Tommi Varho

Samanaikaisuuden toteutuksen vaihtoehdot C#-kielessä

Tietotekniikan Kandidaatin tutkielma

1. joulukuuta 2022

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Tommi Varho

Yhteystiedot: tommi.o.varho@student.jyu.fi

Ohjaaja: Jonne Itkonen

Työn nimi: Samanaikaisuuden toteutuksen vaihtoehdot C#-kielessä

Title in English: Implementation choices of Concurrency in C#-language

Työ: Kandidaatin tutkielma

Opintosuunta: Tietotekniikka

Sivumäärä: 24+0

Tiivistelmä: Tässä tutkielmassa tuodaan esille samanaikaisuuden toteutuksen eri mahdollisuuksia C#-kielessä. Tutkielma esitetään tekstein, kuvioin ja koodiesimerkein. Lopussa tehdään vertailu näistä samanaikaisuuden toteutus vaihtoehdoista C#-kielessä.

Avainsanat: Samanaikaisuus, rinnakkaisuus, reaktiivinen ohjelmointi, asynkronisuus, actor, promise.

Abstract: In this study, we will see different possibilities, how to implement concurrency in C#-language. This study is presented with texts, figures and code examples. At the end, a comparison is made of these concurrency implementation options in the C#-language.

Keywords: Concurrency, parallelism, reactive programming, asynchronous, actor, promise

Kuviot

Kuvio 1. Samanaikaisuus kuvio mukailtu lähteestä Lewis ja Berg 1995 2
Kuvio 2. Rinnakkaisuus kuvio mukailtu lähteestä Lewis ja Berg 1995...................... 2
Kuvio 3. Fork ja join 4
Kuvio 4. Parallel invoke mukailtu (“Parallel.Invoke Method” 2022) 5
Kuvio 5. Synkronisuuden ja asynkronisuuden erot 6
Kuvio 6. Async ja await mukailtu (“Async and Await in C#” 2022)......................... 8
Kuvio 7. Takaisinkutsu 9
Kuvio 8. Delegaatti mukailtu (“Using Delegates (C# Programming Guide)” 2022)........ 10
Kuvio 9. Actor, eli toimija 11
Kuvio 10. Tapahtumavirta.. 13
Kuvio 11. Reaktiivinen laajennos mukailtu (“ASP.NET Core Series 05: Don’t block your code, be reactive” 2022) 14

Sisällys

1 JOHDANTO ... 1

2 SAMANAIKAISUUS ... 2
 2.1 Monisäikeisyys ... 3
 2.2 *Fork ja join* ... 4

3 ASYNKRONISUUS ... 6
 3.1 *Promise*, eli lupaus ... 7
 3.2 *Callback*, eli takaisinkutsu ... 9
 3.3 *Actor*, eli toimija ... 11
 3.4 Reaktiivinen ohjelmointi ... 12

4 ANALYYSI ... 15

5 YHTEENVETO ... 17

LÄHTEET ... 18

1 Johdanto

Tämä kandidaatintutkielma käsittelee samanaikaisuuden toteutuksen mahdollisuuksia, erityisesti C#:n kannalta. Opintojen aikana on hyvin vähän käsitelty samanaikaisuuden käsitettä tai toteutusta. Ei ole myöskään tutustuttu miten tietokone hoitaa taustalla tehtävien suorittamista. Miksi tietokone ei enää lukkiudu, kun suoritetaan jotain pidempää tai vaikeampaa tehtävää. Näihin vastauksena on samanaikaisuus.

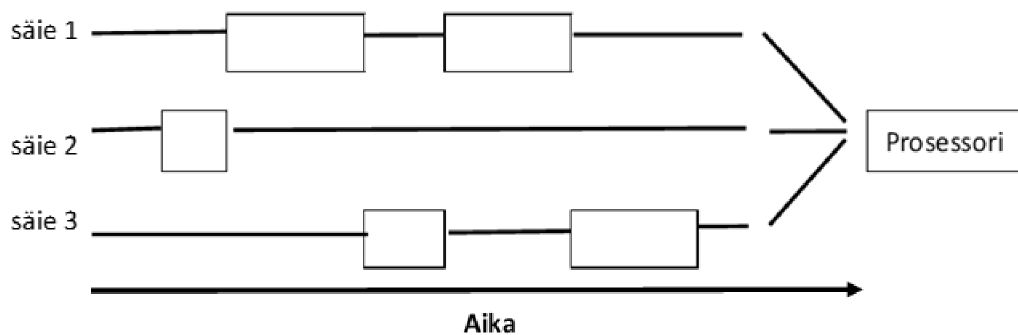
Tämän tutkielman tavoitteena on tuoda lukijalle yleiskäsitys samanaikaisuuden toteutus mahdollisuuksista, erityisesti C#:in kannalta. Tuodaan siis esille eri keinoja toteuttaa samanaikaisuus. Tutkielma toteutetaan kirjallisuuskatsauksena. Tutkitaan uusia ja vanhoja tieteellisiä lähteitä, sekä tutustutaan muun muassa C#-kielen ja .NET-alustan verkkodokumentaatioon.

Tässä tutkielmassa käytetään seuraavia termejä tai peruskäsitteitä. Ensinnäkin ohjelmaa, joka on käynnissä, kutsutaan prosessiksi. Ohjelmakokonaisuus voi suorittaa monta samaan aikaan käynnissä olevaa prosessia tai tehtävää. Tehtävä on ohjelmoinnin perusyksikkö. Sen käsite voi tarkoittaa työn yksikköä ohjelmassa tai ohjelman suoritusta, eli prosessia, riippuen määrittelystä tai siitä yhteydestä missä sanaa käytetään. Prosessi voi sisältää monta säiettä samaan aikaan. Kukin säie käyttää samoja resursseja, kuten esimerkiksi muisti, kuin sen prosessi, joka säikeen sisältää. Lisäksi jokainen säie voi suorittaa omia käskyjään riippumatta muiden säikeiden toiminnasta (Lewis ja Berg 1995, luku 2).

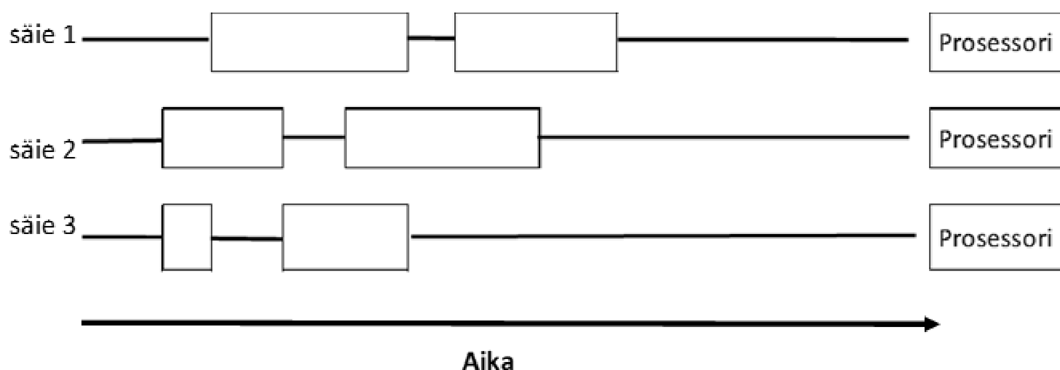
Useiden prosessien tai tehtävien toteuttamista samaan aikaan kutsutaan englanniksi *concurrency* ja suomeksi samanaikaisuus, tämä käydään läpi kappaleessa 2. Lisäksi kappaleessa käydään muutama samanaikaisuuteen liittyvän toteutustavan selitys ja se miten se on toteutettu C#:lla. Kappaleessa 3 selitetään asynkronisuuden käsite. Tämän lisäksi käydään läpi muutama asynkronisuuden toteutustapa, kuten reaktiivinen ohjelmointi. Lisäksi selitetään, miten toteutustavat on toteutettu C#:lla. Kappaleessa 4 analysoidaan C# samanaikaisuuden toteutustapoja sekä niiden soveltuvuutta eri tilanteisiin, hyötyjä ja mahdollisia huomiotavia riskejä. Lopuksi kappaleessa 5 tehdään yhteenveto tästä tutkielmasta.

2 Samanaikaisuus

Samanaikaisuus käsitteenä tarkoittaa monen prosessin suorittamista yhtä aikaa. Käytännössä se voi tarkoittaa sitä, että prosessit suoritetaan vuorotelleen kahdessa tai useammassa eri säikeessä, yhdessä prosessorissa, kuten näkyy kuviossa 1 (Lewis ja Berg 1995, luku 2). Kuviossa 1 tämä kuvataan vaakaviivoilla ja laatikoilla, jotka kuvaavat säikeitä ja prosesseja suhteessa aikaan. Säikeet suoritetaan kuviossa vuorotellen, koska on vain yksi prosessori.



Kuvio 1. Samanaikaisuus kuvio mukailtu lähteestä Lewis ja Berg 1995



Kuvio 2. Rinnakkaisuus kuvio mukailtu lähteestä Lewis ja Berg 1995

Toisaalta, jos eri prosesseja toteutetaan monessa eri säikeessä samaan aikaan useammassa prosessorissa, kutsutaan tätä rinnakkaisuudeksi, kuten näkyy kuviossa 2 (Lewis ja Berg 1995, luku 2). Kuviossa 2 prosessit kuvataan laatikoilla ja säikeet viivoilla. Siinä kuvataan eri säikeissä tapahtuvia prosesseja rinnakkaisesti suhteessa aikaan.

Rinnakkaisuus voidaan katsoa samanaikaisuuden erityiskäsitteeksi. Siinä eri prosesseja to-

teutetaan monessa eri säikeessä samaan aikaan useammassa prosessorissa. Toisin kuin samanaikaisuudessa, rinnakkaisuudessa prosessit toteutuvat aidosti samaan aikaan.

Samanaikaisuuteen voidaan lukea lisäksi mukaan käsite asynkronisuus. Asynkronisuus tarkoittaa ohjelmointia, jossa yksi tehtävä alkaa, mutta se ei estä muiden tehtävien suorittamista (Davies 2012). Asynkronisuus käsitellään kappaleessa 3.

Samanaikaisuuden toteutuksessa pitää ottaa huomioon mahdolliset virhe- ja ongelmatilanteet. Fiedor ym. 2011 kokoaa samanaikaisuuden ongelmia konferenssissa, joista tässä kappaleessa seuraavaksi käsitellään *kilpajuoksutilanne (engl. datarace)* ja *lukkiutuma (engl. deadlock)*. Fiedor esitykseessään selvittää että, *kilpajuoksutilanne* on ongelma, jossa kaksi tai useampi säie, yrittää saada tietoja samasta muuttujasta. Näistä säikeistä ainakin yksi säie yrittää kirjoittaa samalla, tällöin voi tulla eri tuloksia riippuen missä järjestyksessä muuttujaan pääsee käsiksi. *lukkiutuma* on taas ongelma, jossa kaksi tai useampi säie odottavat signaalia jatkaakseen toimintaansa, mutta tämä signaali pitäisi tulla toiselta säikeeltä, joka taas odottaa vastaavaa signaalia ensimmäiseltä säikeeltä.

2.1 Monisäikeisyys

Monisäikeisyys on samanaikaistamisen yksi keino. Siinä prosessit jaetaan säikeiksi ja ne mahdollistavat monen tehtävän suorituksen samanaikaisesti (Lewis ja Berg 1995, luku 2). Esimerkiksi yksi säie voi tehdä laskutehtäviä ja toinen säie ylläpitää käyttöliittymää, ei siis tarvitse odottaa laskutehtävän lopputulosta ennen kuin käyttöliittymä voi jatkaa omaa toimintaa. Kuten aiemmin kuviossa 1 esitettiin, jos on vain yksi prosessori, säikeet odottavat vuoroa toteuttaakseen tehtävänsä. Skaalattavissa järjestelmissä monisäikeisyys on hyödyllinen, koska työmäärän lisääntyessä voidaan lisätä säikeitä toteuttamaan tehtäviä samanaikaisesti ("Using threads and threading" 2022).

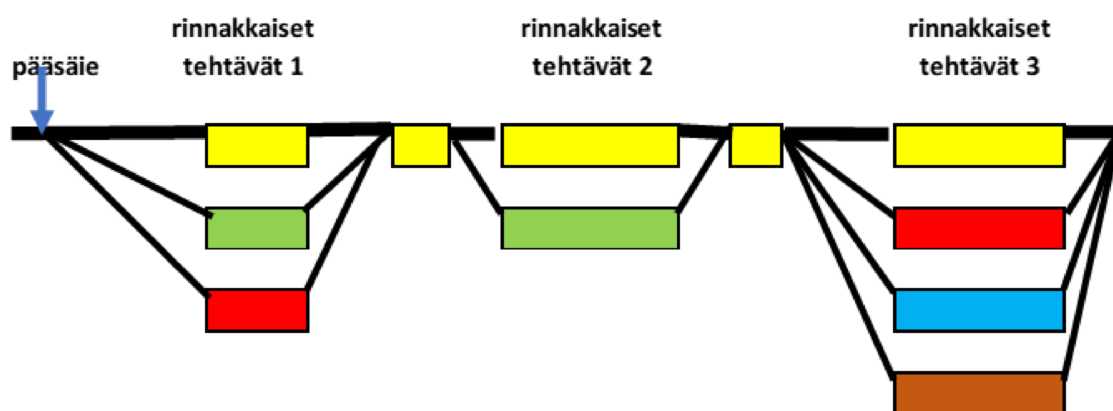
Nykyään C#:ssa käytetään *thread pool* luokkaa. Se sijaitsee .NET-alustan *System.Threading* kirjastossa. *Thread pool* luokalla voidaan automaattisesti luoda useita säikeitä toteuttamaan monia samanaikaisia tehtäviä monisäikeisyyden avulla (Cleary 2019). Esimerkiksi uudelle luodulle säikeelle annetaan aliohjelma, joka odottaa 5 sekuntia. Tällöin tämä aliohjelma odottaa sen 5 sekuntia uudessa säikeessä ja pääohjelma voi suorittaa muita tehtäviä alkupe-

räisessä säikeessä.

2.2 *Fork ja join*

Fork ja join -metodi on yksi hyvä tapa toteuttaa monisäikeisyyttä. Conway 1963 esitti ensimmäisenä idean tästä metodista. Tässä metodissa *fork* jakaa tehtävän moneen pienempään tehtävään ja nämä tehtävät jaetaan monelle eri säikeelle, jotta niitä voidaan suorittaa monella eri prosessorilla rinnakkaisesti. *Fork*-komennon jälkeen tarvitaan *join*, joka odottaa kaikkien pienempien tehtävien valmistumista ja yhdistää ne taas uudestaan yhdeksi tehtäväksi. Tällä tavalla pystytään nopeasti suorittamaan tehtäviä, joita voidaan jakaa osiin.

Cleary 2019 huomauttaa, että *fork ja join* -metodin yksi ongelma on se, että metodi ei aina välttämättä osaa jakaa tietokoneen laskentatehoa optimaalisesti, jos tehtävät ovat liian isoja tai pieniä. Cleary 2019 toteaa, että *fork ja join* -metodi on hyvä tapa toteuttaa esimerkiksi asiakasrajapinnan tehtävät, koska asiakkaan käyttöpäätteen prosessoritehoja ei välttämättä paljoa käytetä. Kuviossa 3 näytetään tehtävän jakautuvan moniin rinnakkaisiin tehtäviin.



Kuvio 3. Fork ja join

Parallel.invoke on C# .NET *System.Threading.Tasks* kirjastossa oleva *fork ja join* -metodi. *Parallel.invoke* toteuttaa metodin molemmat tehtävät, sen jakamisen ja takaisin yhdistämisen. Cleary 2019 kertoo, että metodissa tarvitsee antaa vain ne tehtävät, mitä halutaan suorittaa rinnakkain. Metodi hoitaa kaikki säikeiden ajoitukset ja skaalaukset tietokoneen prosessorien määrän mukaan

```
Parallel.Invoke(
    () =>
        {
        Console.WriteLine("Ensimmainen testi, Thread={0}",
        Thread.CurrentThread.ManagedThreadId);
        },
    () =>
        {
        Console.WriteLine("Toinen testi, Thread={0}",
        Thread.CurrentThread.ManagedThreadId);
        },
    () =>
        {
        Console.WriteLine("Kolmas testi, Thread={0}",
        Thread.CurrentThread.ManagedThreadId);
        }
);/*Yksi mahdollinen tuloste:
* Toinen testi, Thread=4
* Ensimmainen testi, Thread=1
* Kolmas testi, Thread=5 */
```

Kuvio 4. Parallel invoke mukailtu (“Parallel.Invoke Method” 2022)

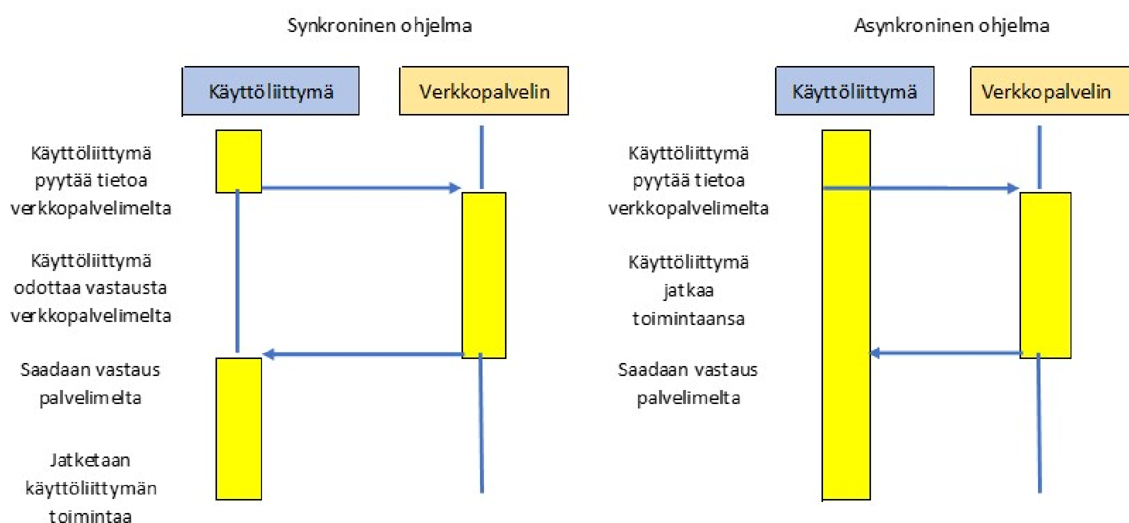
Kuviossa 4 käytetään *Parallel.Invoke* -metodia tulostamaan tässä esimerkissä kolme eri tekstiä. Nämä tekstit tulostuvat eri säikeillä rinnakkaisesti ja siksi tekstit tulostuvat satunnaisessa järjestyksessä.

Lisäksi *System.Threading.Tasks*-kirjastossa on muun muassa metodit *Parallel.ForEach* ja *Parallel.For*. Nämä metodit toimivat kuten *foreach* ja *for* -metodit, mutta jakavat tehtävät moneen eri osaan, jotta niitä voidaan suorittaa rinnakkaisesti (Cleary 2019, luku 1).

3 Asynkronisuus

Asynkroninen ohjelmointi on sitä, että yksi tehtävä alkaa, mutta ohjelma ei odota sen valmistumista ennen ohjelmakoodin jatkamista. Tällöin voi tehtäviä tapahtua samanaikaisesti, toisin kuin synkronisessa ohjelmoinnissa, jossa odotetaan jokaisen prosessin loppuun suorittamista ennen seuraavaan tehtävään siirtymistä. Asynkronisessa ohjelmoinnissa vapautetaan pääsäie jatkamaan toimintaansa ja ei jäädä odottamaan muiden säikeiden tehtävien loppuun suorittamista. Tämä vapauttaa prosessointi kykyä muille prosesseille (Davies 2012). Tietokoneen ei tarvitse olla moniprosessorinen asynkronisessa ohjelmoinnissa, vaan se voi toimia yhdelläkin prosessorilla peräkkäin asynkronisessa järjestyksessä (Madsen, Lhoták ja Tip 2017a).

Esimerkkinä asynkronisuudesta voisi olla käyttöliittymä, joka pyytää tietoa verkon yli tietokannalta. Tässä tapauksessa voi hetken kestää saada vastaus pyyntöön. Jos käyttöliittymä on asynkroninen, ei se jää odottamaan vastausta vaan käyttöliittymä pystyy odottaessaan tekemään muita asioita. Jos käyttöliittymä olisi synkroninen, ei odottamisen aikana pystyisi tekemään mitään (Cleary 2019, luku 1). Eli käyttöliittymä lukittuisi ja sitä ei voisi käyttää odottamisen aikana. Kuviossa 5 näytetään juuri tämä. Synkronisessa ohjelmassa käyttöliit-



Kuvio 5. Synkronisuuden ja asynkronisuuden erot

tymä odottaa vastausta palvelimelta ja ei pysty tekemään odotusaikana muita toimintoja.

Asynkronisessa ohjelmassa taas käyttöliittymässä toiminta jatkuu ja sitä voi käyttää odotuksen aikana.

Vaikeus asynkronisella ohjelmoinnilla on tietää, milloin pitkäaikainen prosessi valmistuu. Synkronisella ohjelmoinnilla tämä ei ole ongelma, koska kaikki tapahtuu järjestyksessä peräkkäin. Asynkronisessa koodissa tulee ongelma, koska prosessit valmistuvat eri aikoina.

3.1 *Promise*, eli lupaus

Promise, eli suomeksi lupaus on keskeinen käsite asynkronisessa ohjelmoinnissa. Sen perusteella toteutetaan moni asynkroninen ohjelmointitapa.

Lupauksen mallin asynkronisessa ohjelmoinnissa esitti ensimmäisenä Friedman ja Wise 1978. Sitä edelleen kehitti nykyiseen muotoonsa Liskov ja Shrira 1988. He määrittelivät *promise* eli suomeksi lupauksen ohjelmassa paikanpitäjäksi arvolle, joka tulee olemaan olemassa tulevaisuudessa. Se luodaan ohjelmakutsun tapahtuessa. Kutsu suorittaa prosessinsa asynkronisesti ja antaa saadun tuloksen lupaukselle. Alkuperäinen ohjelmakutsun toimija saa tämän jälkeen tarvitsemansa tuloksen lupaukselta.

Lupaus voi olla olemassa kolmessa tilassa: odottamassa, tehty, tai hylätty (Madsen, Lhoták ja Tip 2017b). Odottamassa -tilassa lupaus odottaa arvoa, joka tulee tulevaisuudessa. Odottamassa –tilasta lupaus voi siirtyä tehty -tilaan tai hylätty -tilaan. Tehty –tilassa lupaus saa odottamansa arvon. Hylätty -tilassa lupaus saa virheilmoituksen, jota taas ohjelmassa voidaan käsitellä eri tavoin.

Async ja *await* -metodit ja *Task ja Task<T>* luokat toteuttavat *Promise* eli lupauksen mallin C#:ssa. Ne sijaitsevat *System.Threading.Tasks* .NET-kirjastossa. Goranova, Kalcheva-Yovkova ja Penkov 2015 konferenssissa esittivät, miten tehtäväpohjaiset *Async* ja *await* -metodit toimivat. *Task* luokka esittää yhtä tehtävää, joka tapahtuu asynkronisesti ja ei palauta tulosta. *Task<T>* luokka taas esittää yhtä tehtävää, joka tapahtuu asynkronisesti ja palauttaa tuloksen. *Async* avainsana tekee tehtävästä asynkronisen ja tällöin voi käyttää *await* avainsanaa. *Await* käynnistää asynkronisen prosessin. Se luo kuuntelijan, joka odottaa lupauksen valmistumista, ja välittömästi palauttaa suoritusoikeuden *async* tehtävää kutsuneelle ohjel-

```
1  public static void AsyncAwait()
2  {
3              var ajastin = new System.Diagnostics.Stopwatch();
4              ajastin.Start();
5              var tehtava1 = EnsimmainenTehtava();
6              var tehtava2 = ToinenTehtava();
7              Task.WaitAll(tehtava1, tehtava2);
8              watch.Stop();
9              Console.WriteLine($"Kulunut aika: " +
10       ajastin.ElapsedMilliseconds);
11 }
12 public static async Task EnsimmainenTehtava()
13 {
14             await Task.Run(() =>
15             {
16                         Thread.Sleep(5000);
17                         Console.WriteLine("Ensimmainen Tehtava");
18             });
19 }
20 public static async Task ToinenTehtava()
21 {
22             await Task.Run(() =>
23             {
24                         Thread.Sleep(2000);
25                         Console.WriteLine("Toinen Tehtava");
26             });
27 }
```

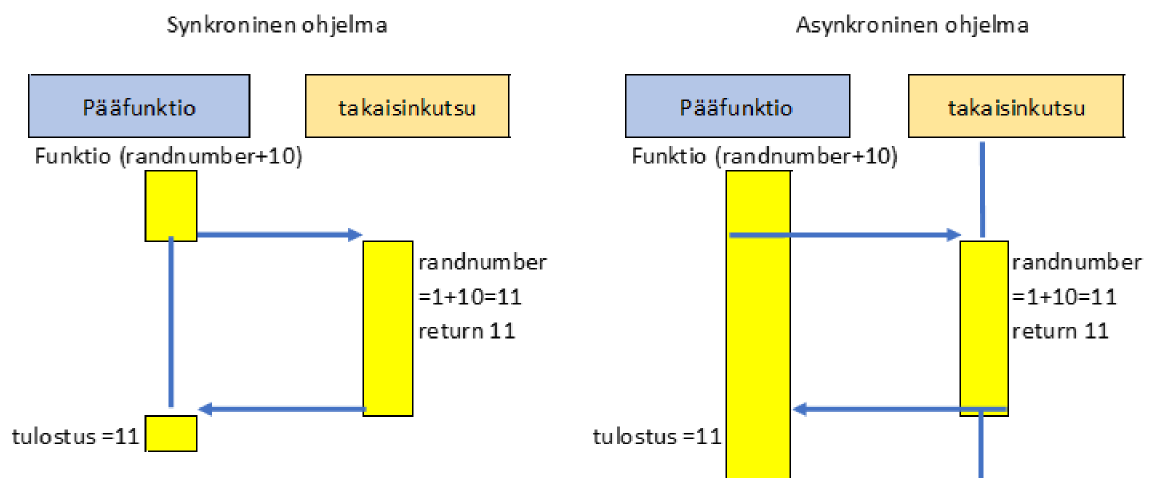
Kuvio 6. Async ja await mukailtu (“Async and Await in C#” 2022)

malle. *Async* prosessin saadessa lupauksen tuloksen, jatkaa se kesken jääneen prosessinsa suorituksen loppuun. Cleary 2019, luku 1 mukaan *Async ja await* sisältää lisäksi valmiina virheen hallinnan. Task-luokka palauttaa sisällään virheen kun se valmistuu, se tarvitsee vain ottaa kiinni *try catch* -metodilla. Tämä helpottaa huomattavasti virheiden hallintaa. Syme, Petricek ja Lomov 2011 keksi ensimmäisenä async ja await -metodin mallin.

“Asynchronous programming” 2022 mukaan tehdyssä kuviossa 6, toteutetaan kaksi eri tehtävää asynkronisesti. Tehtävien toiminta pysäytetään eri ajoiksi erillisillä säikeillä, kun ne taas jatkavat toimintaansa, tulostavat ne oman tekstinsä. Lopuksi tulostetaan tehtävien yhteensä kulunut aika. Kaikissa tehtävissä on *async ja await* avainsanat, jotka mahdollistavat asynkronisen tehtävien suorittamisen. Jos avainsanoja ei olisi, tehtävät suoritettaisiin siinä järjestyksessä, missä ne käynnistetään .

3.2 *Callback*, eli takaisinkutsu

Callback tai suomeksi takaisinkutsu on funktio, joka annetaan argumenttina toiselle funktiolle. Toisen funktion tarkoituksena olisi toteuttaa takaisinkutsufunktio heti tai tulevaisuudessa, riippuen siitä onko funktion toteutus synkroninen vai asynkroninen (Gallaba, Mesbah ja Beschastnikh 2015). Takaisinkutsufunktioita toteutetaan eri tavoilla eri kielissä, muun muassa funktio-osoittimilla tai lambda-lausekkeilla.



Kuvio 7. Takaisinkutsu

Kuviossa 7 esitetään synkroninen ja asynkroninen takaisinkutsu. Siinä pääfunktio kutsuu takaisinkutsu funktiota, joka luo satunnaisen luvun ja lisää siihen 10. Takaisinkutsu funktio suorittaa tämän kutsutun tehtävän ja palauttaa sen tuloksen pääfunktiolle.

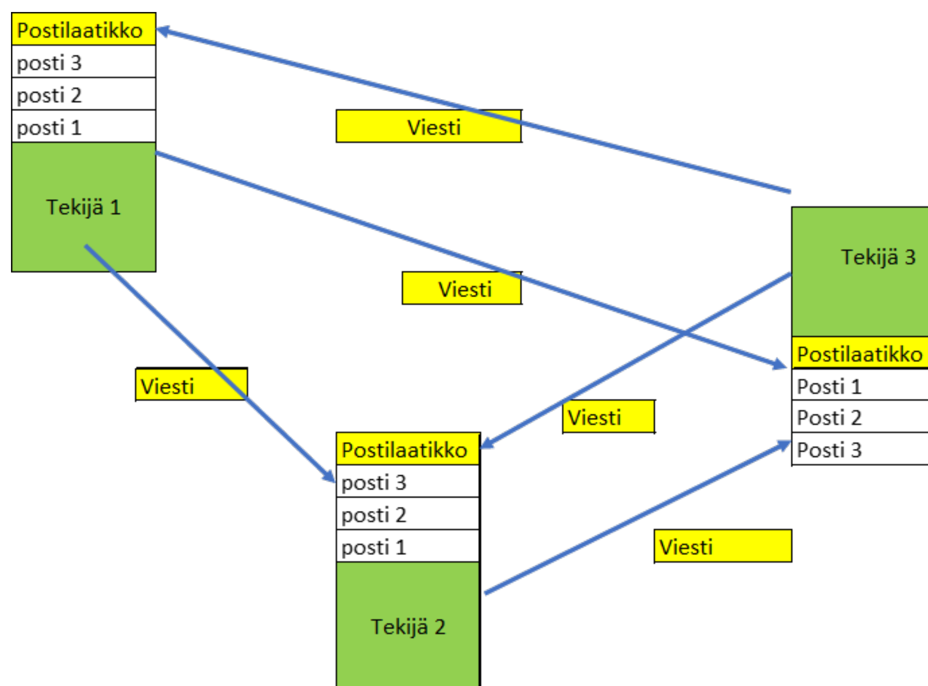
```
1  class Program
2  {
3      public delegate void Delegaatti(string message);
4      static void Main(string[] args)
5      {
6          // Luodaan delegate viittaus
7          Delegaatti handler = DelegaattiMetodi;
8              // Kutsutaan delegaattia
9          handler("Hello World");
10     }
11     // Luodaan metodi, johon viitataan
12     public static void DelegaattiMetodi(string message)
13     {
14         Console.WriteLine(message);
15     }
16 }
```

Kuvio 8. Delegaatti mukailtu (“Using Delegates (C# Programming Guide)” 2022)

C#:ssa takaisinkutsu toteutetaan lambda lausekkeella ja delegate -metodilla. Lambda lausekkeella luodaan nimetön funktio, joka ottaa ensin parametrit ja sen jälkeen funktion lausekkeet, esimerkiksi $(a,b) => a+b$. Tässä esimerkissä a ja b ovat parametreja ja a+b on lauseke. Tämän lausekkeen voi sisällyttää mihin tahansa funktion kutsuun ja näin se pystytään antamaan funktion argumenttina toiselle funktiolle (“Lambda expressions (C# reference)” 2022). Delegate on viittaus, johonkin tiettyyn funktioon, jolla on määrätyt parametrit. Kuviossa 8 näytetään, miten luodaan yksinkertainen delegaatti. Tässä esimerkissä ohjelma tulostaa tekstin, mitä *handler* delegaatin kutsumisen yhteydessä annetaan (“Delegates (C# Programming Guide)” 2022).

3.3 *Actor*, eli toimija

Actor model luotiin Hewitt, Bishop ja Steiger 1973 toimesta. *Actor* eli suomeksi toimija on matemaattinen malli samanaikaisuuden laskemiselle. Agha 1986 teki laajennuksia Hewittin esittämään toimija malliin. Tässä kappaleessa käydään läpi sen keskeisimmät asiat. Yhdellä toimijalla on vain muutama toiminto, se voi lähettää viestejä toisille toimijoille, jonka niin kutsutun *postiosoitteen* se tietää. Lisäksi toimija voi luoda uusia toimijoita toteuttamaan jotain uutta tehtävää ja se voi muuttaa käytöstään, eli miten se toimii seuravan viestin tullessa. Millään muulla tavalla toimijoita ei voi muokata. Tämä tarkoittaa sitä, että toimija voi vain muuttua, kun se prosessoi viestiä. Toimija käsittelee vain yhtä viestiä kerrallaan viestijonosta, aina vanhin viesti ensin. Täten ei siis erikseen tarvitse hallita vuoroja, koska viestien lukeminen yksi kerrallaan hoitaa jo sen.



Kuvio 9. Actor, eli toimija

Kuviossa 9 esitetään yksinkertaisesti toimijoiden välinen viestien kulku. Viestit saapuvat toimijan *postilaatikkoon*, josta ne käsitellään saapumisjärjestyksessä eli jonossa.

Actor eli toimijan toteuttamiseen C#:ssa on muun muassa kirjasto *The Distributed Application Runtime, eli Dapr*. Bernstein ym. 2014 loi erillisen toteutustavan toimijoiden luomisek-

si nimeltään *virtual actor pattern*. *Dapr*-kirjasto toteuttaa toimijat käyttämällä *virtual actor pattern* mallia. Bernstein selitti, että *virtual actor pattern* mallissa käsitellään toimijoita virtuaalisina. Toimijat ovat aina olemassa ja niitä ei luoda tai poisteta, ne aktivoituvat automaattisesti, kun niille tulee viesti. Jos tietty toimija ei ole aktivoituneena, niin luodaan uusi instanssi toimijasta verkkopalvelimelle vapaaseen paikkaan. Toimijat ovat käytettävissä koko ohjelmalle, koska ne luodaan verkko palvelimelle. Palvelimella toimijat sijoitetaan eri ryhmiin ja solmuihin, Toimijat voivat olla aktiivisia, kun niitä tarvitaan ja ne voivat lopettaa toimintansa, kun niitä ei tarvita. *Virtual actor pattern* käyttää *promise* mallia toimiakseen samanaikaisesti.

"Dapr for .NET developers" 2022 kertoo tapauksista, missä on hyvä käyttää *Dapr* toimijoita. Niitä on muun muassa hyvä käyttää silloin, kun on riski *kilpajuoksutilanne tai lukkiutuma* toteutumiseen. Lisäksi *Dapr* toimijat ovat hyviä, kun ongelma voidaan jakaa pieniin riippumattomiin osiin. Toisaalta ongelma toimija mallissa on muun muassa se, jos moni asiakasohjelma pyytää samaan aikaan samaa toimijaa, tällöin se hidastaa ohjelman toimintaa, koska toimijat käyvät viestejä läpi jonossa yksi kerrallaan.

3.4 Reaktiivinen ohjelmointi

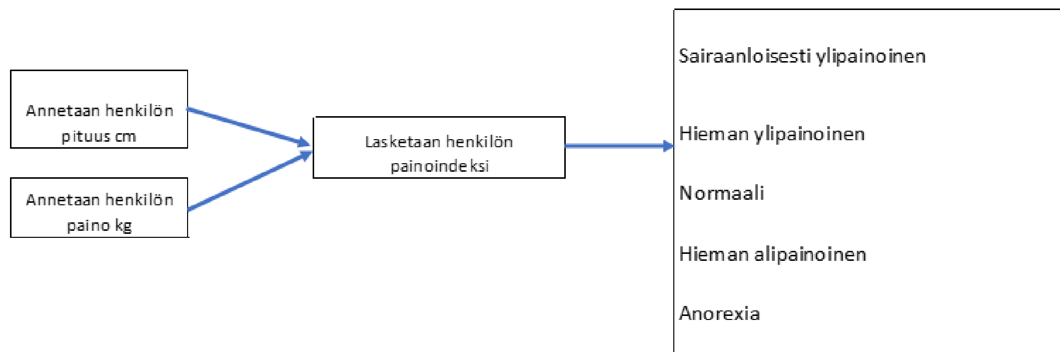
Reaktiivinen ohjelmointi on asynkroninen ohjelmointiparadigma, jonka tarkoituksena on jatkuva automaattinen reagointi muutoksiin ohjelmassa ja niiden lähetys eteenpäin muille ohjelmille, jotka niitä tarvitsevat (Elliott ja Hudak 1997). Esimerkiksi käyttäjän muokatessa arvoja ohjelmassa, nämä muutokset lähetetään saman tien eteenpäin ja niiden mukaan muokataan arvoja ohjelmassa. Toinen yksikertainen esimerkki olisi vaikka automaattisessa laskennassa. On neljä attribuuttia a, b, c ja d. C riippuu a:sta ja b:stä. Ja d riippuu taas c:stä. Esim $a = 5$, $b = 3$, $c = a - b$ ja $d = c + 4$. Tässä esimerkissä c riippuu a:sta ja b:stä ja d riippuu vielä c:stä. Näiden arvojen muuttuessa tapahtuu automaattinen päivitys muihin arvoihin, esimerkiksi taulukkolaskenta on reaktiivisen ohjelmoinnin hyvä käyttökohde.

Elliott ja Hudak 1997 toteaa, että reaktiivinen ohjelmointi on deklaratiivinen ohjelmointikieli. Siinä on monia erilaisia data tyyppejä, jotka mahdollistavat deklaratiivisen ohjelmoinnin toteuttamisen. Ohjelmoijan tarvitsee siis vain kertoa mitä haluaa ohjelmassa tapahtuvan ja

ohjelmointikieli tai kirjasto hoitaa loput .

Singh 2018 kertoo, että reaktiivisessa ohjelmoinnissa olennaisen tärkeitä käsitteitä ovat tapahtuma (event) ja tapahtumavirta (event stream). Tapahtumavirta kuvaa listaa tapahtumia ajassa ja ne voivat sisältää joko dataa tai olla arvoja jostain määrätystä joukosta. Tapahtumavirtaa voidaan suodattaa, vaikka filter -metodilla tai muokata vaikka map -metodilla. Näiden metodien käytöllä luodaan uusi tapahtumavirta, jota käsitellään, missä otetaan mukaan vain seulotut tapahtumat. Ehto voisi esimerkiksi ottaa huomioon vain ne arvot, jotka ovat pienimpiä kuin 5.

Singh 2018 selittää asian toisellakin tavalla. Hän kertoo, että tapahtumavirtojen läpi kulkee luotujen sääntöjen mukainen data, tätä yksittäistä dataa, esimerkiksi näppäinpainallus, kutsutaan tapahtumaksi. Tapahtuma ja sen data siirtyvät luotujen sääntöjen mukaisesti toiseen tapahtumavirtaan. Tapahtumavirrat, säännöt ja luodut riippuvuudet eivät muutu, vain data ja lopputulos muuttuu.



Kuvio 10. Tapahtumavirta

Kuviossa 10 esitetään tapahtumavirta henkilön pituuden ja painon suhteesta eli paino indeksista, joka taas kuvaa henkilön painoluokkaa. Kun henkilölle asetetaan pituus ja paino, tapahtumavirrat lähettävät tapahtumaviestit kuuntelijoille, jotka tekevät laskutoimitukseen. Ne laskevat painoindeksin. Painoindeksin arvon asettaminen tapahtuu uudessa tapahtumavirrassa, josta se lähettää tapahtumaviestin edelleen sen kuuntelialle, joka esittää henkilön painoluokan arvion.

Reaktiivisessa ohjelmoinnissa tavoitellaan sitä, että kehittäjän tarvitsee vain kertoa ohjelmal-

le mitä pitäisi saada aikaan ja jättää yksityiskohdat ohjelmointikielen hoidettavaksi.

“Reactive Extensions” 2011 mukaan C#:ssa reaktiivinen ohjelmointi toteutetaan muun muassa .Net-kirjaston *reactive extensions System.Reactive.Linq* kautta. Kirjastossa on muun muassa *IObservable* ja *IObservable.subscribe*. Niillä voidaan toteuttaa perustoiminnot reaktiiviseen ohjelmointiin. *IObservable* on tapahtuma ja siihen voidaan *subscribe* eli merkitä se kuunneltavaksi. Tällöin, kun *IObservable* saa tapahtuman, se ilmoittaa siitä kaikille kuuntelijoille, jotka on sen merkannut kuunneltavaksi .

Kuviossa 11 luodaan uusi *IObservable*. Sille luodaan tapahtumavirta, jossa yhden sekunnin välein tapahtuu tapahtuma. *Take(5)* luo tapahtumavirrasta vielä uuden tapahtumavirran, jossa on vain viisi ensimäistä tapahtumaa. *obj.Subsribe* ottaa luodun *IObservable* tarkasteluun ja aina kun siinä tapahtuu muutos se toteuttaa tehtävänsä. Tässä tapauksessa ohjelma tulostaa numerot 0–4.

```
1 static async Task Main(string[] args)
2 {
3     IObservable<long> obj =
4     Observable.Interval(TimeSpan.FromSeconds(1)).Take(5);
5
6     obj.Subscribe(x => Console.WriteLine(x));
7
8     Console.ReadKey();
9 }
```

Kuvio 11. Reaktiivinen laajennos mukailtu (“ASP.NET Core Series 05: Don’t block your code, be reactive” 2022)

4 Analyysi

Samanaikaisuuden toteutukseen C#:lla on esitetty tässä tutkielmassa eri metodeja ja nyt niitä analysoidaan ja vertaillaan eri käyttötarkoituksissa.

Cleary 2019 mukaan *parallel.invoke* -metodi on tehokas silloin, kun halutaan tehdä pieniä laskutoimituksia tai tehtäviä samanaikaisesti. Tämä metodi toteuttaa toisistaan riippumattomien tehtävien jakamisen useaksi säikeeksi ja kokoaa ne taas yhteen, kun kaikki tehtävät ovat valmiita. Hän toteaa, että *parallel* -metodit ovat hyviä asiakasrajapinnan toteuttamisessa. Toisaalta Cleary varoittaa siitä, että *parallel.invoke* -metodin ongelma on se, jos samanaikaisesti suoritettavat tehtävät ovat liian isoja tai liian pieniä, metodi ei välttämättä osaa jakaa suorituskykyä optimaalisesti ja siten ei saada ohjelman tehokkuutta nostettua tavoitetulla tavalla.

Cleary 2019, luku 1 kertoo, että asynkronisella ohjelmoinnilla on ainakin kaksi etua, se tarjoaa graafisille käyttöliittymille reaktiivisuutta ja palvelin puolelle se tarjoaa hyvää skaalausmahdollisuutta. Eli voidaan ottaa monia pyyntöjä vastaan samaan aikaan, koska pääsäie vapautetaan heti kun pyyntö on käsitelty ja siirretty toiselle säikeelle .

Asynkronisuuden lupaus mallia C#:ssa nykyisin toteuttaa muun muassa *(*async ja await) -metodi. Tätä metodia on helppo käyttää, koska normaalin synkroniseen ohjelmointimalliin tarvitsee vain lisätä *(*async ja await) avainsanat (Cleary 2019). Nämä avainsanat muokkaavat synkronisesta tehtävästä asynkronisen tehtävän. Tällöin ohjelma voi jatkaa muuta toimintaa samanaikaisesti asynkronisen tehtävän tapahtuessa toisella säikeellä.

C#:ssa muun muassa *Dapr*-kirjasto toteuttaa toimijat, jotka ratkaisevat osan samanaikaisuuden ongelmista, kuten esimerkiksi *kilpajuoksutilanne* ja *lukkiutuma*. Toimija mallissa viestejä käsitellään jonossa, aina vanhin viesti ensin, ja täten ei tarvitse huolehtia edellä mainituista ongelmista (Agha 1986). Toisaalta yhtenä ongelmana tässä mallissa on se, jos yhtä toimijaa ylikuormitetaan liian monella samanaikaisella viestillä, koska viestit käsitellään jonossa, tällöin koko ohjelman toiminta voi hidastua .

Cleary 2019 toteaa *reactive extensions* olevan vaikeampi oppia, kuin muut asynkroniset ta-

pahtumakäsittelijämenetelmät. Lisäksi Cleary sanoo, että reaktiivista ohjelmaa on vaikeampi myös ylläpitää, mutta toisaalta, kun reaktiivisen ohjelmoinnin hallitsee, on se hyvin tehokas työväline tekemään reagoivia ohjelmia, esimerkiksi käyttöliittymiä. Lisäksi Elliott ja Hudak 1997 toteavat, koska reaktiivinen ohjelmointi on deklaratiivinen ohjelmointitapa, se lyhentää huomattavasti ohjelmakoodin pituutta.

Edellä esitettyjen perusteella voidaan todeta, että C#:in eri käyttötarkoituksiin on löydettävissä erilaisia ratkaisuja. Samanaikaiset *Parallel* -metodit toimivat hyvin muun muassa rinnakkaisissa taulukkolaskentatehtävissä tai monien pienten riippumattomien tehtävien toteutuksessa. Kun taas asynkroniset *async ja await* -metodit toimivat hyvin, kun tarvitsee tehdä esimerkiksi reagoivia käyttöliittymiä tai palvelimia. *Reactive extensions* on kokonaisratkaisuna todennäköisesti tehokkain tapa toteuttaa reagoivat käyttöliittymät ja palvelimet. *Dapr*-kirjasto on hyvä tapa toteuttaa samanaikaisuutta toimijoilla, koska sen toimintamalli estää monet samanaikaisuuden ongelmat.

5 Yhteenveto

Tässä tutkielmassa käsiteltiin samanaikaisuuden toteuttamista C#:lla, kuten *Parallel* -metodit, *async ja await* -metodit, *Dapr*-kirjaston toimijat sekä *reactive extensions*-kirjasto. Tutkielmassa luotiin lukijalle yleiskäsitys asiasta tekstein, kuvioin ja koodiesimerkein. Tutkielmaa tehtäessä tutustuttiin alan kirjallisuuteen, konferenssimateriaaleihin ja ohjelmaoppaisiin.

Peruskäsitteistä käytiin läpi samanaikaisuus, rinnakkaisuus ja asynkronisuus. Samanaikaisuus on monen prosessin suorittamisesta yhtä aikaa ja rinnakkaisuus on erityiskäsite samanaikaisuudelle, jossa prosessit toteutuvat aidosti samaan aikaan. Asynkronisuudella taas tarkoitetaan sitä, että yksi tehtävä alkaa, mutta ohjelma ei odota sen valmistumista ennen ohjelmakoodin jatkamista.

Tutkielmassa päädyttiin tuloksiin, jossa samanaikaisuuden toteuttamiseen C#:lla ei ole vain yhtä ainoata ratkaisua, vaan voidaan löytää eri käyttötapauksiin erilaisia soveltuvia toteutustapoja. Todettiin, että *Parallel* -metodit toimivat hyvin taulukkolaskenta tyyppisissä tehtävissä tai esimerkiksi asiakasrajapinnassa toteutettaviin tehtäviin. Asynkroniset metodit, kuten *async ja await* ja *Reactive extensions* ovat hyviä toteuttamaan esimerkiksi reaktiivisia käyttöliittymiä. Toimijat taas ovat yksinkertaisempi tapa toteuttaa samanaikaisuutta, koska ne estävät muutaman normaalin samanaikaisuuden ongelman, kuten *kilpajuoksutilanne* ja *lukkiutuma*.

Kaikkia näitä samanaikaisuuden malleja voidaan käyttää samassa ohjelmassa, aina etsien parasta mahdollista tapaa tietyn ongelman ratkaisuun ohjelmassa, eli ei ole vain yhtä ainoaa ratkaisua toteuttaa samanaikaisuutta C#:ssa.

Lähteet

Agha, Gul. 1986. *Actors: a model of concurrent computation in distributed systems.* MIT press.

"ASP.NET Core Series 05: Don't block your code, be reactive". 2022. Viitattu 19. marraskuuta 2022. https://www.gokhan-gokalp.com/en/asp-net-core-series-05-dont-block-your-code-be-reactive/.

"Async and Await in C#". 2022. Viitattu 16. marraskuuta 2022. https://origin.geeksforgeeks.org/async-and-await-in-c-sharp/.

"Asynchronous programming". 2022. Viitattu 19. marraskuuta 2022. https://learn.microsoft.com/en-us/dotnet/csharp/async.

Bernstein, Phil, Sergey Bykov, Alan Geller, Gabriel Kliot ja Jorgen Thelin. 2014. *Orleans: Distributed Virtual Actors for Programmability and Scalability.* Tekninen raportti MSR-TR-2014-41. Maaliskuu. https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/.

Cleary, Stephen. 2019. *Concurrency in C# Cookbook: Asynchronous, Parallel, and Multithreaded Programming.* O'Reilly Media.

Conway, Melvin E. 1963. "A Multiprocessor System Design". Teoksessa *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference,* 139–146. AFIPS '63 (Fall). Las Vegas, Nevada: Association for Computing Machinery. ISBN: 9781450378833. https://doi.org/10.1145/1463822.1463838. https://doi.org/10.1145/1463822.1463838.

"Dapr for .NET developers". 2022. Viitattu 16. marraskuuta 2022. https://learn.microsoft.com/en-us/dotnet/architecture/dapr-for-net-developers/actors.

Davies, Alex. 2012. *Async in C# 5.0.* "O'Reilly Media, Inc.".

"Delegates (C# Programming Guide)". 2022. Viitattu 19. marraskuuta 2022. https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/.

Elliott, Conal, ja Paul Hudak. 1997. "Functional Reactive Animation". *SIGPLAN Not.* (New York, NY, USA) 32, numero 8 (elokuu): 263–273. ISSN: 0362-1340. https://doi.org/10.1145/258949.258973. https://doi.org/10.1145/258949.258973.

Fiedor, Jan, Bohuslav Křena, Zdeněk Letko ja Tomáš Vojnar. 2011. "A uniform classification of common concurrency errors". Teoksessa *International Conference on Computer Aided Systems Theory,* 519–526. Springer.

Friedman ja Wise. 1978. "Aspects of Applicative Programming for Parallel Processing". *IEEE Transactions on Computers* C-27 (4): 289–296. https://doi.org/10.1109/TC.1978.1675100.

Gallaba, Keheliya, Ali Mesbah ja Ivan Beschastnikh. 2015. "Don't Call Us, We'll Call You: Characterizing Callbacks in Javascript". Teoksessa *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM),* 1–10. https://doi.org/10.1109/ESEM.2015.7321196.

Goranova, Mariana, Elena Kalcheva-Yovkova ja Stanimir Penkov. 2015. "Task-based Asynchronous Pattern with async and await". Teoksessa *International Scientific Conference Computer Science,* 150.

Hewitt, Carl, Peter Bishop ja Richard Steiger. 1973. "Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence". Teoksessa *Advance Papers of the Conference,* 3:235. Stanford Research Institute Menlo Park, CA.

"Lambda expressions (C# reference)". 2022. Viitattu 19. marraskuuta 2022. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions.

Lewis, Bil, ja Daniel J Berg. 1995. *Threads primer: a guide to multithreaded programming.* Prentice Hall Press.

Liskov, B., ja L. Shrira. 1988. "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems". *SIGPLAN Not.* (New York, NY, USA) 23, numero 7 (kesäkuu): 260–267. ISSN: 0362-1340. https://doi.org/10.1145/960116.54016. https://doi.org/10.1145/960116.54016.

Madsen, Magnus, Ondřej Lhoták ja Frank Tip. 2017a. "A Model for Reasoning about JavaScript Promises". *Proc. ACM Program. Lang.* (New York, NY, USA) 1, numero OOPSLA (elokuu). https://doi.org/10.1145/3133910. https://doi.org/10.1145/3133910.

———. 2017b. "A Model for Reasoning about JavaScript Promises". *Proc. ACM Program. Lang.* (New York, NY, USA) 1, numero OOPSLA (lokakuu). https://doi.org/10.1145/3133910. https://doi.org/10.1145/3133910.

"Parallel.Invoke Method". 2022. Viitattu 19. marraskuuta 2022. https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.invoke?view=net-7.0.

"Reactive Extensions". 2011. Viitattu 17. marraskuuta 2022. https://learn.microsoft.com/en-us/previous-versions/dotnet/reactive-extensions/hh242985(v=vs.103)?redirectedfrom=MSDN.

Singh, Navdeep. 2018. *Reactive Programming with Swift 4: Build Asynchronous Reactive Applications with Easy-To-maintain and Clean Code Using Rxswift and Xcode 9.* 1. painos. Packt Publishing, Limited.

Syme, Don, Tomas Petricek ja Dmitry Lomov. 2011. "The F# Asynchronous Programming Model". Teoksessa *Practical Aspects of Declarative Languages,* toimittanut Ricardo Rocha ja John Launchbury, 175–189. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-642-18378-2.

"Using Delegates (C# Programming Guide)". 2022. Viitattu 19. marraskuuta 2022. https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/using-delegates.

"Using threads and threading". 2022. Viitattu 14. marraskuuta 2022. https://learn.microsoft.com/en-us/dotnet/standard/threading/using-threads-and-threading.