut

Node.js on saavuttanut lyhyessä ajassa runsaasti suosiota ja siksi sen ympärille onkin ehtinyt kehittyä merkittävä ekosysteemi [4]. Kolmannen osapuolen kirjastoja on tarjolla hyvin paljon ja niiden käyttämiseen Node.js:n mukana tulee npm-

paketinhallintatyökalu, jonka avulla kirjastojen käyttöönotto on hyvin yksinkertaista [10]. Myös useat ohjelmistokehykset, kuten Connect ja Express, tarjoavat Node.js:lle täyden tuen sekä palvelin- että asiakaspään ohjelmointiin [1].

Node.js:n ydintoiminnallisuus on pyritty pitämään minimissä ja lisätoiminnallisuus tarjotaan sovellusrajapintoina. Myös Node.js:n asentaminen on tehty yksinkertaiseksi, eikä erillisille konfiguraatiotiedostoille ole tarvetta. Myöskään sovelluskehittäminen ei vaadi muuta kuin sovellusrajapintojen käytön opettelun, mikäli JavaScript ja asynkroninen ohjelmointi ovat jo ennestään tuttuja. [3]

6 TAPAHTUMAPOHJAISEN OHJELMOINNIN HAASTEET NODE.JS:SSÄ

Node.js:ää ei ole tarkoitettu vastaamaan pyyntöihin staattisella tiedostolla ja siksi se ei pärjääkään Apachelle tai nginxille staattisten tiedostojen tarjoamisessa. Artikkelissa ”Is Node.js viable option for building modern web applications” [2] tehdyssä testissä, jossa palvelimen täytyy vastata asiakkaan pyyntöihin 13,5 kilotavun kokoisella tiedostolla, Node.js käyttää huomattavasti enemmän muistia kuin nginx tai Apache. [2] Node.js:n heikkoutena on myös suorittimella tehtävä kova laskenta eikä se toimikaan kovin hyvin esimerkiksi alkulukuja laskettaessa [9].

Node.js on keskeneräinen ja kokoajan kehittyvä tuote, mikä täytyy ottaa huomioon sovelluskehityksessä. Node.js:n dokumentaatiossa esitellään sen sovellusrajapinnan eri osille stabiliteettitasoja, jotka kertovat kuinka todennäköistä rajapinnan muuttuminen tulevaisuudessa on. Taso nolla tarkoittaa sitä, että rajapinta tiedetään ongelmalliseksi ja muutokset ovat suunniteltuja, kun taas tason viisi rajapinnat ovat lukittuja ja muuttuvat vain, jos vakavia virheitä löydetään. [4]

Ongelmia aiheuttaa myös Node.js:n käyttämä continuation-passing-ohjelmointityyli. Se voi johtaa callback-pyramidiksi kutsuttuun tilanteeseen, joka on yleinen ongelma continuation-passing-syntaksia käytettäessä. Callback-pyramidi syntyy, kun callbackit kasautuvat toisten callbackien sisään useamman tason syvyyteen. Tästä seuraa myös pyramidinimitys koodirivien muistuttaessa sivuttain olevia pyramideja. [6] Koodilistauksen 12 callback-pyramidiesimerkissä tehdään tarkastelut tiedoston olemassaololle ja varmistetaan tiedoston olevan tiedosto, eikä esimerkiksi kansio. Tarkastelujen jälkeen tiedosto luetaan ja tulostetaan sen sisältö konsoliin.

```
1       var fs = require("fs");
2
3       fs.exists("tiedosto", function(exists) {
4          if(exists) {
5             fs.stat("tiedosto", function(err, stats) {
6                if(err) {
7                   throw err;
8                }
9                if(stats.isFile()) {
10                  fs.readFile("tiedosto", "utf8", function(err, data) {
11                     if(err) {
12                        throw err;
13                     }
14
15                     console.log(data);
16                  });
17               }
18            });
19         }
20      });
```

Koodilistaus 12: Callback-pyramidit alkavat kasaantua ja ohjelmakoodin luettavuus heikkenee huomattavasti. Perustuu lähteeseen [6].

Koodilistauksesta 12 näkee selvästi, että koodin sisennystaso kasvaa huomattavasti jo näin yksinkertaisessa esimerkissä. Callback-pyramidit ovat siis todellinen ongelma monimutkaisimmissa Node.js-sovelluksissa, koska jo esimerkin kaltaisen ohjelmakoodin lukeminen ja ymmärtäminen on hyvin hankalaa. Tällä tavalla ohjelmoimalla todellisissa suuremmissa web-sovelluksissa ohjelmakoodin seuraaminen saattaa käydä mahdottomaksi tai ainakin erittäin työlääksi. Useita sisäkkäisiä callbackeja sisältävä ohjelmakoodi on myös hankalasti ylläpidettävää. [6]

Callback-pyramideista voidaan kuitenkin osittain päästä eroon luomalla callbackeista first-class-funktioita anonyymien funktioiden sijaan [6]. Tämä on hyvä tapa muutenkin, mikäli funktioita käytetään Node.js:ssä tapahtumankäsittelijöinä, koska tällöin niiden kuuntelijoiden poistamisessa ei synny ongelmia [5]. Koodista tulee kuitenkin hiukan pidempää ja mahdollisesti hieman vaikeammin seurattavaa, mutta ohjelmakoodin sisennystaso pysyy järkevänä ja hallittavissa [6].

Toinen tapa vältellä callback-pyramideja ovat tapahtuman emittoijat. Perimällä oma toteutus EventEmitter-oliosta ja kapseloimalla klosuurin sisään toiminnallisuus, voidaan tapahtumia emittoimalla toteuttaa koodilistauksen 10 esimerkki koodilistauksen 13 tapaan, jolloin vältetään callback-pyramidit tapahtumien avulla. [6]

```
1       var EventEmitter = require("events").EventEmitter;
2       var util = require("util");
3       var fs = require("fs");
4
5       function TiedostonLukija(tiedosto) {
6          var _self = this;
7
8          EventEmitter.call(_self);
9
10         _self.on("stats", function() {
11            fs.stat(tiedosto, function(err, stats) {
12               if(err) {
13                  throw err;
14               }
15
16               if(stats.isFile()) {
17                  _self.emit("read");
18               }
19            });
20         }
21
22         _self.on("read", function() {
23            fs.readFile(tiedosto, "utf8", function(err, data) {
24               if(err) {
25                  throw err;
26               }
27            console.log(data);
28            });
29         }
30
31         fs.exists(tiedosto, function(exists) {
32            if(exists)  {
33               _self.emit("stats");
34            }
35         });
36      };
37
38      util.inherits(TiedostonLukija, EventEmitter);
39      var lukija = new TiedostonLukija("esimerkkitiedosto");
```

Koodilistaus 13: Callback-pyramidien välttäminen käyttäen omaa tapahtumaemittoijaa. Perustuu lähteeseen [6].

Koodilistauksessa 13 peritään EventEmitterin ominaisuudet util-moduulin avulla TiedostonLukija-luokalle. TiedostonLukija-luokalle annetaan parametrina tiedosto, jonka olemassaolo tarkastetaan ensin rivillä 31. Sen jälkeen emittoidaan ”stats”-tapahtuma ja suoritus siirtyy riville 10, joka onnistuessaan emittoi ”read”-tapahtuman, jolloin siirrytään riville 22. Lopulta tiedoston sisältö saadaan tulostettua rivillä 27, mikäli tiedosto on olemassa oleva tiedosto ja sen lukeminen onnistuu. Yksityisen muuttujan _self avulla pystytään viittaamaan kyseiseen instanssiin asynkronisten callbackien sisällä, koska callbackien sisällä this-muuttuja ei viittaa enää samaan instanssiin. Tämän takia emit()-metodia ei kutsuta this-muuttujan kautta callbackeissa

vaan sitä kutsutaan _self-muuttujan kautta. Tällä tavalla tapahtumia emittoimalla voidaan päästä callback-pyramideista eroon ja pitää ohjelmakoodi hallittavana. [6]

7 YHTEENVETO

Node.js tarjoaa vaihtoehtoisen yksisäikeisen tapahtumapohjaisen sovellusalustan monisäikeisten järjestelmien rinnalle ja monisäikeisistä toteutuksista se eroaa pääasiassa kahdella tavalla. Node.js:n sisäänrakennettu tapahtumapohjaisuus pakottaa sovelluskehittäjät alusta asti ohjelmoimaan tapahtumapohjaisesti ja se käyttää ohjelmointikielenään JavaScriptiä sekä palvelin- että asiakaspäässä. Node.js:n avulla kehittäjät voivat helpommin luoda nopeita ja skaalautuvia web-sovelluksia.

Node.js on pyritty tekemään mahdollisimman tehokkaaksi ja vähän muistia kuluttavaksi suurenkin kuormituksen alla. Tehokkuus saavutetaan sillä, että Node.js käyttää Googlen V8 JavaScript -moottoria, joka on optimoitu suorittamaan JavaScript-sovelluskoodia hyvin nopeasti. Node.js on suurimmaksi osaksi toteutettu C:llä, C++:lla ja JavaScriptillä, joiden avulla Node.js:n muistinkäyttö on saatu alhaiseksi. Node.js on myös lukkiutumaton, mikä vähentää sovelluskehittäjien tarvetta huolehtia synkronisoinnista.

Node.js:n huonoja puolia ovat suorituskyky staattisten tiedostojen tarjoamisessa ja raskas laskenta. Raskaasta laskennasta Node.js suoriutuu huonosti yksisäikeisen tapahtumasilmukan takia, joka lukkiutuu tällaisen laskennan seurauksena. Node.js:ssä tapahtumapohjaiseen ohjelmointiin käytetty continuation-passing-syntaksi aiheuttaa callback-pyramideina tunnetun ongelman. Callback-pyramideilla tarkoitetaan callback-funktioiden kasautumista toistensa sisään, jolloin sovelluskoodista tulee vaikeasti ylläpidettävää ja ymmärrettävää.

Kaiken kaikkiaan Node.js on hyvin tehokas ja vähän muistia käyttävä sovellusalusta websovelluksille, joiden täytyy kestää paljon liikennettä. Koska Node.js käyttää JavaScriptiä ohjelmointikielenä päästä päähän, se on hyvä työkalu kehittäjille, jotka osaavat JavaScriptiä hyvin. Joihinkin tilanteisiin, kuten staattisten tiedostojen tarjoamiseen tai raskaaseen laskentaan, Node.js ei kuitenkaan ole aina hyvä valinta.

LÄHTEET

[1] Stefan Tilkov, Steve Vinoski, Node.js: Using JavaScript to Build High Performance Network Programs, IEEE Computing Society, pp. 80-83.

[2] Ioannis K. Chaniotis, Kyriakos-Ioannis D. Kyriakou, Nikolaos D. Tselikas, Is Node.js a Viable option for building modern web applications? A performance evaluation study, Springer, 2014.

[3] Zammetti, Frank, Pro iOS and Android Apps for Business with jQuery Mobile, Node.js, and MongoDB, Apress, 2013, 286 p.

[4] Nodejs, verkkosivu, saatavissa (viitattu 02.10.2014): http://node.js.org.

[5] Teixera, Pedro, Professional Node.js, John Wiley & Sons, Inc., 2013, 371 p.

[6] Ihrig, Colin, Pro Node.js for Developers, Apress, 2013, 277 p.

[7] Rauch, Guillermo, Smashing Node.js JavaScript Everywhere, John Wiley & Sons, Inc., 2012, 308 p.

[8] Kunkle.org, verkkosivu, saatavissa (viitattu 09.10.2014): http://kunkle.org/nodejs-explained-pres/#/event-loop.

[9] Nguyen, Don, Jump Start Node.js, Sitepoint Pty, 2012, 154 p.

[10] npm, verkkosivu, saatavissa (viitattu 14.10.2014): https://www.npmjs.org.