Vilma Lappi

Haskellin käyttö web-palvelinohjelmoinnissa

Tietotekniikan kandidaatintutkielma

13. toukokuuta 2018

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Vilma Lappi

Yhteystiedot: vilma.j.lappi@student.jyu.fi

Ohjaaja: Antti-Juhani Kaijanaho

Työn nimi: Haskellin käyttö web-palvelinohjelmoinnissa

Title in English: The Use of Haskell in Web Server Programming

Työ: Kandidaatintutkielma

Sivumäärä: 22+0

Tiivistelmä: Funktio-ohjelmointia ei tavallisesti juurikaan käytetä web-kehityksessä. Tässä tutkielmassa selvitetään, olisiko Haskell-nimisestä funktio-ohjelmointikielestä kuitenkin hyötyä web-palvelinohjelmoinnissa. Tutkimuskysymyksenä on "Miten Haskellia voi hyödyntää web-palvelinohjelmoinnissa"ja tutkimusmenetelmänä on kirjallisuuskatsaus. Tuloksena saadaan, että Haskellilla ohjelmoitujen web-palvelinten etuja ovat erityisesti nopeus ja pieni koko.

Avainsanat: Haskell, web-palvelin, funktio-ohjelmointi

Abstract: Functional programming isn't normally used much in web development. In this thesis we investigate whether Haskell, a functional programming language, could nonetheless be useful in web server programming. The research problem is "How can Haskell be utilized in web server programming?" and the method used is literacy review. As a result we find out that advantages of web servers programmed in Haskell are speed and small size.

Keywords: Haskell, web server, functional programming

Sisältö

1 Johdanto

Web-palvelimet ovat nykypäivänä osa jokaisen suomalaisen elämää. Internet on tärkeä tiedonhaku- ja viestintäväline, puhumattakaan viihdekäytöstä. Lisäksi yhä useammat palvelut siirtyvät internetiin ja niihin pääsyn sekä niiden toiminnan mahdollistavat web-palvelimet.

Suurin osa tällä hetkellä käytössä olevista web-palvelimista on ohjelmoitu C- tai C++-ohjelmointikielellä, joista molemmat ovat imperatiivisen ohjelmointiparadigman mukaisia. Deklaratiivisilla kielillä, kuten funktio-ohjelmointikielillä, on kuitenkin omat etunsa. Funktio-ohjelmat ovat esimerkiksi yleensä lyhyempiä ja tiiviimpiä kuin vastaavat imperatiivisella kielellä ohjelmoidut ohjelmat, ja funktio-ohjelmointikielille tyypillinen vahva ja staattinen tyypitys estää ohjelman kaatumisen tyyppivirheisiin ajon aikana. Tässä tutkielmassa selvitetään, olisiko funktio-ohjelmoinnin erityispiirteistä hyötyä web-palvelinten ohjelmoinnissa.

Tutkielman tarkoituksena on selvittää, miten Haskell soveltuu web-palvelinten toteutukseen. Tutkimuskysymys on "Miten Haskellia voi hyödyntää web-palvelinohjelmoinnissa". Haskell on yksi suosituimmista puhtaista funktio-ohjelmointikielistä. Sitä käytetään paljon opetuksessa, ja teollisuudessa sitä käyttää esimerkiksi Facebook, joka käyttää sitä roskapostin ja muun haitallisen materiaalin tunnistamiseen ja poistamiseen.

Tutkielman luvussa 2 määritellään funktio-ohjelmointi ja esitellään sen ominaisuuksia sekä esitellään Haskellin omat erityispiirteet. Luvussa 3 kerrotaan web-palvelinten toiminnasta, ominaisuuksista ja ohjelmoinnista. Luvussa 4 esitellään Haskellin ominaisuuksien sopivuutta web-palvelinten ohjelmointiin ja verrataan Haskellia muihin ohjelmointikieliin, joita web-palvelinten ohjelmoinnissa käytetään. Luvussa 5 kootaan yhteen tärkeimpiä johtopäätöksiä.

2 Funktio-ohjelmointi

Funktio-ohjelmointi on deklaratiivinen ohjelmointiparadigma, jossa laskenta tapahtuu matemaattisten funktioiden avulla. Tässä luvussa esitellään funktio-ohjelmointia yleisesti ja Haskellin omia ominaisuuksia. Aluksi vertaillaan imperatiivista ja deklaratiivista ohjelmointiparadigmaa luvussa 2.1, sitten esitellään funktio-ohjelmointikielten tyypillisiä piirteitä luvussa 2.2 ja lopuksi esitellään Haskell sekä sen erityispiirteitä luvussa 2.3.

2.1 Imperatiivinen ja deklaratiivinen ohjelmointi

Hudakin (1989) mukaan imperatiivinen ohjelmointi on ohjelmointiparadigma, jolla toteutetulla ohjelmalla on aina tila, jota muokataan komennoilla. Yksi esimerkki ohjelman tilan muokkaamisesta on sijoituslause, jossa muuttujaan sijoitetaan uusi arvo. Komennot suoritetaan järjestyksessä alusta loppuun. Niiden suoritusjärjestykseen voi vaikuttaa ohjausrakenteiden, kuten silmukoiden ja ehtolauseiden, avulla. Tietyn ongelman ratkaisu esitetään algoritmina, joka sisältää ohjeet askel askeleelta ongelman ratkaisuun.

Hudakin (1989) mukaan deklaratiivinen ohjelmointi sen sijaan on ohjelmointiparadigma, jossa ei ole muuttuvaa tilaa. Sitä voidaan kuvata imperatiivisen paradigman vastakohtana. Sen sijaan, että esitetään ohjeet tiettyyn tilaan pääsemiseen, ongelma ratkaistaan lausekkeiden avulla. Lloydin (1994) mukaan deklaratiivisessa ohjelmoinnissa ei esitetä, miten ongelma ratkaistaan, vaan esitetään ratkaisu suoraan. Hänen mukaansa deklaratiiviselle paradigmalle on olennaista myös, että sen mukaiset ohjelmointikielet noudattavat johonkin tiettyyn teoriaan pohjautuvaa logiikkaa. Funktio-ohjelmoinnin noudattama teoria olisi tämän määritelmän mukaan lambda-laskenta. Koska deklaratiivisessa ohjelmoinnissa lausekkeita ei suoriteta tietyssä etenevässä järjestyksessä, siinä ei käytetä silmukoita vaan rekursiota. Funktio-ohjelmointikielet pohjautuvat deklaratiiviseen paradigmaan, kun taas useimmat olio-ohjelmointikielet perustuvat imperatiiviseen paradigmaan.

Yksi imperatiivista ja deklaratiivista paradigmaa erottava tekijä on myös sivuvaikutuksellisuus. Tietotekniikassa sivuvaikutuksellisuus tarkoittaa sitä, että aliohjelma tai komento muokkaa jotakin tilaa oman näkyvyysalueensa ulkopuolella. Tämä on tyypillistä imperatiivisessa

ohjelmoinnissa, jossa saatetaan esimerkiksi muokata globaalia muuttujaa tai kirjoittaa tiedosto. Deklaratiivisessa ohjelmoinnissa ja siten funktio-ohjelmoinnissa sivuvaikutuksellisuus pyritään minimoimaan tai jopa eliminoimaan. Hudakin (1989) mukaan sivuvaikutuksettomuuden tuloksena on viitteellinen läpinäkyvyys (referential transparency).

Hudak (1989) määrittelee viitteellisen läpinäkyvyyden niin, että jos esimerkiksi muuttujaan x on sijoitettu luku 5, niin muuttuja x voidaan korvata luvulla 5 kaikissa yhteyksissä missä sitä käytetään. Samaa ei voisi tehdä imperatiivisessa ohjelmoinnissa suoraan, vaan ensin pitäisi tutkia, onko muuttujaan x sijoitettu jonkin muu arvo edellisen sijoituksen jälkeen. Field ja Harrison (1988) sen sijaan määrittelevät viitteellisen läpinäkyvyyden niin, että tietyn lausekkeen arvo on aina sama eivätkä ohjelman muut osat voi sitä muuttaa. Pohjimmiltaan molemmat määritelmät tarkoittavat samaa asiaa ja johtavat siihen, että sivuvaikutuksettomassa ohjelmoinnissa muuttujat ovat siis muuttumattomia.

2.2 Funktio-ohjelmointikielten pääpiirteet

Funktio-ohjelmoinnin juuret ovat lambda-laskennassa, Alonzo Churchin 1920-1930 -luvuilla kehittämässä formaalin laskennan mallissa. Fieldin ja Harrisonin (1988, s. 111) mukaan funktio-ohjelmointikielet ovatkin osittain vain helpompikäyttöistä lambda-laskentaa ja funktio-ohjelmat ovat käännettävissä vastaaviksi lambda-lausekkeiksi. Lambda-laskentaa ei tässä tutkielmassa käsitellä tarkemmin, sillä sen tarkka tunteminen ei ole oleellista aiheen kannalta. Tärkeimmät siihen pohjautuvat funktio-ohjelmoinnin ominaisuudet esitellään tässä luvussa.

Hudakin (1989) mukaan funktio-ohjelmointikielille tyypillisiä piirteitä ovat higher-order-funktiot, laiskuus, vahva ja staattinen tyypitys sekä pattern matching. Hänen mukaansa higher-order-funktio on funktio, jota kohdellaan samalla tavalla kuin muitakin data-objekteja, kuten esimerkiksi muuttujia ja tietorakenteita. Niitä voi siis esimerkiksi antaa parametrina tai palauttaa toisesta funktiosta. Higher-order-funktioihin liittyvät myös anonyymit funktiot, jotka ovat Hudakin ym. (2007) mukaan peräisin lambda-laskennasta. Anonyymit funktiot ovat nimensä mukaisesti nimeämättömiä funktioita, joita käytetään tyypillisesti esimerkiksi higher-order-funktioina.

Laiskuudella tarkoitetaan sitä, että tietty osa funktiota tai koodia suoritetaan vain tarvittaessa. Hudakin (1989) mukaan se mahdollistaa loputtomien listojen käsittelyn. Lisäksi Fischerin, Kiselyovin ja Shanin (2009) mukaan laiskuus varmistaa, että jokaisen muuttujan arvo lasketaan vain kerran, vaikka sitä käytettäisiin useassa eri ohjelman kohdassa. Laiskuuden tehokkuudesta esitetään kirjallisuudessa eriäviä mielipiteitä. Esimerkiksi Hudakin ym. (2007) mukaan arvojen laskemisen viivyttäminen, kunnes niitä tarvitaan, ja niiden tallentaminen, jottei niitä tarvitse laskea uudestaan, verottaa ohjelmointikielen tehokkuutta. Toisaalta esimerkiksi Fischerin, Kiselyovin ja Shanin (2009) mukaan Bird, Jones ja De Moor (1997) esittävät, että muuttujien arvojen laskeminen vain kerran parantaa tehokkuutta ainakin joissain tilanteissa. Laiskuudesta saatava hyöty on siis tilannekohtaista.

Tyypitysten määritelmistä on kirjallisuudessa erilaisia kuvauksia. Termiä staattinen tyypitys käytetään usein kuvaamaan sitä, että tyypit tarkistetaan ennen ohjelman suoritusta, eikä ajon aikaisesti. Tätä määritelmää tukevat esimerkiksi Hudak (1989) ja Cardelli ja Wegner (1985). Tässä määritelmässä staattisen tyypityksen vastakohtana pidetään dynaamista tyypitystä, jossa tyypit tarkastetaan ohjelman ajon aikana. Toisaalta esimerkiksi Cardelli (1996) esittää, että kaikki tyypitetyt kielet olisivat staattisesti tyypitettyjä. Tämän näkemyksen mukaan kielet, joissa ei käytetä staattista tarkistusta, eivät ole tyypitettyjä. Yhtä mieltä eri määritelmät ovat siitä, että staattinen tarkistus tehdään ennen ohjelman suoritusta ja tätä menetelmää useat funktio-ohjelmointikielet käyttävät. Termit vahva ja heikko tyypitys liittyvät siihen, miten turvallinen kieli on tyyppien suhteen (type safety). Cardellin (1996) mukaan kieli on vahvasti tyypitetty, jos se tunnistaa virheet niin hyvin, ettei niistä seuraa virhetilanteita ohjelman ajon aikana. Heikosti tyypitetyllä kielellä kirjoitetut ohjelmat taas saattavat kaatua ajon aikana virhetilanteen vuoksi.

Hudakin (1989) mukaan pattern matching tarkoittaa sitä, että sama funktio voidaan määritellä useilla eri parametreilla ja versioista valitaan se, joka mihinkin tilanteeseen sopii. Tarkastellaan pattern matchingia seuraavan esimerkin avulla:

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Yllä on kuvattu funktio `fib`, joka laskee n:nnen luvun Fibonaccin lukujonosta (`fib 1` antaa ensimmäisen Fibonaccin luvun, `fib 2` antaa toisen Fibonaccin luvun jne.). Esimerkissä funktiosta on annettu kolme erilaista versiota, joista oikea valitaan annetun parametrin mukaan. Jos parametrina annetaan 0, valitaan ensimmäinen versio ja jos parametrina annetaan 1, valitaan toinen versio. Jos sen sijaan parametrina annetaan mikä tahansa muu positiivinen kokonaisluku, valitaan kolmas versio. Pattern matchingin avulla samasta funktiosta voidaan tehdä versioita myös eri tyyppisillä parametreilla.

Siirrännän toteuttaminen funktio-ohjelmointikielillä on haastavaa. Siirräntä (Input/Output) on tiedonsiirtoa, jonka avulla ohjelma ottaa vastaan ja antaa syötteitä, esimerkiksi tulostaa näytölle tekstiä tai ottaa vastaan käyttäjän syötteitä näppäimistöllä. Näytölle tulostaminen ja muu siirräntä ovat Peyton Jonesin (2001) mukaan useissa kielissä jonkin ohjelman suorituksen sivuvaikutus, mutta puhtaissa funktio-ohjelmointikielissä sivuvaikutukset pyritään eliminoimaan. Joidenkin funktio-ohjelmointikielten, kuten Lispin ja Standard ML:n, siirräntä on toteutettu sivuvaikuksellisesti, mutta tämä tekee niistä epäpuhtaita funktio-ohjelmointikieliä. Luvussa 2.3 esitellään Haskellin siirrännän toteutus, joka oli ensimmäinen täysin funktionaalinen tapa toteuttaa se.

2.3 Haskellin ominaisuuksia

Haskell on funktio-ohjelmointikieli, jonka ensimmäinen versio julkaistiin vuonna 1990. Peyton Jones (2003) kuvaa Haskellin alkumetrejä artikkelin esipuheessa. Haskell kehitettiin Functional Programming Languages and Computer Architecture (FPCA) 1987 -konferenssin seurauksena. Konferenssissa muodostettiin komitea, jonka tarkoituksena oli luoda funktio-ohjelmointi kieli, joka toisi yhteen ominaisuuksia jo olemassa olevista funktio-ohjelmointikielistä ja loisi pohjan tulevaisuuden funktio-ohjelmoinnille.

Haskell toteuttaa luvussa 2.2 kuvatut funktio-ohjelmointikielille tyypilliset piirteet. Uutena ominaisuutena aiempiin funktio-ohjelmointikieliin verrattuna Haskell sisälsi muun muassa täysin funktionaalisen siirrännän toteutuksen. Hudakin ym. (2007) mukaan Haskellin siirräntä on toteutettu monadien avulla. Monadit perustuvat kategoriateoria-nimiseen matematiikan osa-alueeseen. Niihin ei perehdytä tässä tutkielmassa syvällisemmin, sillä niiden tarkka toi-

minta ei ole tutkielman aiheen kannalta oleellinen. Peyton Jonesin (2001) mukaan monadeihin perustuva siirräntä oli huomattavasti monipuolisempi ja tehokkaampi kuin aikaisemmat yritykset funktionaaliseen siirräntään. Lisäksi sen kehitys antoi pohjan myös esimerkiksi rinnakkaisuuden ja poikkeusten toteutukseen Haskellilla, joita käsitellään luvussa 4.1.

Haskell oli myös ensimmäinen ohjelmointikieli, jossa oli tyyppiluokkia. Hudakin ym. (2007) mukaan ne ovat yksi Haskellin tunnuksenomaisimmista piirteistä. Tyyppiluokkien avulla voidaan varmistaa, että tietyllä tyypillä on jokin haluttu ominaisuus. Esimerkiksi jonkin tyypin toteuttaessa tyyppiluokan `Eq` voidaan olla varmoja, että sen tyyppisiä muuttujia voidaan vertailla keskenään. Lisäksi tyyppimuuttujien avulla voidaan kehittää abstraktimpia funktioita, jotka toimivat esimerkiksi kaikille tyypeille, jotka toteuttavat tyyppiluokan `Eq`. Tällöin ei tarvitse toistaa samankaltaista funktiota kaikille tyypeille erikseen, mutta voidaan silti olla varmoja, että niillä on kaikilla tarvittavat ominaisuudet.

Haskellista on useita eri toteutuksia, joista Fausakin (2017) mukaan suosituin on Glasgow Haskell Compiler (GHC). GHC sisältää mahdollisuuden muun muassa debuggaukseen sekä testaukseen ja lisäksi siinä on kattava tuki laajennoksille ja kirjastoille. Se tukee myös web-palvelinohjelmoinnissa hyödyllisiä kirjastoja, joita esitellään luvussa 4.1. Yamamoton (2017) mukaan GHC:n huono puoli on se, että sen kääntämisnopeus on hidas.

3 Web-palvelimet

Palvelin on ohjelma tai ohjelmisto, joka tarjoaa erilaisia palveluita asiakkaille. Asiakas on tässä yhteydessä toinen ohjelma, joka käyttää palveluita joko paikallisesti tai verkon välityksellä. Asiakkaan käytettävissä olevat palvelut riippuvat palvelimen tyypistä. Comerin ja Stevensin (2001) mukaan useimmat palvelimet noudattavat asiakas-palvelin -mallia, jonka mukaisesti palvelimen toiminta on karkeasti seuraavanlainen:

1. Palvelin odottaa yhteyttä asiakkaalta
2. Asiakas ottaa yhteyden ja lähettää palvelimelle jonkin pyynnön
3. Palvelin prosessoi pyynnön ja lähettää sen mukaisen vastauksen asiakkaalle
4. Yhteys katkaistaan ja palvelin jää odottamaan uusia yhteyksiä.

Palvelimia on useita eri tyyppisiä, kuten web-, sähköposti-, nimi-, sovellus- ja pelipalvelimia. Tässä tutkielmassa keskitytään vain web-palvelimiin. Tämä luku rakentuu kahdesta osasta: web-palvelinten määrittelystä ja niiden toiminnan tarkemmasta kuvauksesta luvussa 3.1 sekä web-palvelinten ohjelmoinnin periaatteiden esittelystä luvussa 3.2.

3.1 Web-palvelimen toiminta

Web-palvelin on palvelin, joka jakaa asiakkaille tiedostoja, kuten websivuja, verkon yli. Web-palvelimen asiakkaana toimii usein web-selain. Web-palvelimet käsittelevät Hypertext Transfer Protocol (HTTP) -protokollan mukaisia pyyntöjä. HTTP-protokollan uusin versio on HTTP/2 ja tämän luvun tiedot HTTP/2-protokollasta perustuvat Belshen, Thomsonin ja Peonin (2015) dokumentaatioon. Esimerkki HTTP/2-protokollan mukaisesta pyynnöstä:

```
:method = GET
:scheme = https
:path = /index.html
:authority = www.esimerkki.fi
date = Tue, 6 Feb 2018 08:15:42 GMT
```

Ensimmäisellä rivillä on metodi, joka kertoo palvelimelle mitä sen pitää tehdä. Esimerkissä

metodi on GET, jonka avulla pyydetään palvelimelta dataa. Muita metodeja ovat esimerkiksi HEAD, joka toimii muuten samoin kuin GET-metodi, mutta palauttaa vain HTTP-vastauksen ilman pyydettyä dataa, ja POST, jonka avulla lähetetään tietoa palvelimelle. Belshen, Thomsonin ja Peonin (2015) mukaan web-palvelimen on toteutettava vähintään GET ja HEAD -metodit. Esimerkin toisella rivillä on palvelimen osoitteen skeema, joka HTTP-protokollan yhteydessä on aina joko http tai https riippuen siitä, että onko yhteys suojattu. Kolmannella rivillä on tiedosto, jota palvelimelta pyydetään. Esimerkin neljännellä rivillä on palvelimen osoite. Pyynnön loput rivit, joita esimerkissä on vain yksi, ovat otsakkeita, jotka ovat useimmiten vapaaehtoisia.

Käsiteltyään saamansa pyynnön web-palvelin lähettää asiakkaalle vastauksen. Esimerkki HTTP/2-protokollan mukaisesta vastauksesta:

```
:status 200
date = Tue, 6 Feb 2018 08:15:50 GMT
content-type = text/html

<html>
<head>
    <title>Esimerkki</title>
</head>
<body>
    Esimerkkisivu
</body>
</html>
```

Vastausesimerkin ensimmäisellä rivillä on status-koodi, joka kertoo pyynnön käsittelyn onnistumisesta. Esimerkissä koodi on 200, joten pyynnön käsittely on onnistunut. Muut koodit kertoisivat, minkälainen virhe käsittelyssä on tapahtunut. Myös vastauksessa voi olla otsakkeita, joita esimerkissä on kaksi. Lopuksi vastaus sisältää pyydetyn tiedoston, mikäli pyynnön käsittely onnistui. Palvelimen tulee pystyä tuottamaan eri tilanteisiin sopivia vastauksia.

Marlow (2002) määrittelee joitain vaatimuksia, jotka web-palvelimen on vähintään täytettä-

vä:

- Virheenkäsittely: palvelimen tulee pystyä palautumaan virhetilanteista nopeasti
- Rinnakkaisuus (concurrency): usean asiakkaan tulee pystyä ottamaan yhteys palvelimeen samanaikaisesti
- Pieni viive (latency): palvelimen tulee pystyä käsittelemään pyyntöjä mahdollisimman pienellä viiveellä

Virheidenkäsittely on tärkeää, jotta palvelin palautuu virhetilanteista nopeasti eikä kaadu. Lisäksi Marlown mukaan palvelin pitäisi voida konfiguroida uudelleen ilman että sen käyttöä katkaistaan. Web-palvelimen rinnakkaisen käytön mahdollistaminen on olennaista, koska muuten esimerkiksi palvelimen jakamalle websivulle pääsisi vain yksi käyttäjä kerrallaan. Web-palvelimen pitää pystyä käsittelemään suuriakin määriä pyyntöjä samanaikaisesti.

Pyyntöjen suuren määrän mahdollistaminen vaatii myös, että ne käsitellään mahdollisimman pienellä viiveellä. Marlown (2002) mukaan pieni viive parantaa palvelimen tehokkuutta, mutta myös auttaa suojaamaan sitä palvelunestohyökkäyksiltä. Jos palvelin pystyy käsittelemään pyynnöt mahdollisimman nopeasti, palvelunestohyökkäyksen syöttämät pyynnöt hidastavat palvelimen toimintaa vähemmän. Pienen viiveen lisäksi palvelimen tulee voida katkaisemaan yhteydet asiakkailta (timeout), joilla esimerkiksi kestää liian kauan vastata. Näin saadaan vapautettua suoritustehoa muilla asiakkaille. Snoymanin (2011) mukaan myös aikakatkaisu auttaa puolustautumaan palvelunestohyökkäyksiä vastaan, sillä turhat yhteydet suljetaan.

Lisäksi palvelimen on pidettävä huolta tietoturvasta. Comerin ja Stevensin (2001, s. 11) mukaan palvelimen pitäisi pystyä varmistamaan asiakkaan identiteetti, selvittämään onko asiakkaalla oikeus päästä käsiksi pyytämäänsä tiedostoon ja pitämään huolta, ettei kukaan ilma oikeuksia pääse käsiksi tietoon tai muuttamaan sitä.

Marlown (2002) mukaan web-palvelimen tulisi pitää myös kirjanpitoa siihen otetuista yhteyksistä ja tapahtuneista virhetilanteista. Yhteyksien kirjanpitoon merkitään usein vähintään aika, jolloin yhteys on muodostettu, asiakkaan osoite sekä osoite, jota pyydettiin, vastauksen status-koodi, aika, joka pyynnön suorittamiseen kului ja siirretyn tiedon määrä. Kirjanpidon avulla voidaan muun muassa seurata palvelimen käyttöä ja tehokkuutta. Virheiden kirjanpi-

toon sen sijaan merkitään vähintään aika, jolloin virhe tapahtui ja jonkinlainen virhekoodi tai kuvaus virheestä. Virhekirjanpidon avulla voidaan seurata, millaisia virhetilanteita ilmenee ja korjata niiden aiheuttajia.

3.2 Web-palvelimen ohjelmointi

Marlown (2002) mukaan web-palvelimen ydinrakenne koostuu karkeasti kahdesta osasta: HTTP-protokollan toteutuksesta, ja pyyntöjen vastaanotosta sekä käsittelystä. Osa joka käsittelee HTTP-pyyntöjä, erottelee pyynnön osat toisistaan, jotta niitä voidaan käsitellä. Palvelimen tulee siis pystyä erottelemaan tietyt osat pyynnöstä ja toimimaan niiden mukaisesti. Lisäksi palvelimen täytyy generoida HTTP-protokollan mukaisia vastauksia riippuen pyynnön käsittelyn tuloksesta. Osa, joka vastaanottaa pyyntöjä on yhteydessä verkkoon. Goralskin (2017) mukaan web-palvelin odottaa passiivisesti ja seuraa jotain porttia tai joitain portteja, kunnes asiakas lähettää pyynnön. Saatuaan pyynnön palvelin palvelin laittaa alulle sen käsittelyn ja jää odottamaan lisää pyyntöjä.

Rinnakkaisuuden toteuttamiseen on useita eri tapoja, joista Marlow (2002) esittelee seuraavia: erilliset prosessit, yksi prosessi I/O-kanavoinnilla (I/O multiplexing), käyttöjärjestelmän säikeet ja käyttäjätilan (user-space) säikeet. Erillisillä prosesseilla rinnakkaisuus toteutetaan niin, että yksi ”pääprosessi” ottaa vastaan uusia yhteyksiä ja tekee jokaiselle yhteydelle oman prosessin. Tämän tavan hyötyjä ovat sen hyvä skaalautuvuus monelle prosessorille ja helppo toteutus. Lisäksi erilliset prosessit kestävät virhetilanteita hyvin, sillä virhe yhdessä prosessissa ei vaikuta muihin prosesseihin. Huonoja puolia metodissa ovat prosessien vaatima suoritusteho, pidempi viive ja prosessien keskinäisen kommunikaation hankaluus. Useiden prosessien metodia käyttävät esimerkiksi suositut Apache-palvelimet.

Yhden prosessin metodissa rinnakkaisuus toteutetaan suoraan käyttämällä useita I/O-kanavia. Tämän metodin etuja ovat nopeus ja pieni viive. Sen huonoja puolia ovat virheenkäsittelyn puutteellisuus ja vaikea toteutus. Käyttöjärjestelmän säikeillä toteutettu rinnakkaisuus muistuttaa ominaisuuksiltaan paljon useilla prosesseilla toteutettuna. Säikeet ovat usein hieman nopeampia kuin prosessit, ja niiden keskinäinen kommunikaatio on helpompaa, mutta ne käsittelevät virheitä huonommin.

Käyttäjätilan säikeillä toteutettu rinnakkaisuus on yhdistelmä yhden prosessin ja käyttöjärjestelmäsäikeiden metodeista. Se toimii yhdessä prosessissa, jonka sisällä toimii useita säikeitä. Tämän metodin etuja ovat I/O-kanavoinnin nopeus yhdistettynä säikeiden toteutuksen helppouteen. Tätä metodia käyttää Concurrent Haskell, jota käsitellään luvussa 4.1.

Virhetilanteet käsitellään useissa ohjelmointikielissä poikkeusten avulla. Poikkeukset voidaan jakaa kahteen kategoriaan: tahdistetut (synchronous) ja tahdistamattomat (asynchronous) poikkeukset. Marlown ym. (2001) mukaan tahdistetut poikkeukset ovat ohjelman itsensä suorituksesta johtuvia, kun taas tahdistamattomat poikkeukset johtuvat jostain ohjelman ulkoisesta tapahtumasta. Tahdistetut poikkeukset tapahtuvat jossain tietyssä kohtaa ohjelman suoritusta, kun taas tahdistamattomat voivat tapahtua missä vain ohjelman suorituksen vaiheessa. Tahdistamattomat poikkeukset ovat erityisen tärkeitä web-palvelinohjelmoinnissa, sillä palvelimen täytyy pystyä reagoimaan esimerkiksi käyttäjästä johtuvaan poikkeukseen riippumatta siitä, missä vaiheessa suoritusta se on. Lisäksi Marlown (2002) mukaan tahdistamattomien poikkeusten (asynchronous exception) avulla voidaan toteuttaa esimerkiksi aikakatkaisut ja niiden avulla palvelimen ylläpitäjä voi tehdä muutoksia palvelimeen ilman, että palvelimeen tulee käyttökatkos.

4 Haskell web-palvelinohjelmoinnissa

Nyt kun ollaan tarkasteltu Haskellin ominaisuuksia ja web-palvelinten toimintaa, voidaan tutkia Haskellin käyttöä web-palvelinohjelmoinnissa. Ensin, luvussa 4.1, selvitetään miten luvussa 2 esitellyt Haskellin ominaisuuden soveltuvat luvussa 3 esiteltyjen web-palvelinten toteuttamiseen. Toiseksi, luvussa 4.2, vertaillaan Haskellia joihinkin muihin kieliin, joita web-palvelinohjelmoinnissa käytetään.

4.1 Haskellin soveltuvuus web-palvelinohjelmointiin

Haskellilla on kehitetty useita web-palvelinkirjastoja. Marlow (2002) esittelee artikkelissaan Haskellilla kehittämäänsä web-palvelinta Haskell web server. Hänen kehittämänsä palvelin ei ole enää ajankohtainen, mutta moni uudempi Haskellilla toteutettu palvelinkirjasto, kuten Warp ja Snap, pohjautuu sen toteutukseen. Uudempia web-palvelinkirjastoja ovat muun muassa Warp, Mighttpd ja Snap. Warp oli Snoymanin (2011) mukaan tehokkain Haskellilla toteutettu web-palvelinkirjasto ja Yamamoton (2016) mukaan sen tehokkuus on kasvanut edelleen HTTP/2-protokollaan siirtymisen myötä. Yamamoton (2011) mukaan Mighttpd on valmis web-palvelin, joka on rakennettu Warpin avulla. Snap on web-ohjelmointikirjasto, johon Collinsin ja Beardsleyn (2011) mukaan kuuluu sisäänrakennettu web-palvelin.

Haskell web serverin rinnakkaisuus on toteutettu Concurrent Haskell -nimisen laajennoksen avulla, jonka ovat kehittäneet Peyton Jones, Gordon ja Finne (1996). Concurrent Haskellissa rinnakkaisuus on toteutettu käyttäjätilan säikeillä, joita käsiteltiin luvussa 3.2 ja se on sisällytetty GHC-kääntäjään. Marlown (2002) mukaan Concurrent Haskellin rinnakkaisuusmalli on kevyt ja pitää viiveen pienenä useille asiakkaille samanaikaisesti. Samaa menetelmää käyttää Yamamoton (2017) mukaan myös Warpin HTTP/1.1-protokollaa käyttävä versio. Hänen mukaansa menetelmä on erittäin tehokas HTTP/1.1-versiolle ja toisessa artikkelissaan (2016) hän toteaa, että HTTP/1.1-koodin kirjoittaminen on yksinkertaisempaa tällä menetelmällä kuin tapahtumiin (event) perustuvilla menetelmillä. Myös Collinsin ja Beardsleyn mukaan Haskellilla toteutettu rinnakkaisuus on erittäin tehokas ja pystyy käsittelemään tuhansia yhteyksiä samanaikaisesti.

Yamamoton (2016) mukaan Concurrent Haskellin rinnakkaisuuden toteutus ei ollut kuitenkaan optimaalisin HTTP/2-protokollaa varten, joten Warpin sitä käyttävään versioon tehtiin muutoksia rinnakkaisuuden suhteen. Tässä versiossa säikeitä on luotu jo valmiiksi, sen sijaan että jokaista vastaanotettua pyyntöä varten luotaisiin oma. Saapuneet pyynnöt laitetaan jonoon, josta säikeet ottavat niitä käsittelyyn ja siirtävät valmiit vastaukset omaan jonoonsa, josta ne lähetetään asiakkaalle. Tämä menetelmä teki rinnakkaisuudesta vielä tehokkaampaa säilyttäen toteutuksen keveyden.

Marlown (2002) mukaan Concurrent Haskellin huonona sen puolena on se, ettei se skaalaudu hyvin usealle prosessorille. Ongelman korjaa kuitenkin myöhemmin kehitetty Parallel Haskell, jota esittelee esimerkiksi Marlow (2012). Parallel Haskell mahdollistaa usean prosessorin samanaikaisen käytön. Lisäksi Voellmyn ym. (2013) työn seurauksena GHC:hen kehitettiin uusi siirrännän hallinta, jonka avulla se voi hyödyntää yli 40 ydintä. Haskell soveltuu siis hyvin useiden yhteyksien samanaikaiseen käsittelyyn, mikä on web-palvelinohjelmoinnissa tärkeää, ja skaalautuu monelle prosessorille, jolloin palvelinta voi laajentaa isompiinkin tarpeisiin.

Virheiden käsittely toteutettiin Haskell web serverille laajennoksen nimeltä Asynchronous Exceptions avulla. Laajennos mahdollistaa nimensä mukaisesti tahdistamattomien poikkeusten käytön ja sen esittelivät alun perin Marlow ym. (2001). Sen avulla on toteutettu myös Haskell web serverin aikakatkaisut. Snoymanin (2011) mukaan Warp käyttää optimoidumpaa aikakatkaisukirjastoa, jota oli mukana kehittämässä muun muassa Snapin kehittäjä ja ylläpitäjä Gregory Collins. Kirjasto käyttää tahdistamattomien poikkeusten sijaan aikakatkaisuista huolehtivaa erillistä säiettä, joka pitää huolen muiden säikeiden toiminnan katkaisemisesta tarvittaessa. Snoymanin mukaan tämä menetelmä on tehokas ja kevyt.

Tahdistamattomia poikkeuksia käytettiin Haskell web serverissä myös palvelimen ajonaikaisen uudelleen konfiguroimisen toteutukseen. Marlow (2002) toteutti palvelimeen ajonaikaisesti tehdyt muutokset niin, että ne eivät vaikuta jo käytössä olevien säikeiden toimintaan, vaan pelkästään uusiin muutoksen jälkeen luotaviin säikeisiin. Palvelimen konfiguraatiota muutetaan muokkaamalla tekstitiedostoa. Muokkauksen jälkeen muita säikeitä luovalle, pyyntöjä vastaanottavalle säikeelle lähetetään tahdistamaton poikkeus, jonka seurauksena säie lukee konfiguraatiotiedoston uudelleen. Tämän jälkeen kaikki uudet luodut säikeet ovat

uuden konfiguraation mukaisia.

Haskell web serverin kirjanpito sekä palvelimen tapahtumista, että virhetilanteista toteutettiin erillisen säikeen avulla. Yksi kirjanpitosäie huolehtii kirjanpidosta ja muut säikeet kommunikoivat sen kanssa. Marlown (2002) mukaan erillinen kirjanpitosäie nopeuttaa ja keventää palvelimen toimintaa, sillä muut säikeet voivat käsitellä useita pyyntöjä ilman, että ne kirjaavat toimintaansa itse ylös välissä ja lisäksi erillinen kirjanpitosäie voi kirjoittaa monta merkintää kerralla sen sijaan, että jokainen säie kirjoittaa niitä yksi kerrallaan.

Marlown (2002) Haskell web server käytti HTTP-pyyntöjen parsimiseen ja vastausten generointiin String-tyyppisiä merkkijonoja, jotka Haskellissa ovat merkkejä sisältäviä listoja. Yamamoton, Snoymanin ja Voellmyn (2013) mukaan merkkijonojen toteutus listojen avulla aiheuttaa kaksi ongelmaa niiden käsittelyssä: merkkien lisääminen listaan on turhan iso operaatio ja listat vievät enemmän tilaa kuin ne oikeasti tarvitsevat (overhead). Tästä syystä heidän mukaansa Warp käyttää blaze-html -nimistä kirjastoa, jonka avulla merkkijonojen käsittely on tehokkaampaa. Web-palvelin käsittelee merkkijonoja jatkuvasti, joten niiden käsittelyn tehokkuus vaikuttaa palvelimen tehokkuuteen huomattavasti.

4.2 Haskellilla ohjelmoidut web-palvelimet verrattuna muihin web-palvelimiin

Netcraftin (2018) selvityksen mukaan huhtikuussa 2018 36.94% web-sivuista käytti Microsoftin web-palvelinta, 25.58% Apachen palvelinta, 22.62% nginx-palvelinta ja 1.26% Googlen palvelinta. Selvityksessä oli mukana noin 1.8 miljardia web-sivua. Kaikki edellä mainitut palvelimet on ohjelmoitu joko C- tai C++-kielellä. Seuraavaksi verrataan Haskellilla ohjelmoituja web-palvelimia näihin suosituimpiin palvelimiin.

Marlown (2002) Haskell web server ei testeissä pärjännyt tehokkuudeltaan Apache-palvelimelle, mutta oli tehokkuudeltaan muuten sopiva pienempiin projekteihin. Haskell web server ei kuitenkaan ollut vielä kovin optimoitu ja muut Haskellilla ohjelmoidut palvelimet pärjäävätkin vertailuissa paremmin. Esimerkiksi Collinsin ja Beardsleyn (2011) mukaan Snap-kirjaston web-palvelin oli 40% nopeampi kuin Apache-palvelin kun se käytti kirjanpitoa ja jopa 2.5% Apache-palvelinta nopeampi, kun kirjanpito oli poissa päältä.

Snoymanin (2011) mukaan Warp oli huomattavasti tehokkaampi kuin muut Haskellilla ohjelmoidut web-palvelimet. Yamamoto (2016) vertasi Warpin ja sen pohjalta kehitetyn Mighttpd:n suorituskykyä nginx-palvelimeen. Hän testasi erikseen kustakin palvelimesta versiota HTTP/-1.1- ja HTTP/2-protokollalle. Testauksen tuloksena oli, että HTTP/2-versiot olivat tehokkaampia kuin HTTP/1.1-versiot ja Warp sekä Mighttpd pärjäsivät paremmin kuin nginx, Mighttpd:n HTTP/1.1-versiota lukuun ottamatta. Warpin HTTP/2-versio oli testissä ylivoimainen muihin verrattuna.

Haskellilla ohjelmoidut web-palvelimet pärjäävät siis tehdyissä testeissä erittäin hyvin suosituimpiin palvelimiin verrattuna. Yamamoto (2016) tosin mainitsee, että eri web-palvelinten testaaminen reilusti on hankalaa, sillä eri palvelimilla on hyvin erilaisia ominaisuuksia. Testeistä voi kuitenkin päätellä, että ainakin nopeuden suhteen monet Haskellilla ohjelmoidut web- palvelimet on kilpailukykyinen verrattuna suosituimpiin palvelimiin.

Haskellin etuna on Collinsin ja Beardsleyn (2011) mukaan myös tiiviit ja lyhyet ohjelmat. Koodin kirjoittaminen vie ohjelmoijalta vähemmän aikaa ja se vie myös vähemmän tilaa kiintolevyltä. Pienen kokonsa vuoksi esimerkiksi Warp-palvelinta voi Snoymanin (2011) mukaan pyörittää esimerkiksi pienissä sulautetuissa järjestelmissä, mutta se skaalautuu hyvin myös isoihin tarpeisiin.

5 Yhteenveto

Haskell on web-palvelinohjelmoinnissa hyödyllinen erityisesti tehokkuutensa ja tiiviin koodinsa vuoksi. Sillä ohjelmoidut palvelimet ovat pienikokoisia ja kevyitä ja ne myös skaalautuvat hyvin suuriinkin tarpeisiin. GHC mahdollistaa useiden hyödyllisten kirjastojen käytön ja sisältää tuen tehokkaaseen rinnakkaisuuteen. Haskellilla on siis potentiaalia web-palvelinohjelmoinnissa, vaikka se ei olekaan suuressa käytössä alalla.

Apachen, Microsoftin ja muiden paljon käytössä olevien palvelinten takana on paljon isompi koneisto kuin minkään Haskellilla kehitetyn palvelimen. Niillä on enemmän näkyvyyttä ja markkinointia ja toisaalta ne ovat todennäköisesti myös paremmin testattuja ja yhteensopivia useampien eri web-kehitystekniikoiden kanssa. Ei siis ole ihme, että Haskellilla ohjelmoidut palvelimet eivät tehokkuudestaan huolimatta ole alansa kärjessä.

Valmiiksi käytössä olevia palvelimia on jossain määrin hankala vaihtaa täysin eri tyyppisiksi ja uuden palvelimen aloittavat todennäköisesti valitsevat mieluummin paljon käytössä olevan palvelimen kuin tuntemattomamman, jonka mahdollisista ongelmista ei ole paljon tietoa. Tilanne ei siis varmaankaan ole muuttumassa lähiaikoina, mutta potentiaalia Haskellilla olisi isompaan käyttöön.

Jatkotutkimusaiheita aiheeseen liittyen ovat web-sisällön ja muunlaisten palvelinten tuottaminen Haskellilla. Kirjastoja web-sisällön tuottamiseen Haskellille löytyy useita ja luonnollinen seuraava askel tästä tutkielmasta olisi selvittää miten ne vertautuvat muihin vastaaviin menetelmiin. Toinen mahdollinen laajennos aiheeseen olisi selvittää minkälaista potentiaalia Haskellilla olisi muun tyyppisten palvelinten, kuten esimerkiksi sähköpostipalvelinten, toteuttamiseen.

Lähteet

Belshe, Mike, Martin Thomson ja Roberto Peon. 2015. “Hypertext transfer protocol version 2 (http/2)”. ISSN: 2070-1721. https://tools.ietf.org/pdf/rfc7540.pdf.

Bird, Richard, Geraint Jones ja Oege De Moor. 1997. “More haste, less speed: lazy versus eager evaluation”. *Journal of Functional Programming* 7 (5): 541–547.

Cardelli, Luca. 1996. “Type Systems”. *ACM Comput. Surv.* (New York, NY, USA) 28, numero 1 (maaliskuu): 263–264. ISSN: 0360-0300. doi:10.1145/234313.234418.

Cardelli, Luca, ja Peter Wegner. 1985. “On Understanding Types, Data Abstraction, and Polymorphism”. *ACM Comput. Surv.* (New York, NY, USA) 17, numero 4 (joulukuu): 471–523. ISSN: 0360-0300. doi:10.1145/6041.6042.

Collins, G., ja D. Beardsley. 2011. “The Snap Framework: A Web Toolkit for Haskell”. ID: 1, *IEEE Internet Computing* 15 (1): 84–87. doi:10.1109/MIC.2011.21.

Comer, Douglas E., ja David L. Stevens. 2001. *Internetworking with TCP/IP. Vol. 3, Client-server programming and applications : Linux/POSIX sockets version.* Upper Saddle River (N.J.): Prentice Hall. ISBN: 0-13-032071-4 sidottu.

Fausak, Taylor. 2017. “2017 State of Haskell Survey Results”. http://taylor.fausak.me/2017/11/15/2017-state-of-haskell-survey-results/.

Field, Anthony J., ja Peter G. Harrison. 1988. *Functional programming.* Wokingham: Addison-Wesley. ISBN: 0-201-19249-7.

Fischer, Sebastian, Oleg Kiselyov ja Chung-chieh Shan. 2009. “Purely Functional Lazy Nondeterministic Programming”. Teoksessa *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming,* 11–22. ICFP ’09. Edinburgh, Scotland: ACM. ISBN: 978-1-60558-332-7. doi:10.1145/1596550.1596556.

Goralski, Walter. 2017. *The illustrated network: how TCP/IP works in a modern network.* Morgan Kaufmann.

Hudak, Paul. 1989. “Conception, evolution, and application of functional programming languages”. *ACM Computing Surveys (CSUR)* 21 (3): 359–411. doi:10.1145/72551.72554.

Hudak, Paul, John Hughes, Simon Peyton Jones ja Philip Wadler. 2007. “A history of Haskell: being lazy with class”, 55. HOPL III. doi:10.1145/1238844.1238856.

Lloyd, John W. 1994. “Practical Advantages of Declarative Programming.” Teoksessa *GULP-PRODE (1),* 18–30.

Marlow, Simon. 2002. “Developing a high-performance web server in Concurrent Haskell”. *Journal of Functional Programming* 12 (4-5): 359–374. doi:10.1017/S095679680200432X.

———. 2012. “Parallel and concurrent programming in Haskell”. Teoksessa *Central European Functional Programming School,* 339–401. Springer.

Marlow, Simon, Simon Peyton Jones, Andrew Moran ja John Reppy. 2001. “Asynchronous Exceptions in Haskell”. Teoksessa *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation,* 274–285. PLDI ’01. Snowbird, Utah, USA: ACM. ISBN: 1-58113-414-2. doi:10.1145/378795.378858.

Netcraft. 2018. “April 2018 Web Server Survey”. https://news.netcraft.com/archives/2018/04/26/april-2018-web-server-survey.html.

Peyton Jones, Simon. 2001. “Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell”. Teoksessa *Engineering theories of software construction,* 47–96. IOS Press, tammikuu. ISBN: ISBN 1 58603 1724. https://www.microsoft.com/en-us/research/publication/tackling-awkward-squad-monadic-inputoutput-concurrency-exceptions-foreign-language-calls-haskell/.

Peyton Jones, Simon L., toimittanut. 2003. *Haskell 98 language and libraries : the revised report.* Cambridge: Cambridge University Press.

Peyton Jones, Simon L., Andrew D. Gordon ja Sigbjorn Finne. 1996. “Concurrent Haskell”. Teoksessa *POPL.* doi:10.1145/237721.237794.

Snoyman, M. 2011. “Warp: A Haskell Web Server”. ID: 1, *IEEE Internet Computing* 15 (3): 81–85. doi:10.1109/MIC.2011.70.

Voellmy, Andreas Richard, Junchang Wang, Paul Hudak ja Kazuhiko Yamamoto. 2013. “Mio: A High-performance Multicore Io Manager for GHC”. *SIGPLAN Not.* (New York, NY, USA) 48, numero 12 (syyskuu): 129–140. ISSN: 0362-1340. doi:10.1145/2578854.2503790.

Yamamoto, Kazuhiko. 2011. “Mighttpd–a High Performance Web Server in Haskell”. *The Monad. Reader Issue 19: Parallelism and Concurrency:* 5.

———. 2016. “Experience report: developing high performance HTTP/2 server in Haskell”. doi:10.1145/2976002.2976006.

———. 2017. “Network Protocol Programming in Haskell”. https://pdfs.semanticscholar.org/9555/c7e7e2b81820d4b61d908fadda807f6196dc.pdf.

Yamamoto, Kazuhiko, Michael Snoyman ja Andreas Voellmy. 2013. “Warp”. *The Performance of Open Source Applications.* http://www.aosabook.org/en/posa/warp.html.