Haaga-Helia
ammattikorkeakoulu Oy

Asynkronisen ohjelmoinnin tekniikoiden vertailu Node.js-ympäristössä

Petka Antonov

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2015

Haaga-Helia
ammattikorkeakoulu Oy

Tiivistelmä

22.11.2015

Tekijä(t)
Petka Antonov

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön otsikko	**Sivu- ja liitesivumäärä**
Asynkronisen ohjelmoinnin tekniikoiden vertailu Node.js-ympäristössä	45 + 9

Opinnäytetyön otsikko englanniksi
Comparison of Asynchronous Programming Techniques in Node.js

Työn tarkoitus oli selvittää, mitä tekniikoita asynkroniseen ohjelmointiin on saatavilla, mitä heikkouksia ja vahvuuksia niillä on ja millaisiin tilanteisiin kukin tekniikka sopii parhaiten. Asynkronisella ohjelmoinnilla tarkoitetaan sitä, ettei ohjelman käskyjä suoriteta siinä järjestyksessä, missä ne esiintyvät lähdekoodissa.

Työssä esiteltiin aluksi synkroninen ja asynkroninen ohjelmointi, sekä asynkroniselle ohjelmoinnille saatavilla olevat tekniikat ympäristöstä huolimatta. Työn vertailu rajattiin kolmeen Node.js-ympäristössä saatavilla olevan asynkronisen ohjelmoinnin tekniikan vertailuun: takaisinkutsufunktioihin, async-apufunktiokirjastoon sekä lupauksiin (promises).

Vertailussa kuvailtiin kolme erilaista ongelmaa, joiden ratkaisut toteutettiin käyttäen kutakin vertailtavaa asynkronisen ohjelmoinnin tekniikkaa. Jokaisella ongelmalla oli myös erilainen samanaikaisuusvaatimus. Vertailun mittareina käytettiin käyttäjäkoodin luettavuutta, kompleksisuutta, suorituskykyä sekä käytettävyyttä.

Tuloksista kävi ilmi, että ongelmien eri vaatimuksista huolimatta lupaukset osoittautuivat parhaaksi tekniikaksi, kun kaikkien mittareiden yhteisvaikutus otetaan keskimäärin huomioon. Lupaukset käyttivät kuitenkin hieman enemmän muistia kuin muut tekniikat.

Asiasanat
asynkroninen ohjelmointi, javascript, node.js, ohjelmointitekniikka

Haaga-Helia
University of Applied Sciences Ltd.

Abstract

22 November 2015

Author(s)
Petka Antonov

Degree programme
Business Information Technology

Report/thesis title	**Number of pages and appendix pages**
Comparison of Asynchronous Programming Techniques in Node.js	45 + 9

The purpose of this thesis was to find out what asynchronous programming techniques are available, what are their weaknesses and strengths and to what situations each technique is best suited for. Asynchronous programming means that a program's instructions are not executed in the same order as they appear in the source code of the program.

First synchronous and asynchronous programming and the techniques available for asynchronous programming regardless of environment were introduced. The comparison was restricted to three asynchronous programming techniques available in Node.js: callback functions, the async helper function library and promises.

For the comparison three different problems were described whose solutions were implemented using each of the selected techniques to be compared. Each problem had a different concurrency requirement. The criteria used for the comparison were readability and complexity of user code, performance and usability.

The results indicated that despite the differences in requirements of the problems, promises were the best technique when all the criteria are considered together overall. On the other hand, promises used slightly more memory than the other techniques.

Keywords
asynchronous programming, javascript, node.js, programming technique

Sisällys

1 Johdanto

Asynkroninen ohjelmointi on tekniikka, jolla saavutetaan ohjelmassa samanaikaisuutta luopumalla vaatimuksesta, että ohjelmalausekkeet pitää suorittaa tietyssä järjestyksessä. Sitä voidaan pitää synkronisen ohjelmoinnin vastakohtana, jossa ohjelman käskyt suoritetaan ajallisesti samassa järjestyksessä kuin ne esiintyvät lähdekoodissa. Asynkroninen ohjelmointi on hiljattain noussut erittäin suosituksi ja tärkeäksi ohjelmointimalliksi.

Tyypillisesti synkronisessa ohjelmoinnissa samanaikaisuutta ei joko saavuteta ollenkaan tai se saavutetaan monisäikeisyyden (multithreading) avulla. Käyttäjäkokemuksen kannalta erityisesti graafiset käyttöliittymäohjelmat eivät tarjoa riittävän hyvää käyttökokemusta ilman samanaikaisuutta. Toisaalta monisäikeisyyttä käyttävät ohjelmat ovat erittäin alttiita ohjelmointivirheille ja vaativat yksisäikeiseen ohjelmointiin verrattuna moninkertaisesti enemmän huolellisuutta ja tarkkaavaisuutta. Lisäksi monisäikeiset ohjelmat ovat monimutkaisempia.

Node.js on suosittu alusta, jolla voi toteuttaa palvelinpuolen sovelluksia käyttäen asynkronista ohjelmointia JavaScript-ohjelmointikielellä. Muihin alustoihin verrattuna Node.js:stä tekee erikoisen se, että se ei pääsääntöisesti tarjoa lainkaan synkronisen ohjelmoinnin rajapintoja. Lisäksi Node.js:lle kirjoitetut ohjelmat voidaan ajaa usein sellaisenaan web-selaimessa, sillä web-selainten ohjelmointikielenä toimii myös JavaScript.

Koska asynkroninen ohjelmointi ei ole yhtä yksinkertaista ja helppoa kuin synkroninen ohjelmointi, on sitä helpottamaan kehitetty useita työkaluja ja tekniikoita. Node.js-ympäristössä on saatavilla useita tekniikoita, kuten esimerkiksi takaisinkutsufunktiot, tapahtumankuuntelijat, lupaukset ja apurifunktiokirjastot.

Tämä työ pyrkii vertaamaan suosittuja asynkronisen ohjelmoinnin tekniikoita ja selvittämään mitä heikkouksia ja vahvuuksia kullakin tekniikalla on sekä mihin samanaikaisuusvaatimuksiltaan tai operaatioluonteeltaan erilaisiin tilanteisiin kukin tekniikka soveltuu parhaiten. Tekniikoita verrataan niiden käytettävyyden, suorituskyvyn ja käyttäjäkoodin kompleksisuuden kannalta. Vertailun tuloksia voidaan käyttää ohjelmistoprojektin arkkitehtuurivalintoja arvioitaessa.

Työ on rajattu asynkronisen ohjelmoinnin tekniikoihin käyttöön Node.js-ympäristössä. Koska Node.js-ympäristössä kaikki rajapinnat ovat pääosin asynkronisia ja

asynkronisessa ohjelmoinnissa voidaan käyttää montaa eri tekniikkaa, tulee Node.js-alustalle kehittävien ohjelmistokehittäjien ja –arkkitehtien tuntea tekniikoiden heikkoudet, vahvuudet ja niiden sopivuus erilaisten asynkronisen ohjelmoinnin ongelmien ratkaisuiksi.

1.1 Sanasto ja käsitteet

Abstraktio	Abstraktiolla tai abstraktiotasolla tarkoitetaan tasoa, jolla järjestelmää tarkastellaan. Mitä korkeampi abstraktio tai abstraktiotaso, sitä enemmän monimutkaisia yksityiskohtia se peittää alleen. Sopiva abstraktion taso riippuu asiayhteydestä. Abstraktiot mahdollistavat monimutkaisten järjestelmien hahmottamisen ja hallitsemisen.
Apufunktio	Apufunktiot ovat funktioita, jotka toteuttavat jonkin yksinkertaisen toiminnallisuuden, jota tarvitaan useammassa kuin yhdessä paikassa.
Asynkroninen ohjelmointi	Asynkronisessa ohjelmoinnissa ohjelmalausekkeiden suoritus ei välttämättä etene siinä järjestykessä kuin ne esiintyvät lähdekoodissa.
JavaScript	JavaScript on ohjelmointikieli, jolle on saatavilla toteutus esimerkiksi jokaisessa verkkoselaimessa.
Lupaus	Lupaukset (promise) ovat olioita jotka kuvaavat jotain tapahtumaa tai tulevaisuudessa saatavilla olevaa arvoa.
Moduuli	Moduuli on itsenäinen koodin organisoinnin yksikkö. Moduuli sisältää kaiken tarvittavan ohjelmistokokonaisuuden toiminnallisuuden osa-alueen toteuttamiseksi.
Node.js	Node.js on sovellusalusta, jolle voidaan kehittää palvelinpuolen sovelluksia JavaScript-ohjelmointikielellä.
Pinojäljitys	Pinojäljitys kertoo mitä funktioita on kutsuttu ja missä järjestykssä niin, että ohjelma on päätynyt siihen kohtaan, jossa pinojäljitystä pyydettiin.

Poikkeus	Poikkeus on olio, joka kuvaa virhettä tai virhetilannetta.
Siirräntä	Siirräntä (Input/Output) on eri komponenttien välillä tapahtuvaa tiedonsiirtoa. Yleisimpiä tapauksia ovat kovalevysiirräntä (Disk I/O) ja verkkosiirräntä (Network I/O).
Synkroninen ohjelmointi	Synkronisessa ohjelmoinnissa ohjelmalausekkeiden suoritus etenee siinä järjestyksessä kuin ne esiintyvät lähedkoodissa.
Takaisinkutsufunktio	Takaisinkutsufunktio (callback function) on rajapinnan käyttäjän määrittelemä funktio, jota rajapinnan toteuttaja kutsuu, kun käyttäjän pyytämän operaation suoritus on toteutunut.
Tapahtumankuuntelija	Tapahtumankuuntelija (event listener) on funktio, jota kutsutaan kun tietty tapahtuma tapahtuu. Tapahtuma voi olla esimerkiksi hiiren liikutus, näppäimistön näppäimen painallus tai verkkoliikennepaketin saapuminen.
Tietoriippuvaisuus	Tietoriippuvaisuus (data dependency) on tilanne, jossa ohjelmalauseke viittaa tietoon, joka voi muuttua edellisten ohjelmalausekkeiden suoritusjärjestyksestä riippuen.

2 Ohjelman suoritusmallit

Tietokoneohjelma voidaan suorittaa joko synkronisesti tai asynkronisesti. Suoritusmalli vaikuttaa ohjelmalausekkeiden suoritusjärjestykseen, ohjelman suorituskykyyn, luettavuuteen, samanaikaisuuteen ja ymmärrettävyyteen. (Pai 2015; Peticolas 2009; Hushan 2015.)

2.1 Synkroninen suoritus

Yksinkertaisimmillaan ohjelma kirjoitetaan suoritettavaksi yhdessä säikeessä (thread) synkronisesti ja lineaarisesti. Synkronisuudella tarkoitetaan, että ohjelmalausekkeet suoritetaan yksi toisensa jälkeen, niin kuin ne esiintyvät lähdekoodissa. (Hushan 2015.)

```
class SynchronousSingleThreaded {
    public static void main (String[] args) {
        int k1 = Integer.parseInt(args[0]);
        int k2 = Integer.parseInt(args[1]);
        int N = Integer.parseInt(args[2])
        double aReal = Math.cos(Math.PI * 2 * k1 / N);
        double aImag = Math.sin(Math.PI * 2 * k1 / N);
        double bReal = Math.cos(Math.PI * 2 * k2 / N);
        double bImag = Math.sin(Math.PI * 2 * k2 / N);
        double outputReal = (aReal * bReal) - (aImag * bImag);
        double outputImag = (aReal * bImag) + (aImag * bReal);

        String imagSign = outputImag < 0 ? "-i" : "i";
        System.out.println("result: " + outputReal + " " + imagSign + Math.abs(outputImag));
    }
}
```

Kuvio 1. Synkronisesti yhdessä säikeessä suoritettava ohjelma.

Kuviossa 1 esitetään yksinkertaisen synkronisesti yhdessä säikeessä suoritettavaksi suunnitellun Java-ohjelman lähdekoodia. Vain yhtä lauseketta suoritetaan kerrallaan ja lausekkeet suoritetaan yksi toisensa jälkeen edeten järjestyksessä ylhäältä alas (Oracle 2010). Koska ohjelma ei sisällä siirräntää eikä prosessoria kuormittavaa laskentaa, on yksisäikeinen synkroninen suoritus riittävä.

Tällä yksinkertaisella suunnittelulla ohjelmat, jotka käyttävät siirräntää, kuten levyltä lukeminen, hidastuvat kuitenkin huomattavasti (Pai 2015). Siirräntä on monta suuruusluokkaa hitaampaa kuin prosessorin tavallisten käskyjen suorittaminen tai päämuistista lukeminen (Norvig 2014). Kuvio 2 havainnollistaa näitä suoritusaikojen eroja.

käskyn suoritus	1 ns
tason 1 välimuistin lukeminen	0,5 ns
käskyhaaran väärinennustaminen	5 ns
tason 2 välimuistin lukeminen	7 ns
poissuljentalukotus	25 ns
päämuistin lukeminen	100 ns
2 kilotavun lähettäminen 1Gbps verkossa	20 000 ns
yhden megatavun lukeminen päämuistista	250 000 ns
satunnainen levyltä lukeminen	8 000 000 ns
1 megantavun lukeminen levyltä järjestyksessä	20 000 000 ns
verkkoliikennepaketin lähettäminen Yhdysvalloista Eurooppaan ja takaisin	150 000 000 ns

Kuvio 2. Eri tietokoneohjelman suorittamien operaatioiden suoritusaikoja (Norvig 2014).

Kuvio 3 esittää lähdekoodia C#-ohjelmasta, joka käyttää siirräntää lukemalla levyltä. Koska ohjelma suoritetaan synkronisesti yhdessä säikeessä, seuraavan tiedoston lukemista ei edes aloiteta ennen kuin edellinen tiedosto on kokonaan luettu ja sen sisältö lisätty lopputulokseen. Ohjelma suoriutuisi huomattavasti nopeammin, jos tiedostot luettaisiin samanaikaisesti. (Pai 2015.)

```
class SynchronousSingleThreadedIO {
    static void Main(string[] args) {
        var result = "";

        for (var fileNameToRead in args) {
            result += System.IO.File.ReadAllText(fileNameToRead);
        }

        System.Console.Write(result);
    }
}
```

Kuvio 3. Synkroninen yksisäikeinen ohjelma, joka käyttää siirräntää.

Synkronisessa ohjelmoinnissa siirrännän käyttöä voidaan nopeuttaa huomattavasti käyttämällä useita säikeitä. Jokaista siirräntäoperaatiota varten luodaan oma säie, joka on operaation aikana odotustilassa. Operaation valmistuttua käyttöjärjestelmä herättää säikeen ja säie voi ilmoittaa tuloksesta pääsäikeelle. Tällöin pääsäie voi jatkaa suoritusta samaan aikaan kuin siirräntäoperaatiot ovat käynnissä. (Asche 1996.)

Kuvio 4 havainnollistaa C#-ohjelmaa, joka käyttää siirräntää monessa säikeessä samanaikaisesti. Jokaisen tiedoston lukemisoperaatio tapahtuu omassa säikeessään, eikä ohjelma odota edellisen operaation valmistumista ennen seuraavan aloitusta. Ohjelma suoriutuu tehtävästi huomattavasti nopeammin kuin kuvion 3 yksisäikeinen versio. (Asche 1996.)

```
using System;
using System.Threading;
using System.Collections.Generic;
using System.IO;

class MultiThreadedIO {
    private class FileReader {
        private string fileName;
        private Thread thread;
        private string result;

        public FileReader(Thread thread, string fileName) {
            this.thread = thread;
            this.fileName = fileName;
        }

        public string GetResult() {
            thread.Join();
            return result;
        }

        public void ReadFile() {
            result = File.ReadAllText(fileName);
        }
    }

    static void Main(string[] args) {
        List<FileReader> fileReaders = new List<FileReader>();
        var result = "";

        for (var fileNameToRead in args) {
            var thread = new Thread(new ThreadStart(fileReader.ReadFile));
            var fileReader = new FileReader(thread, fileNameToRead);
            fileReaders.Add(fileReader);
            thread.Start();
        }

        for (var fileReader in fileReaders) {
            result += fileReader.GetResult();
        }

        System.Console.Write(result);
    }
}
```

Kuvio 4. Synkroninen monisäikeinen ohjelma, joka käyttää siirräntää.

Huomattavaa kuvion 4 ohjelmassa on sen lähdekoodin määrä ja monimutkaisuus verrattuna kuvion 3 ohjelmaan.

2.2 Asynkroninen suoritus

Asynkronisessa ohjelmoinnissa ohjelmalausekkeiden suoritus ei asynkronisia operaatioita käyttäessä mene siinä järjestyksessä kuin ne esiintyvät lähdekoodissa (Peticolas 2009). Kuvion 5 JavaScript-ohjelma havainnollistaa asynkronisen suorituksen järjestystä.

```
console.log("first");
$.get("/profile.php", function(result) {
    console.log("second");
});
console.log("third");
```

Kuvio 5. Asynkronisen suorituksen harhaanjohtava suoritusjärjestys.

Kuvion 5 ohjelmassa esiintyvä lausekkeiden järjestys antaa ymmärtää, että ohjelma tulostaa tekstit järjestyksessä *first*, *second*, *third*. Kuitenkin *$.get*-funktiokutsu on asynkroninen ja sille argumenttina annettua takaisinkutsufunktiota ei kutsuta kuin vasta myöhemmin, jolloin *third*-teksti on jo tulostettu (Vollmer 2011). Ohjelma suorittaa tekstientulostusta ja siirräntää samanaikaisesti, ja tulostusjärjestys on *first*, *third*, *second*.

Asynkroninen ohjelmointi mahdollistaa sen, että yksisäikeinen ohjelma voi suorittaa useita siirräntäoperaatioita samanaikaisesti. Vertailu kuvioiden 3 ja 4 välillä osoittaa, miksi on haluttavaa, että ohjelmakoodi on suunniteltu ajettavaksi yhdessä säikeessä. Monen siirräntäoperaation samanaikaisen suorituksen mahdollisuus poistaa paljon suorituskyky- ja käytettävyysongelmia, joista synkroninen yksisäikeinen ohjelma kärsii. Kuvio 2 osoitti miksi siirräntäoperaatioiden samanaikaistaminen on suorituskyvylle tärkeää. (Hushan 2015.)

Koska laskentaoperaatiot vaativat prosessoriaikaa eikä niitä tällöin voi laittaa taustalle odottamaan, ei asynkroninen ohjelmointi ei mahdollista prosessorilla suoritettavien laskentaoperaatioiden samanaikaistamista, vaan tähän tulee käyttää muita tekniikoita, kuten monisäikeistämistä (Peticolas 2009).

Koska asynkronisessa ohjelmoinnissa asynkroniset funktiokutsut eivät pysäytä ohjelmakoodin suoritusta, usean asynkronisen funktiokutsun tekeminen samaan aikaan aiheuttaa niiden takana olevien operaatioiden suorittamisen samanaikaisesti (Peticolas 2009). Tällöin ohjelmassa aloitettujen samanaikaisten operaatioiden valmistumisen

reagointiin tarvitsee käyttää synkronointia (Caldwell 2013). Kuvio 6 havainnollistaa useiden samanaikaisten asynkronisten funktiokutsujen valmistumiseen reagointia JavaScript-ohjelmakoodilla.

```
var operationsDone = 0;
var operationResults = [];

$.get("/profile.php", function(profileResult) {
    operationResults[0] = profileResult;
    checkDone();
});

$.get("/comments.php", function(commentResult) {
    operationResults[1] = commentResult;
    checkDone();
});

$.get("/picture.php", function(pictureResult) {
    operationResults[2] = pictureResult;
    checkDone();
});

function checkDone() {
    operationsDone++;
    if (operationsDone === 3) {
        console.log("all done", operationResults);
    }
}
```

Kuvio 6. Monen asynkronisen operaation synkronointi takaisinkutsufunktioilla.

Kuviossa 6 jokaisen operaation valmistumisen jälkeen tarkistetaan, ovatko muut operaatiot jo valmistuneet, sillä operaatioiden valmistumisjärjestys ei ole ennalta määrätty ja vaihtelee ohjelman suorituskertojen välillä.

Asynkronisten operaatioiden suorittaminen järjestyksessä on takaisinkutsufunktioita käyttäessä monimutkaista, sillä synkronisaatio pitää tehdä jokaisen operaation jälkeen, minkä suoritusjärjestystä halutaan hallita (Creamer 2013). Operaatioiden suorittaminen järjestyksessä on välttämätöntä silloin kun ne ovat tietoriippuvaisia toisistaan (Hennessy ym. 2003). Kuvio 7 esittää JavaScript-ohjelman, joka käyttää takaisinkutsufunktioita kolmen peräkkäisen asynkronisen operaation synkronisoimiseen, jotta ne suoritettaisiin järjestyksessä.

```
console.log("starting to fetch profile");
$.get("/profile.php", function(profileResult) {
    console.log("got profile", profileResult);
    console.log("starting to fetch comments");

    $.get("/comments.php", function(commentResult) {
        console.log("got comments", commentResult);
        console.log("starting to fetch picture");

        $.get("/picture.php", function(pictureResult) {
            console.log("got picture", pictureResult);
        });
    });
});
```

Kuvio 7. Asynkronisten operaatioiden suorittaminen järjestyksessä takaisinkutsufunktioita käyttämällä.

Kuviosta 7 näkyy kuinka operaatioiden peräkkäinen järjestyksessä suorittaminen asynkronisessa ohjelmoinnissa takaisinkutsufunktioita käyttäessä voi helposti tehdä ohjelmakoodista vaikeasti luettavaa.

3 Asynkronisen ohjelmoinnin tekniikat

Edellisessä luvussa on käytetty asynkroniseen ohjelmointiin takaisinkutsufunktio-tekniikkaa. Asynkronisia ohjelmointitekniikoita on takaisinkutsufunktioiden lisäksi myös lupaukset, Async/Await ja tapahtumakuuntelijat (Archibald 2013; Hunter II 2015).

3.1 Lupaukset

Takaisinkutsufunktioissa on huomattavaa kuinka paljon niiden toiminta eroaa tavallisesta synkronisesta funktiokutsusta. Tavallinen synkroninen funktiokutsu palauttaa arvon tai heittää poikkeuksen. Takaisinkutsufunktioita käyttävät asynkroniset funktiota eivät palauta hyödyllistä arvoa, vaan kutsuvat myöhemmin sivuvaikutuksena argumenttina annettua takaisinkutsufunktiota. (Motta 2015.)

Lupaukset ovat olioita, jotka mahdollistavat asynkronisten arvojen tai operaatioiden ilmaisemisen ensiluokkaisina arvoina (Motta 2015). Näin lupaukset mahdollistavat synkronisesta maailmasta tutut arvojen ja poikkeusten yhdistelyn (composition), joka ei ole mahdollista takaisinkutsufunktioita käyttäessä (Denicola 2012).

Lupaus voi olla kolmessa eri tilassa: *toteutettu* (fulfilled), *hylätty* (rejected) tai *avoinna* (pending). Silloin kun lupaus ei ole enää *avoinna*, eli se on joko *toteutettu* tai *hylätty*, voidaan lupauksesta sanoa, että se on *päätetty* (settled). Uusi lupausolio on *avoinna*-tilassa, ja se voi siirtyä *avoinna*-tilasta joko *toteutettu*- tai *hylätty*-tilaan. Toteutuneella lupauksella on aina arvo, jolla se on toteutettu. Toteutumisarvo heijastaa synkronisen ohjelmoinnin funktioiden tavallista paluuarvoa. Hylätyllä lupauksella on hylkäyssyy, jonka vuoksi lupaus on hylätty. Hylkäyssyy on poikkeusolio, joka heijastaa synkronisen ohjelmoinnin heitettyä poikkeusta. (Archibald 2013; Denicola 2012.)

Lupaukseen voidaan liittää tapahtumankäsittelijä käyttämällä sen *then*-metodia, jota kutsutaan, kun lupaus on toteutunut. Metodi palauttaa uuden lupauksen, jonka tila määräytyy argumenttina annetun tapahtumankäsittelijäfunktion käyttäytymisen perusteella. Tapahtumankäsittelijä määrää palautetun lupauksen tilan palauttamalla joko arvon tai uuden lupauksen tai heittämällä poikkeuksen. Jos tapahtumalle ei ole käsittelijää, saa palautettu lupaus saman päätöksen kuin alkuperäinen lupaus. (Archibald 2013; Denicola 2012.)

Kuviossa 8 *promise1*-lupaus tulee toteutumaan *"/picture.php"*-sivun HTTP-vastauksella. Sille annettu tapahtumankäsittelijä palauttaa arvon *3*, joten *promise2*-lupaus tulee toteutumaan arvolla *3*.

```
var promise1 = $.getAsync("/picture.php");
var promise2 = promise1.then(function() {
    return 3;
});
```

Kuvio 8. Lupauksen toteutumisen tapahtumankäsittelijä aiheuttaa toteutumisen suoralla arvolla.

Kuviossa 9 *promise1*-lupaus tulee toteutumaan *"/picture.php"*-sivun HTTP vastauksella. Sille annettu tapahtumankäsittelijä yrittää kutsua HTTP-vastausolion (*result*) *toNumber*-metodia, jota ei ole olemassa, joten tapahtumankäsittelijä tulee heittämään poikkeuksen, jolloin *promise2*-lupaus tulee hylätyksi *TypeError*-tyyppipoikkeuksella.

```
var promise1 = $.getAsync("/picture.php");
var promise2 = promise1.then(function(result) {
    return result.toNumber();
});
```

Kuvio 9. Lupauksen *promise1* toteutumisen tapahtumankäsittelijä aiheuttaa promise2-lupauksen hylkäyksen.

Kuviossa 10 *promise1*-lupaus tulee toteutumaan *"/picture.php"*-sivun HTTP-vastauksella. Sille annettu tapahtumankäsittelijä palauttaa lupauksen, joten *promise2*-lupauksen päätös tulee olemaan sama kuin tapahtumankäsittelijän palauttaman lupauksen päätös.

```
var promise1 = $.getAsync("/picture.php");
var promise2 = promise1.then(function(result) {
    return $.getAsync("/profile.php");
});
```

Kuvio 10. Lupauksen toteutumisen tapahtumankäsittelijä ratkaisee lupauksen uudella lupauksella.

Kuvio 11 havainnollistaa kuinka lupauksista voidaan muodostaa niiden yhdisteltävyyttä hyväksikäyttäen järjestyksessä suoritettavien asynkronisten operaatioiden ketju.

```
$.getAsync("/picture.php").then(function(pictureResult) {
    return $.getAsync("/profile.php");
}).then(function(profileResult) {
    return $.getAsync("/comments.php");
}).then(function(commentResult) {
    return $.getAsync("/posts.php");
}).then(function(postsResults) {

});
```

Kuvio 11. Järjestyksessä suoritettavien asynkronisten operaatioiden ketju.

Lupausten *then*-metodilla voidaan rekisteröidä vain toteutumista koskevia tapahtumankäsittelijöitä. Hylkäyksen tapahtumankäsittelijä rekisteröidään lupausten *catch*-metodilla (Archibald 2013; Denicola 2012.) Jos kuviossa 11 ensimmäinen operaatio epäonnistuisi, ei mitään tapahtumankäsittelijää kutsuttaisi, sillä hylkäystapahtumalle ei ole yksikään lupaus rekisteröinyt tapahtumankäsittelijää. Kuviossa 12 annetaan hylkäyksen tapahtumankäsittelijä, jonne koodin suoritus siirtyy suoraan kun missä tahansa operaatiossa tai totetumisen tapahtumankäsittelijässä tapahtuu virhe.

```
$.getAsync("/picture.php").then(function(pictureResult) {
    return $.getAsync("/profile.php");
}).then(function(profileResult) {
    return $.getAsync("/comments.php");
}).then(function(commentResult) {
    return $.getAsync("/posts.php");
}).then(function(postsResults) {

}).catch(function(e) {
    console.error("Error: " + e.message);
});
```

Kuvio 12. Hylkäyksen (asynkroninen poikkeus) käsittely lupausketjussa.

Kuviosta 12 ilmenee kuinka lupaukset mahdollistavat olennaisesti samankaltaisen ohjelmointi-ilmaisutehon kuin synkroninen ohjelmointi menettämättä asynkronisen ohjelmoinnin tuomia hyötyjä.

Useat ohjelmointiympäristöt tai -kielet sisältävät toteutuksen lupauksista. JavaScriptissä lupausrajapinnan toteuttaa Promise-luokka (Archibald 2013), Javassa Future-luokka (Oracle 2014) ja C#:ssä Task (Microsoft 2015a). JavaScript-ympäristöissä lupauksista on saatavilla myös useita kolmannen osapuolen toteutuksia, kuten Q, when, WinJS ja RSVP.js (Archibald 2013; Motta 2015).

3.2 Async/Await

Async/Await viittaa Microsoftin C#-kielen versiossa 5.0 ilmestyneeseen ominaisuuteen, joka mahdollistaa asynkronisen ohjelmoinnin lähes täysin synkronisella syntaksilla (Microsoft 2015b). Ominaisuus perustuu jatkettaviin funktioihin (resumable function), joissa *yield*-lauseke korvataan *await*-lausekeella (Nishanov ym. 2014).

Ominaisuus ei ole suoraan saatavilla tämänhetkisissä JavaScript-ympäristöissä, vaan sen käyttöön tarvitsee esikääntäjän. Async/Await toteutetaan JavaScriptin ES7-versiossa. (Archibald 2014.)

Kuvio 13 ilmaisee kuvion 3 ohjelman käyttäen C#-kielen Async/Await-ominaisuutta. Ohjelma on myös yhtä hidas kuin kuvion 3 ohjelma, sillä se odottaa aina yhden tiedoston lukemisen valmistumista ennen seuraavan aloittamista.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading.Tasks;

class AsynchronousIO {
    private async Task<string> ReadFiles(string[] args) {
        var result = "";

        for (var fileNameToRead in args) {
            result += await Task.Run(() => File.ReadAllText(fileName));
        }

        return result;
    }

    public static void Main(string[] args) {
        Task<string> resultTask = ReadFiles(args);
        resultTask.Wait();
        Console.Write(resultTask.Result);
    }
}
```

Kuvio 13. Tiedostojen lukeminen käyttäen Async/Await-tekniikkaa C#-kielessä.

Kuvio 14 siirtää tehtävien (Task) odotuksen erilliseen silmukkaan, jotta niiden suoritus tapahtuisi samanaikaisesti.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading.Tasks;

class AsynchronousIO {
    private async Task<string> ReadFiles(string[] args) {
        var tasks = new List<Task<String>>();
        var result = "";

        for (var fileNameToRead in args) {
            tasks.Add(Task.Run(() => File.ReadAllText(fileName)));
        }

        for (var task in tasks) {
            result += await task;
        }

        return result;
    }

    public static void Main(string[] args) {
        Task<string> resultTask = ReadFiles(args);
        resultTask.Wait();
        Console.Write(resultTask.Result);
    }
}
```

Kuvio 14. Tiedostojen lukeminen samanaikaisesti käyttäen Async/Await-tekniikkaa C#-kielessä.

3.3 Tapahtumankuuntelijat

Tapahtumankuuntelija on funktio, jota kutsutaan joka kerta kun sen kuuntelema tapahtumatyyppi tapahtuu. Lupauksia voidaan verrata tapahtumakuuntelijoihin, jotka ovat rajoitettu vain yhteen tapahtumaan (Archibald 2013). Kuviossa 15 on esimerkki JavaScript-tapahtumankuuntelijasta, jota kutsutaan joka kerta kun käyttäjä liikuttaa hiirtä verkkosivulla.

```
document.addEventListener("mousemove", function(event) {
    document.body.textContent = "Your mouse is currently at x: " + event.x , +
                                                    "y: " + event.y;
});
```

Kuvio 15. Tapahtumankuuntelija JavaScriptissä.

4 Asynkronisten tekniikoiden vertailu

Työn tarkoitus on verrata asynkronisia tekniikoita Node.js-ympäristössä ja selvittää, mitä vahvuuksia ja heikkouksia niillä on ja mihin samanaikaisuusvaatimuksiltaan tai operaatioluonteeltaan erilaisiin tilanteisiin kukin tekniikka soveltuu parhaiten.

Node.js on suosittu palvelinpuolen sovellusalusta, joka on rakennettu V8-JavaScript moottorin päälle. V8 on alun perin Google Chromea varten kehitetty JavaScript-moottori. V8 kääntää JavaScript-lähdekoodin suoraan prosessorikeskeiseksi natiivikoodiksi tulkitsemisen (interpreter) sijaan. Node.js:n tarkoitus on mahdollistaa skaalautuvien ja nopeiden palvelinsovellusten helppo kehittäminen. (Cantelon ym. 2013.)

Node.js:ssä sovellukset kehitetään JavaScript-ohjelmointikielellä. JavaScript-ohjelmointikieli on alun perin tarkoitettu verkkosivujen kevyeen skriptaukseen, mutta on alkanut muuttumaan vuodesta 2005 lähtien ohjelmointikieleksi, jolla kirjoitetaan täysivaltaisia ohjelmistoja. (Cantelon ym. 2013.)

JavaScriptin vahvuuksia palvelinsovellusten kehityksessä ovat (Cantelon ym. 2013):

- Web-sovelluksessa asiakaspuoli ja palvelinpuoli voivat käyttää samaa lähdekoodia.
- JavaScript tukee suosittua JSON-dataformaattia natiivisti.
- JavaScript on kääntämisen kohteeksi soveltuva kieli ja useimmille kielille löytyy kääntäjä, joka kääntää kielen JavaScriptiksi.

JavaScript toimii Node.js:ssä aivan kuten se toimii verkkoselaimissa. Node.js on pohjimmiltaan tapahtumakeskeinen ja asynkroninen, eikä se sisällä synkronisen ohjelmoinnin rajapintoja muuten kuin erityistapauksissa. Oletuksena alusta sisältää rajapinnat ajastimille, konsolisiirrännälle, tiedostojärjestelmälle, HTTP:lle, TLS:lle, HTTPS:lle, UDP:lle ja TCP:lle. Ainoastaan konsolirajapinta on oletuksena synkroninen. Tiedostojärjestelmärajapinnoista on olemassa synkroniset versiot ja muista rajapinnoista on vain asynkroniset rajapinnat. (Cantelon ym. 2013.)

Node.js:n mukana toimitetaan NPM-paketinhallintaohjelmisto, jolla hallitaan ja asennetaan Node.js-kirjastoja ja -työkaluja (Dierx 2015).

Koska Node.js-ympäristössä kaikki rajapinnat ovat pääosin asynkronisia ja asynkronisessa ohjelmoinnissa voidaan käyttää montaa eri tekniikkaa, tulee Node.js-alustalle kehittävien ohjelmistokehittäjien ja –arkkitehtien tuntea tekniikoiden heikkoudet, vahvuudet ja niiden sopivuus erilaisten asynkronisen ohjelmoinnin ongelmien ratkaisuiksi.

4.1 Tutkimusmenetelmä

Tämän työn empiirinen osa muodostuu keskeisimpien asynkronisen ohjelmoinnin tekniikoiden vertailusta Node.js-ympäristössä. Keskeisimpinä tekniikkoina pidetään takaisinkutsufunktioita ja lupauksia, sillä kaikki Node.js-rajapinnat toteuttavat ensimmäisen ja JavaScript-ympäristö tukee natiivisti jälkimmäistä.

Käytännössä Node.js-ympäristössä takaisinkutsufunktioita käytetään joko suoraan, tai async-kirjaston apufunktioiden avulla. Lupauksia taas käytetään joko bluebird- tai Q-kirjaston toteutusten kautta. Arviot perustuvat NPM-latausmääriin (NPM 2015a; NPM 2015b; NPM 2015c).

Vertailtaviksi tekniikoiksi valitaan takaisinkutsufunktiot suoraan käytettynä, takaisinkutsufunktiot async-kirjaston avulla käytettynä sekä lupaukset bluebird-kirjaston toteutuksena. Suoraan käytetyt takaisinkutsufunktiot ovat käytettävissä Node.js-ympäristössä ilman erillisiä kirjastoja ja se on tällöin asynkronisten rajapintojen oletuskäytäntö. Async-kirjaston apufunktiot ovat taas NPM-latausmäärien perusteella suosituin vaihtoehtoinen asynkronisten rajapintojen vaihtoehtoinen kulutuskeino. Bluebird on NPM-latausmäärien perusteella nopeiten kasvava lupauksia käyttävän tekniikan toteutus. (NPM 2015a; NPM 2015b; NPM 2015c)

Tavoitteena on tuoda esille valittujen tekniikkojen heikkouksia ja vahvuuksia käytettävyyden, suorituskyvyn ja käyttäjäkoodin kompleksisuuden kannalta sekä tunnistaa jokaiseen ongelman ratkaisemiseksi parhaiten soveltuva tekniikka.

Jokaista tekniikkaa arvioidaan soveltamalla niitä kolmeen erilaiseen asynkronisen ohjelmoinnin ongelman ratkaisuun.

Tekniikan käytettävyyttä mitataan aiheuttamalla virhetilanne ja tarkastelemalla tulostettua pinojäljitystä. Pinojäljityksestä katsotaan sen ilmoittama rivinumero ja vertaamalla rivinumeron etäisyyttä virheen aiheuttamaan todelliseen koodiriviin.

Taulukko 1. Käytettävyyden pisteytystasot.

Lopputulema	Pisteytys
Ei pinojäljitystä	$S_{us} = 0$
Ei viittausta riviin	$S_{us} = 5$
Viittaus riviin ei ole viimeisin pinojäljityksen pinokehys	$S_{us} = 10$
Viittaus riviin, mutta pinojäljitys ei sisällä kaikkia tapahtumia	$S_{us} = 15$
Viittaus riviin ja pinojäljitys sisältää kaikki tapahtumat	$S_{us} = 20$

Tekniikan suorituskyky mitataan ajamalla tekniikalla toteutettu ratkaisu ongelmasta riippuvan määrän mukaan. Pisteytys:

$$S_p = \frac{5000}{t_{ms}} + \frac{100}{MMax_{mb}}$$

Jossa t_{ms} on suoritukseen kulunut aika millisekunneissa ja $MMax_{mb}$ on suurin suoritukseen aikana käytetty muistimäärä megatavuissa.

Käyttäjäkoodin kompleksisuutta mitataan laskemalla tekniikalla toteutetun ratkaisun lähdekoodin rivimäärä. Pisteytys:

$$S_c = \frac{1000}{L}$$

Jossa L on rivimäärä.

Kokonaispisteytys:

$$S_{total} = S_{us} + S_p + S_c$$

Pisteytys on valittu niin, että eri osa-alueiden tulokset ovat verrattavissa toisiinsa kullakin mittarilla saadun pienimmän ja suurimman arvon puitteissa. Eniten kokonaistulokseen vaikuttaa rivimäärien muutokset ja vähiten suorituskyky. Painotus kokonaispisteytyksessä on tällöin kallistunut eniten luettavuuden puolelle. Samojen mittareiden tulosten keskenään vertailua varten ei tarvitse tulosarvoja muuntaa pistemääriksi. Eri mittareiden vertailua varten pistelaskukaavoja voidaan säätää omien tarpeiden mukaan, jos painotusodotukset eroavat edellä mainitusta.

4.2 Ongelmien kuvaukset

Ongelmien ratkaisut suoritetaan tässä työssä 64-bittisessä Ubuntu 14.04 -ympäristössä käyttäen Node.js-versiota 5.1.0.

Ratkaisun oikeellisuuden testausta varten tulee ratkaisumoduulin viedä ulostulevassa (export) rajapinnassaan *run*-niminen funktio, joka ottaa argumenttina takaisinkutsufunktion, jota ratkaisumoduulin tulee kutsua kahdella argumentilla riippuen moduulin tuloksesta:

1. Jos moduulin tuloksena tapahtui poikkeus, tulee takaisinkutsufunktiota kutsua poikkeusolio ensimmäisenä argumenttina.

2. Jos moduuli suoriutui normaalisti, tulee takaisinfunktiota kutsua *null* ensimmäisenä argumenttina ja tulos toisena argumenttina.

```
exports.run = function(done) {
    done(null, "hello world");

    done(new Error("exception"));
};
```

Kuvio 16. Esimerkki toteutettavasta rajapinnasta ratkaisun oikeellisuuden testaamista varten.

Suorituskykymittausta varten ratkaisumoduulin tulee viedä ulostulevassa rajapinnassaan *testPerformance*-niminen funktio, joka ottaa argumenttina funktion, jota ratkaisumoduulin tulee kutsua kun ratkaisun tekemä operaatio on suoritettu loppuun. Kuviossa 17 esitetään esimerkki rajapinnan toteutuksesta.

```
exports.testPerformance = function (done) {
    done();
};
```

Kuvio 17. Esimerkki suorituskykymittausta varten toteutettavasta rajapinnasta.

Kuvio 18 havainnollistaa suorituskyvyn mittausohjelmaa.

```
const solution = require("./solution");

var currentIteration = 0;
var totalIterations = 1000;

var startTime = Date.now();
var memMax = -Infinity;

solution.testPerformance(function perfLoop() {
    memMax = Math.max(memMax, process.memoryUsage().rss / 1024 / 1024);
    currentIteration++;
    if (currentIteration < totalIterations) {
        solution.testPerformance(perfLoop);
    } else {
        const timeTaken = Date.now() - startTime;
        console.log("time taken: ", timeTaken, "mx",
                    "max memory", memMax.toFixed(2), "MB");
    }
});
```

Kuvio 18. Pohja suorituskyvyn mittaamiseksi.

Kaikki ongelmat suoritetaan *tämänhetkinen työskentelykansio* (current working directory, cwd) asetettuna kansioon, jonka rakenne ja sisältö ovat kuvion 19 mukaiset.

```
petka@petka-VirtualBox ~/oppari
$ ls -R test-folder/
test-folder/:
aaa.txt  abc.txt  bbc.txt  bca.txt  bur.txt  ffa.txt  ffu.txt  kek.txt  nnn.txt  rnd.txt
```

Kuvio 19. Ratkaisujen suorituksessa käytettävän tämänhetkisen työskentelykansion rakenne ja sisältö

4.2.1 Ongelma 1 – tiedostojen ryhmittely

Ohjelman tulee lukea kaikki tämänhetkisessä työskentelykansiossa olevien tiedostojen nimet ja ryhmitellä ne tiedostonimen ensimmäisen kirjaimen mukaan. Jokaisesta tällaisestä ryhmästä on laskettava ryhmään kuuluvien tiedostojen määrä ja niiden kokonaistiedostokoko tavuissa. Ryhmät tulee järjestää kuhunkin ryhmään kuuluvan tiedostomäärän mukaan suurimmasta pienempään. Tulostusta varten tuloksen pitää olla merkkijono (string), jossa jokainen ryhmä on muotoiltu yhdelle riville:

```
"Group: " M " Filecount: " S " Total size: " S "[new line]"
```

Jossa `M` on ryhmään kuuluvien tiedostojen nimien ensimmäinen kirjain, `S` ryhmään kuuluvien tiedostojen määrä ja `S` on ryhmään kuuluvien tiedostojen kokonaistiedostokoko tavuissa.

Ohjelman tulee suorittaa kaikki siirräntä samanaikaisesti.

Kuviossa 20 esitetään lähdekoodi ohjelmalle, joka tarkistaa ratkaisun oikeellisuuden.

```
const Path = require("path");
const solutionFilePath = process.argv[2];
const solution = require("./" + solutionFilePath);
const assert = require("assert");

process.chdir(Path.join(process.cwd(), "test-folder"));

process.on("uncaughtException", fail);
process.on("unhandledRejection", fail);

solution.run(function(err, result) {
    try {
        if (err) throw err;

        const expected = "Group: b Filecount: 3 Total size: 177\n" +
                         "Group: a Filecount: 2 Total size: 80\n" +
                         "Group: f Filecount: 2 Total size: 220\n" +
                         "Group: k Filecount: 1 Total size: 65\n" +
                         "Group: n Filecount: 1 Total size: 52\n" +
                         "Group: r Filecount: 1 Total size: 102";

        assert.strictEqual(expected, result);
        console.log("test success, result output: ");
        console.log(result);
    } catch (e) {
        fail(e);
    }
});

function fail(error) {
    console.error("Test fail: " + error.message);
}
```

Kuvio 20. Ongelman 1 ratkaisun oikeellisuuden tarkistava ohjelma.

Kuvio 21 esittää lähdekoodin malliratkaisusta ongelmaan 1. Toteutus käyttää synkronisia rajapintoja selkeyden vuoksi eikä voi siis täyttää vaatimusta tiedostolukemisen samanaikaisuudesta.

```
const fs = require("fs");
const Path = require("path");

function run(done) {
    var currentGroup = {fileNames: []};

    var result = fs.readdirSync(process.cwd()).sort()
        .map(function(fileName) {
            if (!currentGroup.firstCharacter) {
                currentGroup.firstCharacter = fileName[0];
                currentGroup.fileNames.push(fileName);
            } else if (fileName[0] === currentGroup.firstCharacter) {
                currentGroup.fileNames.push(fileName);
            } else {
                currentGroup = {
                    fileNames: [fileName],
                    firstCharacter: fileName[0]
                };
            }
            return currentGroup;
        }, {fileNames: []})
        .filter(function(group, index, groups) {
            return groups.indexOf(group, index + 1) === -1;
        })
        .map(function(group) {
            group.fileCount = group.fileNames.length;
            group.totalSize = group.fileNames.reduce(function(totalSize, fileName) {
                var size = fs.statSync(Path.join(process.cwd(), fileName)).size;
                return totalSize + size;
            }, 0);

            return group;
        })
        .sort(function(aGroup, bGroup) {
            return bGroup.fileCount - aGroup.fileCount;
        })
        .map(function(group) {
            return "Group: " + group.firstCharacter + " " +
                   "Filecount: " + group.fileCount + " " +
                   "Total size: " + group.totalSize;
        })
        .join("\n");

    done(null, result);
}

exports.testPerformance = function(done) {
    run(function() {
        done();
    });
};

exports.run = run;
```

Kuvio 21. Ongelman 1 malliratkaisu, joka on toteutettu synkronisia rajapintoja käyttäen.

```
petka@petka-VirtualBox ~/oppari
$ node problem1test.js problem-1-model-solution.js
test success, result output:
Group: b Filecount: 3 Total size: 177
Group: a Filecount: 2 Total size: 80
Group: f Filecount: 2 Total size: 220
Group: k Filecount: 1 Total size: 65
Group: n Filecount: 1 Total size: 52
Group: r Filecount: 1 Total size: 102
```

Kuvio 22. Ongelman 1 malliratkaisun oikeellisuustarkistajaohjelman suorituksen antama tuloste.

4.2.2 Ongelma 2 – tiedostojen yhteenliittäminen

Ohjelman tulee lukea kaikki tämänhetkisessä työskentelykansiossa olevien tiedostojen sisällöt ja yhdistää niiden sisällöt toisiinsa. Tiedostot tulee lukea aakkosjärjestyksessä ja kaksi tiedostoa kerrallaan. Jos tiedostoja on pariton määrä, voidaan viimeinen tiedosto lukea ilman, että muita tiedostojenlukuoperaatioita on samanaikaisesti käynnissä.

Kuviossa 23 esitetään lähdekoodi ohjelmalle, joka tarkistaa ratkaisun oikeellisuuden.

```
const Path = require("path");
const solutionFilePath = process.argv[2];
const solution = require("./" + solutionFilePath);
const assert = require("assert");

process.chdir(Path.join(process.cwd(), "test-folder"));

process.on("uncaughtException", fail);
process.on("unhandledRejection", fail);

solution.run(function(err, result) {
    try {
        if (err) throw err;

        const expected = "beayvsmjzliaofdmhbuifmoyoucxfmfhbadpwt" +
                         "dvseokziyuictlezyescfrzqhibatbdspppogi" +
                         "yvaezlnfevwwvkznoctbsxhxtgthiiwvtijqyu" +
                         "rlelqskudzthijtxvmxuadqtdqftppqhogxicj" +
                         "tcnlwayxxvwxiyaejfvvemicdoatkmuucphrtk" +
                         "wocxdsptbhuxfesvbpbjmblxrbyanjjklhypjj" +
                         "ypnrkoteeflyqqjaegckrghjdilbzyxkkbtyby" +
                         "fnpngoeahpfgqypstedtvwdsadcxkbxogipodg" +
                         "surischyomlfdaapooeugzeygdcslywlinlusa" +
                         "ddkxxkcjegpssxogisrifqxcpbslsphngnakos" +
                         "uhmzrkocaaiaskrgpwvmorckpeldcrplphmjnc" +
                         "wrnwmkapmexxqtjdjjrzrxnfrpnzwkmrcwqeay" +
                         "dxrzhvntohrxuxhefnybsjkouiwsviuzrjrltf" +
                         "rtkyokpjrkzfubuhjbwjzjrchmticprxosujzt" +
                         "hgvdwbdutzethamslonrninbydadcfnfbtyljx" +
                         "yukydgppzbjtvgckbshajvluuunsxrkjqljnlm" +
                         "bfbcxstgeyhynapgiifpvsmmppevsbwohygujr" +
                         "ffdlnmivybtpjtrqhgogztjdcdkpqemormhwrd" +
                         "cfkblglupjam";

        assert.strictEqual(expected, result);
        console.log("test success, result output: ");
        console.log(result);
    } catch (e) {
        fail(e);
    }
});

function fail(error) {
    console.error("Test fail: " + error.message);
}
```

Kuvio 23. Ongelman 2 ratkaisun oikeellisuuden tarkistava ohjelma.

Kuvio 24 esittää lähdekoodin malliratkaisusta ongelmaan 2. Toteutus käyttää synkronisia rajapintoja selkeyden vuoksi eikä siis voi täyttää vaatimusta tiedostolukemisen samanaikaisuudesta.

```
const fs = require("fs");
const Path = require("path");

function run(done) {
    var result = fs.readdirSync(process.cwd()).sort()
            .reduce(function(output, fileName) {
                const path = Path.join(process.cwd(), fileName);
                return output + fs.readFileSync(path).toString();
            }, "");
    done(null, result);
}

exports.testPerformance = function(done) {
    run(function() {
        done();
    });
};

exports.run = run;
```

Kuvio 24. Ongelman 2 malliratkaisu, joka on toteutettu synkronisia rajapintoja käyttäen.

```
petka@petka-VirtualBox ~/oppari
$ node problem2test.js problem-2-model-solution.js
test success, result output:
beayvsmjzliaofdmhbuifmoyoucxfmfhbadpwtdvseokziyuictlezyescfrzqhibatbdspppogiyvae
zlnfevwwvkznoctbsxhxtgthiiwvtijqyurlelqskudzthijtxvmxuadqtdqftppqhogxicjtcnlwayx
xvwxiyaejfvvemicdoatkmuucphrtkwocxdsptbhuxfesvbpbjmblxrbyanjjklhypjjypnrkoteefly
qqjaegckrghjdilbzyxkkbtybyfnpngoeahpfgqypstedtvwdsadcxkbxogipodgsurischyomlfdaap
ooeugzeygdcslywlinlusaddkxxkcjegpssxogisrifqxcpbslsphngnakosuhmzrkocaaiaskrgpwvm
orckpeldcrplphmjncwrnwmkapmexxqtjdjjrzrxnfrpnzwkmrcwqeaydxrzhvntohrxuxhefnybsjko
uiwsviuzrjrltfrtkyokpjrkzfubuhjbwjzjrchmticprxosujzthgvdwbdutzethamslonrninbydad
cfnfbtyljxyukydgppzbjtvgckbshajvluuunsxrkjqljnlmbfbcxstgeyhynapgiifpvsmmppevsbwo
hygujrffdlnmivybtpjtrqhgogztjdcdkpqemormhwrdcfkblglupjam
```

Kuvio 25. Ongelman 2 malliratkaisun oikeellisuustarkistajaohjelman suorituksen antama tuloste

4.2.3 Ongelma 3 – yksittäisen tiedoston käsittely

Ohjelman tulee valita tämänhetkisestä työskentelykansiosta satunnainen tiedosto, joka ei kuitenkaan ole ohjelman lähdekooditiedosto. Ohjelman tulee lukea tämän tiedoston *viimeksi muokattu* –päiväys ja kirjoittaa se tiedoston päätyyn. Tuloksena tulee palauttaa olio, jolla ovat attribuutit *modifiedDate*, joka on valitun tiedoston viimeksi muokattu -päiväys sekä *fileName*, joka on valitun tiedoston tiedostonimi.

Kuviossa 26 esitetään lähdekoodi ohjelmalle, joka tarkistaa ratkaisun oikeellisuuden.

```
const Path = require("path");
const solutionFilePath = process.argv[2];
const solution = require("./" + solutionFilePath);
const cp = require("child_process");
const assert = require("assert");
const fs = require("fs");

process.chdir(Path.join(process.cwd(), "test-folder"));

process.on("uncaughtException", fail);
process.on("unhandledRejection", fail);

solution.run(function(err, result) {
    try {
        if (err) throw err;

        const filePath = Path.join(process.cwd(), result.fileName);
        const fileContents = fs.readFileSync(filePath).toString();
        assert.strictEqual(result.modifiedDate,
            fileContents.slice(fileContents.length - result.modifiedDate.length,
                               fileContents.length));

        console.log("test success");
    } catch (e) {
        fail(e);
    } finally {
        // Restore all files in the test-folder to normal state.
        process.chdir(Path.join(process.cwd(), ".."));
        cp.execSync("rm -rf test-folder && " +
                    "mkdir -p test-folder && " +
                    "cp test-folder-backup/*.* test-folder/");
    }
});


function fail(error) {
    console.error("Test fail: " + error.stack);
}
```

Kuvio 26. Ongelman 3 ratkaisun oikeellisuuden tarkistava ohjelma.

Kuvio 27 esittää lähdekoodin malliratkaisun ongelmaan 3.

```
const fs = require("fs");
const Path = require("path");

function run(done) {
    const filePool = fs.readdirSync(process.cwd()).filter(function(fileName) {
        return fileName !== __filename;
    });
    const selectedFileName = filePool[Math.floor(Math.random() * filePool.length)];
    const filePath = Path.join(process.cwd(), selectedFileName);
    const modifiedDate = fs.statSync(selectedFileName).mtime.toString();
    const handle = fs.openSync(filePath, "a+");
    fs.writeSync(handle, modifiedDate);
    fs.closeSync(handle);

    var result = {
        fileName: selectedFileName,
        modifiedDate: modifiedDate
    };

    done(null, result);
}

exports.testPerformance = function(done) {
    run(function() {
        done();
    });
};

exports.run = run;
```

Kuvio 27. Ongelman 3 malliratkaisu, joka on toteutettu synkronisia rajapintoja käyttäen.

```
petka@petka-VirtualBox ~/oppari
$ node problem3test.js problem-3-model-solution.js
test success
```

Kuvio 28. Ongelman 3 malliratkaisun oikeellisuustarkistajaohjelman suorituksen antama tuloste.

4.3 Takaisinkutsufunktiot

Node.js-rajapinnat toteuttavat takaisinkutsusopimuksen oletuksena, joten niiden käyttöön ei tarvitse asentaa erillistä kirjastoa. Ratkaisulähdekoodeista on otettu niiden pituuden takia vain olennaisin osa kuvioihin.

Kuviosta 29 ilmenee kuinka takaisinkutsufunktioita käyttäessä pinojäljitys ei kerro koko tarinaa virheeseen johtaneesta tapahtumaketjusta. Virhetilanteessa vain viimeisimmän tapahtuman pinojäljitys on saatavilla, joskin sen antama pinojäljitys sisältää virheen aiheuttaneen koodiriviviittauksen tarkasti.

```
petka@petka-VirtualBox ~/oppari
$ node problem2test.js problem-2-callback-solution.js
Test fail: ReferenceError: fileReadsInProcess is not defined
    at startReading (/home/petka/oppari/problem-2-callback-solution.js:31:50)
    at Array.forEach (native)
    at /home/petka/oppari/problem-2-callback-solution.js:57:23
    at FSReqWrap.oncomplete (fs.js:82:15)
```

Kuvio 29. Takaisinkutsufunktioratkaisun antama pinojäljitys virhetilanteessa.

Koska takaisinkutsufunktiot eivät automaattisesti tarkista, onko funktiota jo kutsuttu takaisin, tulee ensin luoda saadusta takaisinkutsufunktiosta uusi ainoastaan kerran kutsuttava funktio käyttämällä *onlyOnce*-apufunktiota.

Takaisinkutsufunktioita käyttäessä asynkronisten virheiden käsittely tapahtuu tarkistamalla onko takaisinkutsufunktion saama ensimmäinen argumentti tyhjä. Tarkistus voidaan tehdä käyttämällä *if*-rakeneteella käyttämällä argumenttia rakenteen konditionaali-ilmaisuna.

Synkroniset virheet, eli heitetyt poikkeukset, tulee takaisinkutsufunktioita käyttäessä käsitellä eri mekanismilla kuin asynkroniset virheet. Heitetyt poikkeukset voidaan käsitellä sulkemalla koodiosa, josta poikkeuksia halutaan siepata, *try-catch*-lohkolla. Lohkon vaikutus ei kuitenkaan siirry asynkronisten tapahtumien välillä, joten jokainen takaisinkutsufunktio joutuu määrittelemään uuden lohkon huolimitta siitä, onko funktion määrittely syntaktisesti jonkun muun *try-catch*-lohkon sisällä.

Asynkronisten ja synkronisten virheiden käsittelyn yhteensopimattomuudesta ja välttämättömästä toistettavuudesta johtuen takaisinkutsufunktioiden käyttäminen johtaa virheidenkäsittelyä tehdessä monimutkaiseen ja virhealttiiseen koodiin.

Koska takaisinkutsufunktiot toimivat sivuvaikutuksina, eivätkä ne anna operaatiosta ensiluokkaista arvoa käsiteltäväksi, on jo olemassa olevien operaatioiden yhdistäminen ja ketjujen muodostaminen mahdotonta tai hankalaa. Myös samanaikaisuuden hallinta pelkästään takaisinkutsufunktioita käyttäessä vaatii virhealtista manuaalista laskureiden ylläpitämistä.

4.3.1 Tiedostojen ryhmittely

Tiedostojen ryhmittelyn ratkaisun lähdekoodi esitetään kuviossa 30. Ensin tämänhetkisen työskentelykansion tiedostojen nimien lukuoperaatio laitetaan alulle kutsumalla *readdir*-metodia. Metodille annetaan takaisinkutsufunktio, jossa tiedostonimet ovat *result*-taulukossa.

Tiedostonimien ryhmittely ensimmäisen kirjaimen perusteella tapahtuu järjestämällä taulukko aakkosjärjestykseen kutsumalla *sort*-metodia ja kartoittamalla sen jälkeen taulukon jokainen alkio ryhmärakenteeksi. Tällöin kartoitetussa taulukossa on vielä kopioita samoista ryhmistä, jotka suodatetaan taulukosta poista käyttämällä taulukon *filter*-metodia.

Ryhmittelyn jälkeen tulee jokaisen ryhmään kuuluvan tiedoston tiedostokoko selvittää. Tiedostokoon selvittämiseen tarvitaan tiedostojärjestelmämoduuli *fs* tarjoamaa *stat*-metodia. Metodi on asynkroninen, ja koska ongelman vaatimuksena on, että tiedostokoot selvitetään samanaikaisesti, tulee *stat*-operaatiot aloittaa samanaikaisesti.

Operaatioiden valmistumisen seuraamista varten luodaan laskurit *groupsStatted* sekä *filesStatted*. *groupsStatted* seuraa kuinka monen ryhmän tiedostokoot on kokonaisuudessaan jo selvitetty, kun taas *filesStatted*-laskurilla seurataan kuinka monta tietyn ryhmän tiedostokokoa on selvitetty.

Kun kaikki ryhmän tiedostokoot ovat selvitetty, kasvatetaan ryhmälaskuria, ja kun kaikkien ryhmien tiedostokoot ovat selvitetty, siirrytään *allGroupStatsComplete*-funktion suorittamiseen, joka järjestelee tiedot vaatimusten mukaisesti ryhmien kuuluvien tiedostomäärien mukaisesti ja muotoilee tiedot merkkijonoksi.

Ratkaisuun käytetään 109 riviä koodia.

Lopuksi tarkistetaan antaako toteutus oikean tuloksen. Tarkistus esitetään kuviossa 31.

```
petka@petka-VirtualBox ~/oppari
$ node problem1test.js problem-1-callback-solution.js
test success, result output:
Group: b Filecount: 3 Total size: 177
Group: a Filecount: 2 Total size: 80
Group: f Filecount: 2 Total size: 220
Group: k Filecount: 1 Total size: 65
Group: n Filecount: 1 Total size: 52
Group: r Filecount: 1 Total size: 102
```

Kuvio 30. Ratkaisun oikeellisuuden tarkistuksen tulos.

```
petka@petka-VirtualBox ~/oppari
$ node perftest.js problem-1-callback-solution.js
time taken:  811 mx max memory 22.29 MB
```

Kuvio 31. Ratkaisun suorituskykymittauksen tulos.

4.3.2 Tiedostojen yhteenliittäminen

Tiedostojen yhteenliittämisen ratkaisun lähdekoodi esitetään liitteessä 1. Ensin tämänhetkisen työskentelykansio saadaan kutsumalla *process.cwd*-metodia. Kansion tiedostojen nimien lukuoperaatio laitetaan alulle kutsumalla *readdir*-metodia. Metodille annetaan takaisinkutsufunktio, jossa tiedostonimet ovat *files*-taulukossa.

Vaatimuksena ratkaisulle on se, että tiedostojen lukeminen tapahtuu samanaikaisesti, mutta kuitenkin vain niin, että maksimissaan kaksi tiedoston lukemista on käynnissä samaan aikaan. Ratkaisussa tämä toteutetaan käyttämällä kahta eri laskuria, *filesRead* ja *fileReadsInProcess*.

filesRead-laskuri ilmaisee kuinka monta tiedostoa on jo luettu. Kun laskuri on kasvatettu yhtä suureksi kuin tiedostonimien määrä, on tiedostojen sisältöjen lukeminen valmistunut. Koska yksittäinen tiedostonlukuoperaatio voi valmistua missä järjestyksessä tahansa, voidaan tiedostojen sisällöt yhdistää vasta kun kaikkien tiedostojen sisällöt ovat luettu.

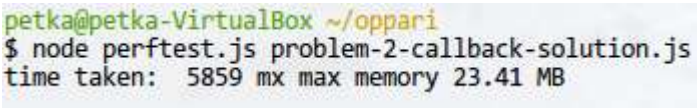
Yksittäisen tiedostonlukuoperaation valmistuessa täytyy tietää, mikä on tiedoston järjestysluku. Järjestysluku saadaan tekemällä *readFile*-metodille annettavasta takaisinkutsufunktiosta sulkeuma (closure), joka sulkee sisällensä *index*-järjestysluvun, joka voidaan operaation valmistuessa antaa argumenttina *readingComplete*-funktiolle.

fileReadsInProcess-laskuri pitää kirjaa kuinka monta tiedostonlukuoperaatiota on tietyllä ajan hetkellä päällä. Laskuri estää useamman kuin kahden operaation samanaikaisen suorituksen ja jokaisen operaation valmistuessa alkuperäiseen taulukkoon merkitään alkio tyhjäksi, jottei kyseisen alkion tiedoston sisältöä ruveta lukemaan uudestaan.

Ratkaisuun käytetään 73 riviä koodia.

```
petka@petka-VirtualBox ~/oppari
$ node problem2test.js problem-2-callback-solution.js
test success, result output:
beayvsmjzliaofdmhbuifmoyoucxfmfhbadpwtdvseokziyuictlezyescfrzqhibatbdspppogiyvae
zlnfevwwvkznoctbsxhxtgthiiwvtijqyurlelqskudzthijtxvmxuadqtdqftppqhogxicjtcnlwayx
xvwxiyaejfvvemicdoatkmuucphrtkwocxdsptbhuxfesvbpbjmblxrbyanjjklhypjjypnrkoteefly
qqjaegckrghjdilbzyxkkbtybyfnpngoeahpfgqypstedtvwdsadcxkbxogipodgsurischyomlfdaap
ooeugzeygdcslywlinlusaddkxxkcjegpssxogisrifqxcpbslsphngnakosuhmzrkocaaiaskrgpwvm
orckpeldcrplphmjncwrnwmkapmexxqtjdjjrzrxnfrpnzwkmrcwqeaydxrzhvntohrxuxhefnybsjko
uiwsviuzrjrltfrtkyokpjrkzfubuhjbwjzjrchmticprxosujzthgvdwbdutzethamslonrninbydad
cfnfbtyljxyukydgppzbjtvgckbshajvluuunsxrkjqljnlmbfbcxstgeyhynapgiifpvsmmppevsbwo
hygujrffdlnmivybtpjtrqhgogztjdcdkpqemormhwrdcfkblglupjam
```

Kuvio 32. Ratkaisun oikeellisuuden tarkistuksen tulos.



Kuvio 33. Ratkaisun suorituskykymittauksen tulos.

4.3.3 Yksittäisen tiedoston käsittely

Yksittäisen tiedoston käsittelyn ratkaisun lähdekoodi esitetään liitteessä 2. Satunnaisen tiedoston valintaa varten täytyy ensin tämänhetkisen työskentelykansion tiedostojen nimet lukea asynkronisesti taulukkoon *result* käyttämällä *readdir*-metodia. Kun taulukko on saatavilla, kutsutaan takaisinkutsufunktiota ja voidaan taulukosta ensin vaatimusten mukaisesti suodattaa pois itse ratkaisuohjelman lähdekooditiedoston nimi käyttämällä taulukon *filter*-metodia. Lähdekooditiedoston nimi saadaan viittaamalla *__filename*-nimiseen muuttujaan.

Suodatetusta taulukosta voidaan nyt valita satunnaisen tiedoston nimi *selectedFileName*-muuttujaan. Tiedoston *viimeksi muokattu* –päiväys saadaan selville käyttämällä *stat*-metodia, joka on siirrännän käytön vuoksi asynkroninen. Operaation valmistuttua, voidaan tuloksesta saada *viimeksi muokattu* –päiväys sen *mtime*-attribuutista. Päiväys on *Date*-tyyppiä, joten se muutetaan merkkijonoksi käyttämällä *toString*-metodia.

Seuravaksi tulee avata kahva tiedostoon käyttämällä *open*-metodia. Kahvan avauksen jälkeen saadaan kahvaan viittaus, ja tiedostoon voidaan kirjoittaa käyttämällä *write*-metodia, jolle annettaan argumentteina tiedostokahva sekä kirjoitettava sisältö. Koska kahva on avattu *a*+-tilassa, kaikki kirjoitus tapahtuu tiedoston loppuun. Kirjoitusoperaation jälkeen kahva tulee vielä vapauttaa. Vapautukseen käytetään *close*-metodia, jonka takaisinkutsufunktiota kutsuttaessa tiedetään, että vapautus on tapahtunut.

Ratkaisuun käytetään 72 riviä koodia.

Kuvio 34. Ratkaisun oikeellisuuden tarkistuksen tulos.

Kuvio 35. Ratkaisun suorituskykymittauksen tulos.

4.4 Takaisinkutsufunktiot async-kirjastoa käyttäen

Async-kirjasto tulee asentaa Node.js:n mukana tulevalla NPM-paketinhallintaohjelmistolla käyttäen komentoa *npm install async*. Asennuksen jälkeen sen rajapintaa voi käyttää kutsumalla *require("async")*-funktiota.

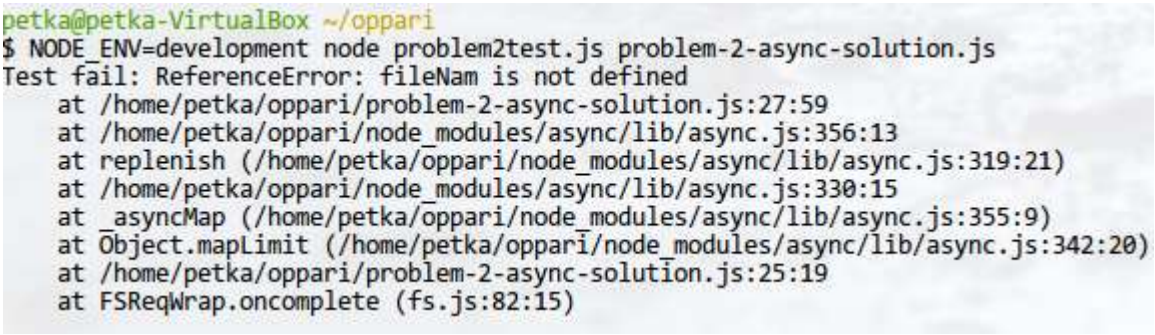
Async-kirjasto tarjoaa takaisinkutsufunktioiden käyttäjille apufunktioita, jotka eivät kuitenkaan poista takaisinkutsufunktioiden fundamentaalisimpia ongelmia, kuten asynkronisen virheenkäsittelyn ja synkronisen virheekäsittelyiden yhteensopimattomuutta sekä takaisinkutsufunktioiden yhdisteltävyyden vaikeutta.

Samanaikaisuuden hallintaan async-kirjasto tarjoaa kuitenkin suurta apua, sillä käyttäjän ei tarvitse samanaikaisuuden hallitsemista varten huolehtia itse erilaisista operaatiolaskureista. Lisäksi peräkkäin järjestyksessä suoritettaviin operaatioihin annettavat apufunktiot mahdollistavat käyttäjäkoodin litteyden, eikä sisennystaso kasva operaatiovaiheita lisättäessä hallitsemattomasti.

Vaikka async-kirjaston apufunktiot huolehtivat siitä, että sen antamia takaisinkutsufunktioiden usea kutsuminen ei aiheuta ikäviä sivuvaikutuksia, joutuu käyttäjä silti huolehtimaan saman takaisinkutsufunktion usealta kutsulta suojaamisen itse. Tällöin *onlyOnce*-apurifunktion käyttö on yhä välttämätöntä.

Koska async-kirjasto ei voi siepata poikkeuksia muuten kuin sen itse kutsumissa takaisinkutsufunktioissa, joutuu käyttäjä yhä käyttämään *try-catch*-lohkoja toistuvasti yhdistääkseen synkroniset poikkeukset asynkronisiin virheargumentteihin, jotta virheidenhallinta olisi yhtenäinen.

Async-kirjasto pyrkii paikkaamaan yhdisteltävyyden puutetta tarjoamalla suuren määrän apufunktioita. Esimerkiksi taulukkojen manipuloimiseen ei voi käyttää jo JavaScriptissä valmiiksi olemassa olevia metodeja kuten *map*, *filter* ja *reduce* vaan niistä tarjotaan erilliset takaisinkutsufunktioversiot: *async.map*, *async.mapSeries*, *async.mapLimit*, *async.filter*, *async.filterSeries*, *async.filterLimit* ja *async.reduce*.



Kuvio 36. Async-kirjastoa käyttävän ratkaisun antama pinojäljitys virhetilanteessa.

Kuviossa 37 esitetään async-kirjastoa käytettäessä käyttäjän saama pinojäljitys virhetilanteessa. Kuten pelkästään takaisinkutsufunktioita käyttäessä, async-kirjastoa käyttäessä pinojäljitykset sisältävät vain viimeisimmän virheeseen johtaneen tapahtuman.

4.4.1 Tiedostojen ryhmittely

Tiedostojen ryhmittelyn ratkaisun lähdekoodi esitetään kuviossa 38. Ratkaisu etenee aluksi aivan kuten vastaavan ongelman ratkaisu pelkästään takaisinkutsufunktioita käyttäessä. Kuitenkin samanaikaisuuden käsittely helpottuu huomattavasti käyttämällä *async.each*- ja *async.map*-apufunktioita. Apufunktiot mahdollistavat sen, ettei käyttäjän tarvitse itse ylläpitää operaatiolaskureita.

Ryhmien muodostamisen jälkeen voidaan ryhmät iteroida läpi käyttämällä async-kirjaston asynkroniseen iterointiin tarjoamaa *async.each*-metodia. Metodille annetaan toisena argumenttina takaisinkutsufunktio, jota kutsutaan jokaisen taulukossa olevan alkion kohdalla. Koska synkronista funktion paluuarvoa ei voida käyttää, takaisinkutsufunktio saa toisena argumenttina takaisinkutsufunktion, jota käyttäjän tulee kutsua kun alkiolle suoritettava asynkroninen operaatio on valmis.

Jokaisen ryhmän kohdalla ryhmässä olevat tiedostonimet kartoitetaan *async.map*-metodilla tiedostojen tiedostokooksi käyttämällä *stat*-metodia. Tiedostokoko saadaan *stat*-metodin tuloksen *size*-attribuutista. Viimeisenä argumenttina *async.map*-metodi saa takaisinkutsufunktion kun koko asynkroninen kartoitusoperaatio on valmis. Tällöin kutsutaan ylemmän asynkronisen iteraattorin, *async.each*, valmistumisen osoittamisen takaisinkutsufunktiota.

Ratkaisu käyttää 103 riviä koodia.

```
petka@petka-VirtualBox ~/oppari
$ node problem1test.js problem-1-async-solution.js
test success, result output:
Group: b Filecount: 3 Total size: 177
Group: a Filecount: 2 Total size: 80
Group: f Filecount: 2 Total size: 220
Group: k Filecount: 1 Total size: 65
Group: n Filecount: 1 Total size: 52
Group: r Filecount: 1 Total size: 102
```

Kuvio 37. Ratkaisun oikeellisuuden tarkistuksen tulos.

```
petka@petka-VirtualBox ~/oppari
$ node perftest.js problem-1-async-solution.js
time taken:  665 mx max memory 22.50 MB
```

Kuvio 38. Ratkaisun suorituskykymittauksen tulos.

4.4.2 Tiedostojen yhteenliittäminen

Tiedostojen yhteenliittämisen ratkaisun lähdekoodi esitetään liitteessä 3. Ensin tämänhetkisen työskentelykansio saadaan kutsumalla *process.cwd*-metodia. Kansion tiedostojen nimien lukuoperaatio laitetaan alulle kutsumalla *readdir*-metodia. Metodille annetaan takaisinkutsufunktio, jossa tiedostonimet ovat *files*-taulukossa.

Kun tiedostonimet ovat saatavilla, voidaan ne laittaa aakkosjärjestykseen kutsumalla taulukon *sort*-metodia. Async-kirjasto tarjoaa kätevän apufunktion rajoitetun samanaikaisuuden hallintaan. Käyttämällä *async.mapLimit*-metodia, voidaan helposti rajoittaa iteraattorifunktion sisältämien operaatioiden samanaikaisuus metodin ottamalla *limit*-argumentilla. Argumentiksi annetaan tässä *2*, vaatimusten mukaisesti.

Tiedostonimet kartoitetaan tiedostojen sisällöiksi antamalle *async.mapLimit*-metodille kolmanneksi argumentiksi funktio, joka kutsuu *readFile*-metodia ja antaa alkion takaisinkutsufunktion argumenttina *readFile*-metodille. Kun tiedoston lukeminen on valmis, kutsutaan alkion takaisinkutsufunktiota.

Viimeisenä argumenttina annettua takaisinkutsufunktiota kutsutaan lopulta kun kaikki tiedostot ovat luettu. Tulos-argumenttina annetaan *results*-taulukko, joka sisältää alkuperäisen taulukon alkiot kartoitettuna kartoittajafunktion määrittelemänä samassa järjestyksessä kuin ne ovat alkuperäisessä taulukossa. Tällöin käyttäjäkoodissa ei tarvitse huolehtia järjestyksestä ja taulukon alkiot voidaan suoraan supistaa yhdeksi merkkijonoksi käyttämällä taulukon *reduce*-metodia.

Ratkaisuun käytetään 56 riviä koodia.

```
petka@petka-VirtualBox ~/oppari
$ node problem2test.js problem-2-async-solution.js
test success, result output:
beayvsmjzliaofdmhbuifmoyoucxfmfhbadpwtdvseokziyuictlezyescfrzqhibatbdspppogiyvaez
lnfevwwvkznoctbsxhxtgthiiwvtijqyurlelqskudzthijtxvmxuadqtdqftppqhogxicjtcnlwayxxv
wxiyaejfvvemicdoatkmuucphrtkwocxdsptbhuxfesvbpbjmblxrbyanjjklhypjjypnrkoteeflyqqj
aegckrghjdilbzyxkkbtybyfnpngoeahpfgqypstedtvwdsadcxkbxogipodgsurischyomlfdaapooeu
gzeygdcslywlinlusaddkxxkcjegpssxogisrifqxcpbslsphngnakosuhmzrkocaaiaskrgpwvmorckp
eldcrplphmjncwrnwmkapmexxqtjdjjrzrxnfrpnzwkmrcwqeaydxrzhvntohrxuxhefnybsjkouiwsvi
uzrjrltfrtkyokpjrkzfubuhjbwjzjrchmticprxosujzthgvdwbdutzethamslonrninbydadcfnfbty
ljxyukydgppzbjtvgckbshajvluuunsxrkjqljnlmbfbcxstgeyhynapgiifpvsmmppevsbwohygujrff
dlnmivybtpjtrqhgogztjdcdkpqemormhwrdcfkblglupjam
```

Kuvio 39. Ratkaisun oikeellisuuden tarkistuksen tulos.

```
petka@petka-VirtualBox ~/oppari
$ node perftest.js problem-2-async-solution.js
time taken:  2685 mx max memory 22.33 MB
```

Kuvio 40. Ratkaisun suorituskykymittauksen tulos.

4.4.3 Yksittäisen tiedoston käsittely

Yksittäisen tiedoston käsittelyn ratkaisun lähdekoodi esitetään liitteessä 4. Ratkaisu koostuu useasta järjestyksessä suoritettavasta operaatiosta, jotka ovat riippuvaisia aikaisempien operaatioiden tuloksista. Tällöin on sopivaa käyttää async-kirjaston metodia *async.auto*, jolla voidaan kuvata jokainen operaatio antamalla sille nimi ja takaisinkutsufunktio.

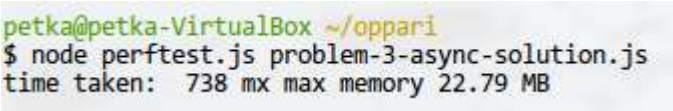
Operaation riippuvuudet edellisten operaatioiden tuloksiin voidaan ilmaista antamalla argumenttina operaatioiden nimet ennen takaisinkutsufunktioita. Tällöin operaation takaisinkutsufunktion saama *results*-avain-arvo-hakemisto sisältää operaatioiden tulokset. Tulos voidaan saada viittaamalla hakemistoon käyttämällä haluttavan operaation nimeä avaimena.

Kuvatut operaatioiden takaisinkutsufunktiot, jotka annetaan argumenttina *async.auto*-metodille, sisältävät käytännössä saman koodin kuin vastaavan ongelman ratkaisu pelkkiä takaisinkutsufunktioita käyttäessä. Async-kirjaston apufunktio mahdollistaa tässä lähdekoodin litteyden, eikä sisennystaso kasva liian suureksi niin, että koodin lukemiseen tarvitsisi käyttää vaakasuoraa rullaamista.

Ratkaisuun käytetään 70 riviä koodia.

```
petka@petka-VirtualBox ~/oppari
$ node problem3test.js problem-3-async-solution.js
test success
```

Kuvio 41. Ratkaisun oikeellisuuden tarkistuksen tulos.



Kuvio 42. Ratkaisun suorituskykymittauksen tulos.

4.5 Lupaukset bluebird-kirjastoa käyttäen

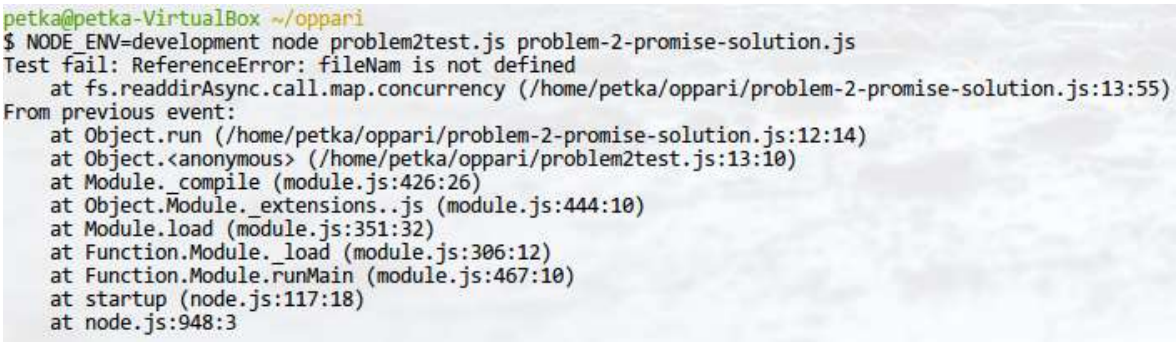
Bluebird-kirjasto tulee asentaa Node.js:n mukana tulevalla NPM-paketinhallintaohjelmistolla käyttäen komentoa *npm install bluebird*. Asennuksen jälkeen sen rajapintaa voi käyttää kutsumalla *require("bluebird")*.

Lupaukset mahdollistavat takaisinkutsufunktioiden sivuvaikutusten paradigmasta siirtymisen arvoparadigmaan, jossa yhdistely on mahdollista. Lisäksi lupauksia käyttäessä virheidenkäsittely hoidetaan kahden yhteensopimattoman mekanismin sijaan yhdellä mekanismilla, joka ei tarvitse saman koodin toistamista.

Lupaukset antavat vakuuden siitä, ettei niiden tapahtumankäsittelijöitä kutsuta useammin kuin kerran. Tämä tarkoittaa, ettei takaisinkutsufunktioita käyttäessä tarpeellista *onlyOnce*-apufunktiota tarvitse lupauksia käyttäessä. Lupausten tapahtumankäsittelijöiden sisällä tapahtuville poikkeuksille ei tarvitse erikseen määritellä *try-catch*-lohkoa, vaan heitetyt poikkeukset muutetaan automaattisesti hylätyiksi lupauksiksi.

Arvojen yhdisteltävyyden ansiosta lupaukset toteuttavan kirjaston ei tarvitse tarjota jokaiselle jo olemassa olevalle metodille erillistä asynkronista versiota, vaan käyttäjä voi yhdistellä haluttavan synkronisen metodikutsun lupausketjuun esimerkiksi lupauksen *call*-metodia käyttämällä.

Kuvio 45 havainnollistaa miten lupauksia käyttäessä virheen tapahtuessa pinojäljitys sisältää kaikki virheeseen johtaneiden tapahtumien ketjun.



Kuvio 43. Lupauksia käyttävän ratkaisun antama pinojäljitys virhetilanteessa.

4.5.1 Tiedostojen ryhmittely

Tiedostojen ryhmittelyn ratkaisun lähdekoodi esitetään kuviossa 46. Aluksi *fs*-moduulin rajapinnasta luodaan lupauksia käyttävä rajapinta metodin *Promise.promisifyAll* avulla. Kutsun jälkeen *fs*-moduulin rajapinta sisältää kaikista moduulin metodeista lupauksia palauttavan version. Lupauksia palauttavien metodien nimet ovat samat kuin vastaavat takaisinkutsufunktioiden nimet, mutta niiden loppuun on lisätty *Async*-pääte.

Tämänhetkisen työskentelykansion tiedostojen nimien lukuoperaatio laitetaan alulle kutsumalla *readdirAsync*-metodia. Koska metodilla on paluuarvo, lupaus, joka tulee tulevaisuudessa toteutumaan arvolla, voidaan arvoon halutut muutokset kuvata kutsumalla palautetun lupauksen metodeja välittömästi.

Taulukon järjestäminen aakkosjärjestyksessä kuvataan kutsumalla lupauksen *call*-metodia argumentilla *"sort"*. Seuraavaksi ketjussa kuvataan kartoitus tiedostonimistä tiedostoryhmiksi ja niiden suodatus.

Suodatuksen jälkeen ketjussa kuvataan uusi kartoitus, jonka sisällä kuvataan tiedostokokojen supistusoperaatio. Koska *group*-hakemiston *totalSize* arvo on vielä kartoituskäsittelijän sisällä lupaus, tulee käsittelijästä palauttaa lupaus, joka kuvaa hakemistoa, jossa arvot ovat saatavilla. Tähän käytetään *Promise.props*-metodia.

Seuraavaksi ketjussa kuvataan edellisestä vaiheesta syntyneen taulukon järjestely tiedostomäärän mukaan suurimmasta pienempään, kartoitus muotoiltuun merkkijonoihin ja lopuksi merkkijonojen yhdistäminen. Lupausten kartoitus takaisin takaisinkutsufunktioiden sivuvaikutuksiksi tapahtuu kätevästi *asCallback*-metodia käyttämällä.

Ratkaisuun käytetään 61 riviä koodia.

```
petka@petka-VirtualBox ~/oppari
$ node problem1test.js problem-1-promise-solution.js
test success, result output:
Group: b Filecount: 3 Total size: 177
Group: a Filecount: 2 Total size: 80
Group: f Filecount: 2 Total size: 220
Group: k Filecount: 1 Total size: 65
Group: n Filecount: 1 Total size: 52
Group: r Filecount: 1 Total size: 102
```

Kuvio 44. Ratkaisun oikeellisuuden tarkistuksen tulos.

```
petka@petka-VirtualBox ~/oppari
$ node perftest.js problem-1-promise-solution.js
time taken:  1089 mx max memory 27.79 MB
```

Kuvio 45. Ratkaisun suorituskykymittauksen tulos.

4.5.2 Tiedostojen yhteenliittäminen

Tiedostojen yhteenliittämisen ratkaisun lähdekoodi esitetään liitteessä 5. Kuten edellisessä ratkaisussa, täytyy ensin *fs*-moduulin rajapintaan luoda lupausrajapinnat käyttämällä *Promise.promisifyAll*-metodia.

Hakemiston tiedostonimien lukuoperaation lupaus aloittaa lupausketjun. Ketjussa kuvataan aluksi lupauksen toteutumisarvona tulevan tiedostonimitaulukon järjestäminen aakkosjärjestykseen käyttämällä lupauksen *call*-metodia argumentilla *"sort"*.

Seuraavaksi taulukolle kuvataan kartoitusoperaatio, jossa alkiot, jotka ovat tiedostonimiä, kartoitetaan tiedostonimiä vastaaviksi tiedostojen sisällöiksi. Jotta vain kaksi tiedostonlukuoperaatioita olisi käynnissä samanaikaisesti, annetaan *map*-metodille toisena argumenttina *concurrency: 2*.

Lopuksi kuvataan supistusoperaatio, jossa taulukko tiedostojen sisältöjä supistetaan yhdeksi merkkijonoksi. Tämä onnistuu käyttämällä lupauksen *reduce*-metodia.

Ratkaisuun käytetään 28 riviä koodia.

```
petka@petka-VirtualBox ~/oppari
$ node problem2test.js problem-2-promise-solution.js
test success, result output:
beayvsmjzliaofdmhbuifmoyoucxfmfhbadpwtdvseokziyuictlezyescfrzqhibatbdspppogiyvaez
lnfevwwvkznoctbsxhxtgthiiwvtijqyurlelqskudzthijtxvmxuadqtdqftppqhogxicjtcnlwayxxv
wxiyaejfvvemicdoatkmuucphrtkwocxdsptbhuxfesvbpbjmblxrbyanjjklhypjjypnrkoteeflyqqj
aegckrghjdilbzyxkkbtybyfnpngoeahpfgqypstedtvwdsadcxkbxogipodgsurischyomlfdaapooeu
gzeygdcslywlinlusaddkxxkcjegpssxogisrifqxcpbslsphngnakosuhmzrkocaaiaskrgpwvmorckp
eldcrplphmjncwrnwmkapmexxqtjdjjrzrxnfrpnzwkmrcwqeaydxrzhvntohrxuxhefnybsjkouiwsvi
uzrjrltfrtkyokpjrkzfubuhjbwjzjrchmticprxosujzthgvdwbdutzethamslonrninbydadcfnfbty
ljxyukydgppzbjtvgckbshajvluuunsxrkjqljnlmbfbcxstgeyhynapgiifpvsmmppevsbwohygujrff
dlnmivybtpjtrqhgogztjdcdkpqemormhwrdcfkblglupjam
```

Kuvio 46. Ratkaisun oikeellisuuden tarkistuksen tulos.

```
petka@petka-VirtualBox ~/oppari
$ node perftest.js problem-2-promise-solution.js
time taken:  2495 mx max memory 28.81 MB
```

Kuvio 47. Ratkaisun suorituskykymittauksen tulos.

4.5.3 Yksittäisen tiedoston käsittely

Yksittäisen tiedoston käsittelyn ratkaisun lähdekoodi esitetään liitteessä 6. Kuten edellisessä ratkaisussa, täytyy ensin *fs*-moduulin rajapintaan luoda lupausrajapinnat käyttämällä *Promise.promisifyAll*-metodia.

Lupaukset valitulle tiedostonimelle, tiedostopolulle sekä valitun tiedoston *viimeksi muokattu*-päiväykselle asetetaan vakioviittauksiin. Lupausten tuloksia tullaan tarvitsemaan myöhemmin, minkä vuoksi niihin on tarve viitata.

Tiedostonimen valinta johdetaan aloittamalla lupausketju kansion tiedostonimien lukuoperaatiolla. Operaation toteutumisarvolle kuvataan suodatusoperaatio käyttämällä lupauksen *filter*-metodia. Taulukosta suodatetaan pois lähdekooditiedoston nimi, joka on saatavilla *__filename* viittauksella. Seuraavaksi kuvataan lupauksen toteutumisen tapahtumankäsittelijä, jossa tulostaulukosta arvotaan satunnaisesti yksi alkio ja se palautetaan tapahtumankäsittelijän tuloksena.

Viimeksi muokattu –päiväys johdetaan lupauksesta tiedostopolulle, sillä sen selvittämiseksi tarvitsee tietää minkä tiedoston päiväys tarvitaan. Lupauksen toteutumisen tapahtumankäsittelijässä kutsutaan *statAsync*-metodia, joka palauttaa lupauksen, joka tulee toteutumaan annetun tiedostopolkuargumentin tiedoilla. Tietoihin kuvataan *mtime*-attribuutin hakuoperaatio käyttämällä lupauksen *get*-metodia argumentilla *”mtime”*. Lopuksi kuvataan *toString*-metodin kutsu, joka muuttaa edellisen lupauksen toteutumisarvon päiväystyypistä merkkijonotyypiksi.

Promise.join-metodin avulla voidaan odottaa usean erillisen lupauksen toteutumista, ja annettua toteutumisen tapahtumankäsittelijää kutsutaan kunkin annetun lupauksen vastaavalla toteutumisarvolla. Tapahtumankäsittelijän sisällä tiedostokahva avataan, siihen kirjoitetaan ja lopuksi kahva suljetaan.

Käyttäen *thenReturn*-metodia voidaan lupausketjun metodin kutsumiskohdalla sivuuttaa lupauksen normaali toteutumisarvo korvaamalla se arvolla, joka annetaan *thenReturn*-metodille argumenttinä.

Ratkaisuun käytetään 44 riviä koodia.

```
petka@petka-VirtualBox ~/oppari
$ node problem3test.js problem-3-promise-solution.js
test success
```

Kuvio 48. Ratkaisun oikeellisuuden tarkistuksen tulos.

```
petka@petka-VirtualBox ~/oppari
$ node perftest.js problem-3-promise-solution.js
time taken:  628 mx max memory 27.53 MB
```

Kuvio 49. Ratkaisun suorituskykymittauksen tulos.

5 Tulokset ja pohdinta

5.1 Tulokset

Ongelmassa 1 parhaan suorituskyvyn saavuttaa async-kirjastolla toteutettu ratkaisu, joka suoriutuu lupauksiin verrattuna lähes puolessa ajassa. Lupauksilla on kuitenkin mahdollista toteuttaa ratkaisu vain lähes puolella async-ratkaisun rivimäärästä. Pisteytys suosii luettavuutta ja käyttäjäkoodin matalaa kompleksisuutta suorituskyvyn yli, joten lupaukset saavat ongelmasta 1 keskimäärin eniten pisteitä.

Ongelmassa 2 parhaan suorituskyvyn saavuttaa jälleen async-kirjastolla toteutettu ratkaisu. Suorituskyky ei kuitenkaan eroa paljoa takaisinkutsufunktioihin tai lupauksiin verrattuna. Ratkaisuun käytetty koodirivimäärä eroaa huomattavasti lupauksilla toteutetun ratkaisun ja muiden ratkaisuiden välillä, rivimäärä on lähes kolmasosa takaisinkutsufunktioihin verrattuna ja puolet async-kirjastoa käyttävään ratkaisuun verrattuna.

Ongelman 3 ratkaisuissa suorituskykyerot ovat yhtä merkityksettömiä kuin ongelman 2, vaikkakin takaisinkutsufunktiot selviytyvät tällä kertaa nopeimmin. Lupauksilla on jälleen mahdollista toteuttaa ratkaisu, joka käyttää vähiten koodirivejä.

Kaikki toteutukset näyttivät virhetilanteessa viimeisimpänä rivinä oikean koodirivin, mutta vain lupaukset näyttivät myös edellisten tapahtumien pinojäljitykset.

Tulokset esitetään tarkemmin taulukossa 2.

Taulukko 2. Vertailun tulokset.

	Takaisinkutsufunktiot		**Async-kirjasto**		**Lupaukset**	
Ongelma 1	Tulos	Pisteet	Tulos	Pisteet	Tulos	Pisteet
Ratkaisun	109	9,17	103	9,70	61	16,39

rivimäärä						
Suoritusaika	811ms	10,65	665ms	11,96	1089ms	8,19
Muistin käyttö	22,29MB		22,50MB		27,79MB	
Yhteensä		19,82		21,66		24.58
Ongelma 2	Tulos	Pisteet	Tulos	Pisteet	Tulos	Pisteet
Ratkaisun rivimäärä	73	13,70	56	17,86	28	35,71
Suoritusaika	5859ms	5,13	2685ms	6,34	2495ms	5,47
Muistin käyttö	23,41MB		22,33MB		28,81MB	
Yhteensä		18,83		24,2		41,18
Ongelma 3	Tulos	Pisteet	Tulos	Pisteet	Tulos	Pisteet
Ratkaisun rivimäärä	72	13,89	70	14,29	44	22,73
Suoritusaika	627ms	12,45	738ms	11,16	628ms	11,59
Muistin käyttö	22,36MB		22,79MB		27,53MB	
Yhteensä		26,34		25,45		34,32
Pinojäljityksen käytetävyyden taso virhetilanteissa	4	15	4	15	5	20
Yhteensä		**79,99**		**86,31**		**120,08**

5.2 Pohdinta

Työssä verrattiin eri asynkroniseen ohjelmointiin saatavilla olevia työkaluja ja kirjastoja Node.js-ympäristössä. Vertailun kohteeksi valittiin takaisinkutsufunktiot, async-kirjasto sekä lupaukset bluebird-kirjaston toteuttamina. Vertailuja varten määritettiin kolme erilaista ongelmaa, joissa oli jokaisessa eri samanaikaisuusvaatimukset ja suoritettavat operaatiot.

Verrattuna takaisinkutsufunktioihin async-kirjasto ei parantanut käyttäjäkoodin luettavuutta merkittävästi ja kirjaston saama pistemäärä on vain 8% suurempi kuin takaisinkutsufunktioiden saama pistemäärä. Async-kirjaston käyttäminen ei näyttänyt vaikuttavan suorituskykyyn merkittävästi, eikä se parantanut tai huonontanut pinojäljitysten käytettävyyttä.

Lupausten käyttäminen paransi luettavuutta pistetasolla selkeästi ja lupausten saama yhteispistemäärä on 50% suurempi kuin takaisinkutsufunktioiden sekä 40% suurempi kuin

async-kirjaston saama yhteispistemäärä. Lupauksilla on myös mahdollista saada pinojäljityksissä koko virheeseen johtanut tapahtumaketju, kun taas async-kirjastolla tai takaisinkutsufunktioita käyttäessä vain viimeisimmästä tapahtumasta näkyy pinojäljitys. Lupausten käyttäminen ei vaikuta merkittävästi suorituskykykyyn ajallisesti, joskin muistinkäyttö on hieman runsaampaa.

Tulosten perusteella on helppo vetää johtopäätös, että ohjelmistokehittäjien tai -arkkitehtien, jotka valitsevat Node.js-alustan käytön ratkaisuissaan, tulisi olla käyttämättä takaisinkutsufunktioita tai async-kirjastoa silloin kun niiden käyttö voidaan korvata lupauksilla. Tosin, jos kehityskohde ei salli lupausten hieman korkeampaa muistinkäyttöä, tulee lupausten käyttöä arvioida tarkemmin, sillä ne käyttävät hieman enemmän muistia kuin takaisinkutsufunktiot tai async-kirjasto.

5.3 Tutkimusmenetelmän analyysi

Tutkimuksessa verrattiin asynkronisen ohjelmoinnin tekniikoita haluttavien ominaisuuksien kannalta. Haluttaviksi ominaisuuksiksi määriteltiin suorituskyky, käytettävyys sekä käyttäjäkoodin luettavuus ja kompleksisuus.

Jokaiselle verrattavalle ominaisuudelle määriteltiin mittarit, joiden kautta ominaisuuksia pystyi arvioimaan mittareista saatujen arvojen perusteella. Tällöin tulosten luotettavuus on riippuvainen valittujen mittareiden luotettavuudesta, käytännön toteutuksesta sekä siitä, missä määrin mittarit edustavat ja heijastavat mitattavaa ominaisuutta.

Käyttäjäkoodin luettavuutta ja kompleksisuutta mitattiin yksinkertaisesti laskemalla toteutetun ratkaisun lähdekoodin rivimäärä. On mahdollista, ettei pelkkä lähdekoodin rivimäärä kerro koko totuutta koodin luettavuudesta ja kompleksisuudesta. Esimerkiksi async-kirjastoa käyttävä ratkaisu yksittäisen tiedoston käsittelyn ongelmaan (Liite 4.) on rivimäärältään lähes sama kuin takaisinkutsuja käyttäessä (Liite 2.), mutta koodin sisennystaso pysyy async-kirjastoa käyttävässä ratkaisussa matalana.

Buse ym. (2010) esitteli mahdollisiksi luettavuuden mittareiksi lähdekoodin rivimäärän lisäksi tunnisteiden määrän, keskimääräisen koodirivin pituuden, lohkojen määrän, maksimirivipituuden, keskimääräisen sisennystason, avainsanojen määrän, tyhjien rivien määrän, tunnisteen maksimipituuden, kommenttien määrän, numeroiden ja välimerkkien määrän sekä silmukka- ja ehtorakenteiden määrän. Syvempi ja perusteellisempi tutkimus, joka käyttää useampia mittareita, voi johtaa erilaisiin luettavuustuloksiin kuin tämä työ.

Jokaisessa luettavuuden mittarissa on kuitenkin Buse ym. (2010) mukaan väärinkäytön mahdollisuus. Mittarit ovat kuvailevia, eikä niitä voida käyttää ohjeina luettavuuden lisäämiseksi. Esimerkiksi, jos pieni koodirivien määrä korreloi luettavuuden kanssa, ei se tarkoita, että ohjelma, joka on kirjoitettu yhdelle riville olisi luettava.

Tutkimuksessa luettavuustulokseen vaikuttaa myös ratkaisun toteuttajan osaamistaso. Matala osaamistaso johtaa tarpeettomaan kompleksisuuteen, eikä toteutus tällöin heijasta tekniikan olennaista kompleksisuutta. (McGregor 2006.)

Tekniikoiden suorituskykyä mitaattiin suorittamalla ratkaisu 1000 kertaa ja mittaamalla kuinka paljon tähän kului aikaa sekä kuinka paljon muistinkulutus nousi korkeimillaan suorituksen aikana. Tälläisen mittauksen suoritusaikaan voi vaikuttaa virheellisesti esimerkiksi kiintolevyn välimuisti ja JavaScript-kääntäjän tekemät optimoinnit suorituksen aikana. Tutkimuksessa ei otettu huomioon näitä tekijöitä. Lisäksi mittauksia suoritettiin vain yksi, eikä tilastotieteellisiä menetelmiä käytetty tulosten jalostamiseen. Syvempi ja perusteellisempi tutkimus, joka ottaa edellä mainitut tekijät huomioon, voi johtaa eri tuloksiin kuin tämä työ.

5.4 Oman oppimisen arviointi

Opinnäytetyön tekemisessä opin paljon async-kirjaston käyttämisestä, sillä se ei ollut minulle ennestään tuttu muuten kuin pinnallisesti. Opinnäytetyön aihe ja sisältö oli kuitenkin muuten minulle hyvin tuttu, mutta työ vaati kuitenkin paljon asioiden erittelyä ja jäsentämistä, mikä selvensi minulle joitakin seikkoja ja opetti minua kommunikoimaan aihepiiriin kuuluvia käsitteitä tehokkaammin ja selkeämmin.

Lähteet

Archibald, J. 2013. JavaScript Promises. Artikkeli. Luettavissa: http://www.html5rocks.com/en/tutorials/es6/promises/. Luettu: 9.8.2015.

Archibald, J. 2014. ES7 async functions. Artikkeli. Luettavissa: https://jakearchibald.com/2014/es7-async-functions/. Luettu: 9.8.2015.

Asche, R. 1996. Multithreading Performance. Artikkeli. Luettavissa: https://msdn.microsoft.com/en-us/library/ms810437.aspx. Luettu: 26.7.2015.

Buse, R. & Weimer, W. 2010. Learning a Metric for Code Readability. IEEE Transactions on Software Engineering, 36, 4, p. 546-58.

Caldwell, O. 2013. Handling concurrency and asynchronous JavaScript. Artikkeli. Luettavissa: http://oli.me.uk/2013/09/11/handling-concurrency-and-asynchronous-javascript/. Luettu: 25.7.2015.

Cantelon, M., Harter, M., Holowaychuk, T.J.. & Rajlich, N. 2013. Node.js in Action. Manning. Shelter Island, New York.

Creamer, J. 2013. Event-Based Programming: What Async Has Over Sync. Artikkeli. Luettavissa: http://code.tutsplus.com/tutorials/event-based-programming-what-async-has-over-sync--net-30027. Luettu: 20.11.2015.

Denicola, D. 2012. You're Missing the Point of Promises. Artikkeli. Luettavissa: https://blog.domenic.me/youre-missing-the-point-of-promises/. Luettu: 21.8.2015.

Dierx, P. 2015. A Beginner’s Guide to npm — the Node Package Manager. Artikkeli. Luettavissa: http://www.sitepoint.com/beginners-guide-node-package-manager. Luettu: 6.10.2015.

Hennessy, J. & Patterson D. 2003. Computer Architecture: a quantitative approach (3rd ed.). Morgan Kaufmann. Burlington, Massachusetts.

Hunter II, T. 2015. The long road to Async/Await in JavaScript. Artikkeli. Luettavissa: https://thomashunter.name/blog/the-long-road-to-asyncawait-in-javascript/. Luettu: 6.10.2015.

Hushan, B. 2015. Concurrency vs Multi-threading vs Asynchronous Programming : Explained. Artikkeli. Luettavissa: http://codewala.net/2015/07/29/concurrency-vs-multi-threading-vs-asynchronous-programming-explained/. Luettu: 3.8.2015.

McGregor, J. 2006. Complexity, it's in the mind of the beholder. Journal of Object Technology, 5, 1, p. 31-7.

Microsoft Corporation. 2015a: Futures. Tekninen dokumentaatio. Luettavissa: https://msdn.microsoft.com/en-us/library/ff963556.aspx. Luettu: 20.11.2015.

Microsoft Corporation. 2015b: Asynchronous Programming with Async and Await (C# and Visual Basic). Tekninen dokumentaatio. Luettavissa: https://msdn.microsoft.com/en-us/library/hh191443.aspx. Luettu: 20.11.2015.

Motta, Q. 2015. How do Promises Work?. Artikkeli. Luettavissa: http://robotlolita.me/2015/11/15/how-do-promises-work.html. Luettu: 20.11.2015.

Mozilla. 2015. EventTarget.addEventListener(). Tekninen dokumentaatio. Luettavissa: https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener. Luettu: 4.8.2015.

NPM Inc. 2015a. bluebird. Tekninen dokumentaatio. Luettavissa: https://www.npmjs.com/package/bluebird. Luettu: 20.11.2015.

NPM Inc. 2015b. Q. Tekninen dokumentaatio. Luettavissa: https://www.npmjs.com/package/q. Luettu: 20.11.2015.

NPM Inc. 2015c. async. Tekninen dokumentaatio. Luettavissa: https://www.npmjs.com/package/async. Luettu: 20.11.2015.

Nishanov, G. & Radigan, J. 2014. Resumable Functions v.2. Artikkeli. Luettavissa: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4134.pdf. Luettu: 4.8.2015.

Norvig, P. 2014. Teach Yourself Programming in Ten Years. Artikkeli. Luettavissa: http://norvig.com/21-days.html#answers. Luettu: 20.11.2015.

Oracle Corporation. 2010. Sun Studio 12: Performance Analyzer. Artikkeli. Luettavissa: https://docs.oracle.com/cd/E19205-01/819-5264/afamw/index.html. Luettu: 27.8.2015.

Oracle Corporation. 2014. Interface Future<V>. Tekninen dokumentaatio. Luettavissa: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html. Luettu: 9.8.2015.

Peticolas, D. 2009. An Introduction to Asynchronous Programming and Twisted. Artikkeli. Luettavissa: http://krondo.com/wp-content/uploads/2009/08/twisted-intro.pdf. Luettu: 25.7.2015.

Sreepathi, P. 2015. CS377P Programming for Performance. Luentomateriaali. Luettavissa: https://www.cs.utexas.edu/~sree/cs377p/fall2015/schedule/10-io.pdf. Luettu: 6.10.2015.

Vollmer, M. 2011. Understanding callback functions in Javascript. Artikkeli. Luettavissa: http://recurial.com/programming/understanding-callback-functions-in-javascript/. Luettu: 25.7.2015.

Liitteet

Liite 1. Tiedostojen ryhmittelyn ratkaisu käyttäen takaisinkutsufunktioita.

```
function run(done) {
    done = onlyOnce(done);
    var currentGroup = {fileNames: []};
    var called = false;

    try {
        fs.readdir(process.cwd(), function(err, result) {
            if (err) return done(err);

            try {
                var groups = result.sort()
                .map(function(fileName) {
                    if (!currentGroup.firstCharacter) {
                        currentGroup.firstCharacter = fileName[0];
                        currentGroup.fileNames.push(fileName);
                    } else if (fileName[0] === currentGroup.firstCharacter) {
                        currentGroup.fileNames.push(fileName);
                    } else {
                        currentGroup = {
                            fileNames: [fileName],
                            firstCharacter: fileName[0]
                        };
                    }
                    return currentGroup;
                }, {fileNames: []})
                .filter(function(group, index, groups) {
                    return groups.indexOf(group, index + 1) === -1;
                });

                var totalGroups = groups.length;
                var groupsStatted = 0;

                function groupStatsComplete() {
                    groupsStatted++;
                    if (groupsStatted >= totalGroups) {
                        allGroupStatsComplete();
                    }
                }

                groups.forEach(function(group) {
                    group.fileCount = group.fileNames.length;
                    var totalFilesToStat = group.fileCount;
                    var filesStatted = 0;
                    var totalFileSize = 0;

                    function statComplete(err, result) {
                        try {
                            if (err) done(err);
                            totalFileSize += result.size;
                            filesStatted++;
                            if (filesStatted >= totalFilesToStat) {
                                group.totalSize = totalFileSize;
                                groupStatsComplete();
                            }
                        } catch (e) {
                            done(e);
                        }
                    }

                    group.fileNames.forEach(function(fileName) {
                        var path = Path.join(process.cwd(), fileName);
                        fs.stat(path, statComplete);
                    });
                });

                function allGroupStatsComplete() {
                    var result = groups.sort(function(aGroup, bGroup) {
                            return bGroup.fileCount - aGroup.fileCount;
                        })
                        .map(function(group) {
                            return "Group: " + group.firstCharacter + " " +
                                   "Filecount: " + group.fileCount + " " +
                                   "Total size: " + group.totalSize;
                        })
                        .join("\n");
                    done(null, result);
                }
            } catch (e) {
                done(e);
            }

        });
    } catch (e) {
        done(e);
    }
}
```

Liite 2. Tiedostojen yhteenliittämisen takaisinkutsufunktioita käyttävän ratkaisun lähdekoodi

```
function run(done) {
    done = onlyOnce(done);

    try {
        fs.readdir(process.cwd(), function(err, files) {
            if (err) return done(err);
            try {
                files = files.sort();
                var filesRead = 0;
                var filesToRead = files.length;
                var fileReadsInProcess = 0;
                var results = [];

                function startReading(fileName, index) {
                    try {
                        if (fileName !== null && fileReadsInProcess < 2) {
                            fileReadsInProcess++;
                            files[index] = null;
                            fs.readFile(Path.join(process.cwd(), fileName), function(err, result) {
                                readingComplete(err, result, index);
                            });
                        }
                    } catch (e) {
                        done(e);
                    }
                }

                function readingComplete(err, result, index) {
                    try {
                        if (err) return done(err);
                        results[index] = result;
                        fileReadsInProcess--;
                        filesRead++;
                        files.forEach(startReading);
                        if (filesRead === filesToRead) {
                            var output = results.reduce(function(output, cur) {
                                return output + cur;
                            }, "");
                            done(null, output);
                        }
                    } catch (e) {
                        done(e);
                    }
                }
                files.forEach(startReading);
            } catch (e) {
                return done(e);
            }
        });
    } catch (e) {
        done(e);
    }
}
```

Liite 3. Yksittäisen tiedoston käsittelyn takaisinkutsufunktioita käyttävän ratkaisun lähdekoodi

```
function run(done) {
    done = onlyOnce(done);

    fs.readdir(process.cwd(), function(err, result) {
        try {
            if (err) done(err);
            const filePool = result.filter(function(fileName) {
                return fileName !== __filename;
            });
            const selectedFileName = filePool[Math.floor(Math.random() * filePool.length)];
            const filePath = Path.join(process.cwd(), selectedFileName);
            fs.stat(selectedFileName, function(err, result) {
                try {
                    if (err) done(err);
                    const modifiedDate = result.mtime.toString();
                    fs.open(filePath, "a+", function(err, handle) {
                        try {
                            if (err) done(err);
                            fs.write(handle, modifiedDate, function(err, result) {
                                try {
                                    if (err) done(err);
                                    fs.close(handle, function(err, result) {
                                        try {
                                            if (err) done(err);
                                            done(null, {
                                                fileName: selectedFileName,
                                                modifiedDate: modifiedDate
                                            });
                                        } catch (e) {
                                            done(err);
                                        }
                                    })
                                } catch (e) {
                                    done(err);
                                }
                            })
                        } catch (e) {
                            done(err);
                        }
                    });
                } catch (e) {
                    done(err);
                }
            });
        } catch (e) {
            done(err);
        }
    });
}
```

Liite 4. Tiedostojen ryhmittelyn ratkaisu käyttäen async-kirjastoa.

```
function run(done) {
    done = onlyOnce(done);
    var currentGroup = {fileNames: []};
    var called = false;

    try {
        fs.readdir(process.cwd(), function(err, result) {
            if (err) return done(err);

            try {
                var groups = result.sort()
                    .map(function(fileName) {
                        if (!currentGroup.firstCharacter) {
                            currentGroup.firstCharacter = fileName[0];
                            currentGroup.fileNames.push(fileName);
                        } else if (fileName[0] === currentGroup.firstCharacter) {
                            currentGroup.fileNames.push(fileName);
                        } else {
                            currentGroup = {
                                fileNames: [fileName],
                                firstCharacter: fileName[0]
                            };
                        }
                        return currentGroup;
                    }, {fileNames: []})
                    .filter(function(group, index, groups) {
                        return groups.indexOf(group, index + 1) === -1;
                    });

                async.each(groups, function(group, callback) {
                    group.fileCount = group.fileNames.length;
                    async.map(group.fileNames, function(fileName, callback) {
                        try {
                            var path = Path.join(process.cwd(), fileName);
                            fs.stat(path, callback);
                        } catch (e) {
                            callback(e);
                        }
                    }, function(err, stats) {
                        if (err) done(err);
                        try {
                            group.totalSize = stats.reduce(function (total, stat) {
                                return total + stat.size;
                            }, 0);
                            callback(null);
                        } catch (e) {
                            callback(e);
                        }
                    });

                }, function(err) {
                    try {
                        if (err) done(err);

                        var result = groups.sort(function(aGroup, bGroup) {
                                return bGroup.fileCount - aGroup.fileCount;
                            })
                            .map(function(group) {
                                return "Group: " + group.firstCharacter + " " +
                                       "Filecount: " + group.fileCount + " " +
                                       "Total size: " + group.totalSize;
                            })
                            .join("\n");
                        done(null, result);
                    } catch (e) {
                        done(e);
                    }
                });
            } catch (e) {
                done(e);
            }

        });
    } catch (e) {
        done(e);
    }
}
```

Liite 5. Tiedostojen yhteenliittämisen async-kirjastoa käyttävän ratkaisun lähdekoodi

```
const fs = require("fs");
const Path = require("path");

const async = require("async");

function onlyOnce(fn) {
    var called = false;
    return function() {
        if (!called) {
            called = true;
            return fn.apply(this, arguments);
        }
    };
}

function run(done) {
    done = onlyOnce(done);

    try {
        fs.readdir(process.cwd(), function(err, files) {
            if (err) return done(err);
            files.sort();
            async.mapLimit(files, 2, function(fileName, callback) {
                try {
                    const path = Path.join(process.cwd(), fileName);
                    fs.readFile(path, callback);
                } catch (e) {
                    callback(e);
                }
            }, function(err, results) {
                if (err) return done(err);
                try {
                    var output = results.reduce(function(output, result) {
                        return output + result;
                    }, "");
                    done(null, output);
                } catch (e) {
                    done(e);
                }
            });
        });

    } catch (e) {
        done(e);
    }
}
```

Liite 6. Yksittäisen tiedoston käsittelyn async-kirjastoa käyttävän ratkaisun lähdekoodi

```
function run(done) {
    done = onlyOnce(done);

    async.auto({
        files: function(callback) {
            fs.readdir(process.cwd(), callback);
        },
        selectedFileName: ["files", function(callback, results) {
            const filePool = results.files.filter(function(fileName) {
                return fileName !== __filename;
            });
            const selectedFileName = filePool[Math.floor(Math.random() * filePool.length)];
            callback(null, selectedFileName);
        }],
        filePath: ["selectedFileName", function(callback, results) {
            const path = Path.join(process.cwd(), results.selectedFileName);
            callback(null, path);
        }],
        modifiedDate: ["filePath", function(callback, results) {
            fs.stat(results.filePath, function(e, stat) {
                try {
                    if (e) return callback(e);
                    callback(null, stat.mtime.toString());
                } catch (e) {
                    callback(e);
                }
            });
        }],
        handle: ["selectedFileName", "modifiedDate", "filePath", function(callback, results) {
            fs.open(results.filePath, "a+", callback);
        }],
        writing: ["modifiedDate", "handle", function(callback, results) {
            fs.write(results.handle, results.modifiedDate, callback);
        }],
        closing: ["writing", "handle", function(callback, results) {
            fs.close(results.handle, callback);
        }]
    }, 1, function(err, results) {
        if (err) return done(err);
        done(null, {
            fileName: results.selectedFileName,
            modifiedDate: results.modifiedDate
        });
    });
}
```

Liite 7. Tiedostojen ryhmittelyn ratkaisu käyttäen lupauksia.

```
const fs = require("fs");
const Path = require("path");

const Promise = require("bluebird");
Promise.promisifyAll(fs);

function run(done) {
    var currentGroup = {fileNames: []};

    fs.readdirAsync(process.cwd()).call("sort")
        .map(function(fileName) {
            if (!currentGroup.firstCharacter) {
                currentGroup.firstCharacter = fileName[0];
                currentGroup.fileNames.push(fileName);
            } else if (fileName[0] === currentGroup.firstCharacter) {
                currentGroup.fileNames.push(fileName);
            } else {
                currentGroup = {
                    fileNames: [fileName],
                    firstCharacter: fileName[0]
                };
            }
            return currentGroup;
        }, {fileNames: []})
        .then(function(groups) {
            return groups.filter(function(group, index, groups) {
                return groups.indexOf(group, index + 1) === -1;
            });
        })
        .map(function(group) {
            group.fileCount = group.fileNames.length;
            group.totalSize = Promise.reduce(group.fileNames, function(totalSize, fileName) {
                const path = Path.join(process.cwd(), fileName);
                return fs.statAsync().get("size").then(function(size) {
                    return totalSize + size;
                });
            }, 0);

            return Promise.props(group);
        })
        .call("sort", function(aGroup, bGroup) {
            return bGroup.fileCount - aGroup.fileCount;
        })
        .map(function(group) {
            return "Group: " + group.firstCharacter + " " +
                   "Filecount: " + group.fileCount + " " +
                   "Total size: " + group.totalSize;
        })
        .call("join", "\n")
        .asCallback(done);
}
```

Liite 8. Tiedostojen yhteenliittämisen lupauksia käyttävän ratkaisun lähdekoodi

```
const fs = require("fs");
const Path = require("path");

const Promise = require("bluebird");
Promise.promisifyAll(fs);

function run(done) {
    fs.readdirAsync(process.cwd())
            .call("sort")
            .map(function(fileName) {
                const path = Path.join(process.cwd(), fileName);
                return fs.readFileAsync(path).call("toString");
            }, {concurrency: 2})
            .reduce(function(output, current) {
                return output + current;
            }, "")
            .asCallback(done);
}

exports.testPerformance = function(done) {
    run(function() {
        done();
    });
};

exports.run = run;
```

Liite 9. Yksittäisen tiedoston käsittelyn lupauksia käyttävän ratkaisun lähdekoodi

```
const fs = require("fs");
const Path = require("path");

const Promise = require("bluebird");
Promise.promisifyAll(fs);

function run(done) {
    const selectedFileNamePromise = fs.readdirAsync(process.cwd()).filter(function(fileName) {
        return fileName !== __filename;
    }).then(function(filePool) {
        return filePool[Math.floor(Math.random() * filePool.length)];
    });

    const pathPromise = selectedFileNamePromise.then(function(fileName) {
        return Path.join(process.cwd(), fileName);
    });

    const modifiedDatePromise = pathPromise.then(function(path) {
        return fs.statAsync(path).get("mtime").call("toString");
    });

    Promise.join(selectedFileNamePromise,
                 pathPromise,
                 modifiedDatePromise, function(fileName, path, modifiedDate) {
        return fs.openAsync(path, "a+").then(function(handle) {
            return fs.writeAsync(handle, modifiedDate).then(function() {
                return fs.closeAsync(handle);
            });
        }).thenReturn({
            fileName: fileName,
            modifiedDate: modifiedDate
        });
    }).asCallback(done);
}

exports.testPerformance = function(done) {
    run(function() {
        done();
    });
};

exports.run = run;
```