Kandidaatintyö
2025

Valentin Rainio

Aalto-yliopisto
Sähkötekniikan korkeakoulu
Sähkötekniikan kandidaattiohjelma

Valentin Rainio

Rinnakkaisuus verkko-ohjelmoinnissa

Kandidaatintyö

9.5.2025

Työn ohjaaja:

Yliopistonlehtori Pasi Sarolahti

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU
PL 13000
00076 AALTO

TEKNIIKAN KANDIDAATINTYÖN TIIVISTELMÄ

Tekijä: Valentin Rainio

Työn nimi: Rinnakkaisuus verkko-ohjelmoinnissa

Koulutusohjelma: Sähkötekniikan kandidaattiohjelma

Päiväys: 5.5.2025 Sivumäärä: 26

Vastuuopettaja: yliopistonlehtori Markus Turunen

Ohjaaja: yliopistonlehtori Pasi Sarolahti

Kieli: suomi

Rinnakkaisuus on verkko-ohjelmoinnin keskeinen ominaispiirre, joka mahdollistaa useamman verkkoyhteyden prosessoinnin samanaikaisesti. Nettipalvelimissa ja muissa laajemmissa useita verkkoyhteyksiä hallitsevissa ohjelmissa rinnakkaisuuden tehokkuudella ja skaalautuvuudella on keskeinen rooli ohjelman suorituskyvyn kannalta. Erityisen tärkeitä ovat käyttöjärjestelmien tarjoamat rajapinnat, jotka mahdollistavat erinäisiä tapoja rinnakkaistaa verkko-ohjelmistoja. Tässä kandidaatintyössä tarkastellaan erityisesti IO-multipleksausrajapintoja Linux ja Windows käyttöjärjestelmissä, ja tutkitaan, miten ne mahdollistavat tehokkaan verkko-ohjelmoinnin. Erityisesti keskitytään siihen, miten nämä verkkorajapinnat ovat ajan myötä kehittyneet ja minkälaisia optimointeja niihin on lisätty.

Kandidaatintyössä esitetään myös yleiskatsaus rinnakkaisajosta, sekä tarkastellaan, miten rinnakkaisajo on vaikuttanut verkko-ohjelmointiin asynkronisen ohjelmoinnin merkeissä. Työn loppupuolella tätä tarkastellaan kahden asynkronisen ohjelmointiympäristön avulla: Node.js ja Tokio. Kirjastoissa tutkitaan, miten asynkroninen ohjelmointi on niissä toteutettu alhaisella tasolla ja miten niissä hyödynnetään taustalla IO-multipleksausta.

Työssä keskitytään nimenomaan rinnakkaisuutta mahdollistaviin rajapintoihin, joten muut rinnakkaisuudessa huomioitavat asiat kuten synkronointi ja muistilukkojen käyttö jäävät vähemmälle huomiolle.

Avainsanat: asynkroninen verkko-ohjelmointi, IO-multipleksaus, rinnakkaisajo

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

BACHELOR'S THESIS
ABSTRACT

Author: Valentin Rainio

Title of the thesis: Concurrency in web programming

Study programme: Electrical engineering

Date: May 5th, 2025 Page count: 26

Responsible teacher: University lecturer Markus Turunen

Instructor: University lecturer Pasi Sarolahti

Language: Finnish

Concurrency is a key feature of network programming, allowing multiple network connections to be processed simultaneously. For web servers and other large-scale multi-network applications, the efficiency and scalability of concurrency play a key role in program performance. Of particular importance are the interfaces provided by operating systems, which allow a variety of ways to parallelize network software. This thesis examines IO multiplexing interfaces in Linux and Windows operating systems and examines how they enable efficient network programming. In particular, the focus will be on how these network interfaces have evolved and what optimizations have been added to them.

The thesis also provides an overview of concurrency and examines how concurrency has influenced network programming in the context of asynchronous programming. Towards the end of the thesis, this is explored using two programming platforms – Node.js and Tokio. The focus is on how asynchronous programming has been implemented in them at a low level and how they make use of IO multiplexing in the background.

The work focuses specifically on the interfaces that enable concurrency, so other issues that need to be considered in concurrent programming, such as synchronization and the use of memory locks, receive less attention.

Keywords: asynchronous web programming interfaces, IO multiplexing, concurrency

Sisällys

1 Johdanto **1**
2 Verkko-ohjelmoinnin yleispiirteitä **3**
2.1 Verkko-operaatiot ja BSD-pistokkeet 3
2.2 IO-operaatiot käyttöjärjestelmän näkökulmasta 4
3 Rinnakkaisuus verkko-ohjelmoinnissa **5**
3.1 Rinnakkaisuuden tärkeys 5
3.2 Rinnakkais-IO:n mallit 6
3.3 Rinnakkaisajon mallit 8
3.4 Rinnakkaisuuden haasteita ja hyvän ohjelmointirajapinnan meriittejä 12
4 Rinnakkais-IO:n käyttöjärjestelmätuki **13**
4.1 Linux-rajapinnat 14
4.2 Windows-rajapinnat 18
5 Asynkronisuus ohjelmointialustoissa **20**
5.1 JavaScript ja Node.js 20
5.2 Rust ja Tokio 22
6 Yhteenveto **25**
Lähdeluettelo **27**

1 Johdanto

Internetin kehittyessä alkoi samalla kehittyä myös verkko-ohjelmointi, joka monin tavoin eroaa perinteisistä ohjelmointimenetelmistä: perinteisessä ohjelmoinnissa tehtäviä voidaan suorittaa peräkkäisesti toinen toisensa jälkeen, kun taas verkko-ohjelmoinnissa joudutaan hoitamaan useaa tehtävää samanaikaisesti. Esimerkiksi chattipalvelimeen saattaa yhdistyä samanaikaisesti monta päätelaitetta, jolloin kunkin päätelaitteen yhteyksiä pitää pystyä prosessoimaan samanaikaisesti. Tehtävien suorittamiseen rinnakkaisesti on kehitetty useita eri tekniikoita, joista osa on suunniteltu erityisesti verkko-ohjelmointia ajatellen hyödyntämään siinä esiintyviä viiveitä.

Viive onkin yksi verkko-ohjelmoinnin tärkeimmistä ominaispiirteistä. Koska tietoliikenne tapahtuu verrattain hitaan verkkolinkin kautta, vastaanotettavaa dataa joudutaan usein odottamaan pitkäänkin. Odottamiseen kuluvan ajan hyödyntäminen muiden tehtävien prosessointiin tarjoaakin oivan mahdollisuuden optimoida verkko-ohjelman suorituskykyä. Hyödyntäminen riippuu kuitenkin käyttöjärjestelmien tuesta suorittaa useita verkko-operaatioita taustalla pysäyttämättä ohjelman ajoa. Tämä tapahtuu multipleksausrajapintojen kautta, jotka ovat viimeisten 30 vuoden aikana kehittyneet suorituskyvyltään ja skaalautuvuudeltaan vastaten internetohjelmien vaativuustason kasvuun.

Näitten rajapintojen käyttö itsessään on kuitenkin monimutkaista, joten moni ohjelmointiympäristö ja kieli tarjoaa korkeamman abstraktiotason asynkronisia työkaluja, joiden avulla tehokkaan rinnakkaisuutta hyödyntävän ohjelman kirjoittaminen olisi mahdollisimman yksinkertaista.

Tässä kandidaatintyössä tutkitaan rinnakkaisuuden eri menetelmiä. Työssä keskitytään erityisesti Linux- ja Windows-käyttöjärjestelmien tarjoamiin multipleksaus-rajapintoihin. Tutkimusnäkökulmana on erityisesti, miten nämä käyttöjärjestelmärajapinnat ovat kehittyneet ajan myötä muistinkäytön ja suorituskyvyn kannalta. Työssä vertaillaan eri ratkaisujen arkkitehtuuria ja käsitellään, miten nämä matalan tason rajapinnat toimivat perustana korkeamman tason ohjelmointialustoille ja kirjastorakenteille. Tarkasteluun otetaan esimerkeiksi Node.js- ja Rust (Tokio) -ympäristöt, joiden avulla havainnollistetaan, miten

käyttöjärjestelmätason rajapintojen kehitys näkyy käytännön verkkosovellusohjelmoinnissa. Työ rajoittuu IO-rajapintojen ja rinnakkaisuuden tekniseen kehitykseen, eikä käsittele syvällisemmin esimerkiksi tietoturvakysymyksiä tai verkkoprotokollia.

Työn alussa tarkastellaan verkko-ohjelmointiin liittyviä perusasioita (luku 2), minkä jälkeen siirrytään rinnakkaisuuden tarkastelemiseen yleisellä tasolla (luku 3). Luvussa 4 syvennytään Linux- ja Windows- käyttöjärjestelmien multipleksausrajapintojen tarkastelemiseen ja pohditaan näitten suorituskykyjä. Lopuksi luvussa 5 päästään tarkastelemaan käytännön verkkoalustoja kahden esimerkin tapauksessa ja osoitetaan, miten asynkroninen ohjelmointi on multipleksausrajapintojen kautta vaikuttanut verkko-ohjelmointiin.

2 Verkko-ohjelmoinnin yleispiirteitä

Nyky-yhteiskunnan informaatio- ja kommunikaatiokeskeisyys on nostanut verkko-ohjelmoinnin tärkeään asemaan. Moni palvelu, ohjelma ja jopa älykäs kotitalouslaite on riippuvainen verkko-ohjelmoinnin toimivasta ratkaisusta. Sovelluksesta riippuen verkko-ohjelmointi voi olla hyvinkin erilaista, mutta keskiössä on kuitenkin aina yhteyksien muodostaminen verkon välityksellä sekä datan eli informaation välittäminen näitten verkkoyhteyksien kautta.

2.1 Verkko-operaatiot ja BSD-pistokkeet

Verkko-ohjelmointi on osa laajempaa IO-ohjelmointi-käsitettä. Termi on lyhenne sanoista Input/Output ja tarkoittaa mitä tahansa ohjelmointia, jonka avulla ohjelma pääsee käsiksi itsensä ulkopuoliseen maailmaan muun muassa lukemaan tiedostoja, piirtämään näytölle tai havaitsemaan hiiriliikettä. Verkko-ohjelmoinnissa ulkopuolinen maailma tarkoittaa muita verkon kautta yhdistettyjä laitteita.

Koska ohjelma toimii käyttöjärjestelmän hallitsemassa ympäristössä, IO-operaatiot ovat täysin riippuvaisia käyttöjärjestelmän tarjoamista komennoista ulkopuolisiin resursseihin käsiksi pääsemisessä. Käyttöjärjestelmissä onkin määritelty tarkat säännöt ja komennot, miten tämä tapahtuu, ja tätä kokonaisuutta kutsutaan rajapinnaksi. IO-rajapinnan yksittäisiä komentoja kutsutaan IO-operaatioiksi. Verkkorajapinnassa monelle verkkokomennolle, eli verkko-ohjelmointiin liittyvät IO-operaatiolle, löytyy läheinen vastaavuus verkossa tapahtuvalle asialle: yhteyden muodostamis (connect) -komennolla muodostetaan etäyhteys, luku- (read-) ja kirjoitus- (write-) komennoilla vastaavaavasti lähetetään ja vastaanotetaan dataa etäyhteyslinkin avulla. [1]

Tärkeä yleisesti käytetty verkkorajapinta on BSD-sokettirajapinta, joka toteutettiin alun perin BSD-käyttöjärjestelmässä, mutta joka sittemmin otettiin käyttöön myös muissa järjestelmissä. BSD-sokettirajapinta on keskeinen osa myös rinnakkaisuutta mahdollistavissa IO-multipleksausrajapinnoissa, joten sen ymmärtäminen on kriittistä tehokkaan verkkosovelluksen kehittämiseksi. BSD-soketirajapinnassa verkkoyhteyksiä käsitellään niin kutsuttujen pistokkeiden (socket) avulla. Pistoke on yksittäistä verkkoyhteyttä kuvastava yksiselitteinen tunniste, jota käytetään aina yhtenä parametrina verkko-operaatioiden yhteydessä. Tällöin käyttöjärjestelmän rajapinta

tietää, mistä verkkoyhteydestä on kyse, kun suoritetaan, esimerkiksi luku tai kirjoitusoperaatio. [2]

BSD-sokettirajapinta on perusmuodossaan blokkaava eli jokainen IO-operaatio pysäyttää koodin ajon, kunnes operaatio saatetaan päätökseen. Esimerkiksi lukuoperaatio palaa vasta kun data saapuu etäyhteyden kautta tietokoneelle. Pistokkeet voi kuitenkin säätää blokkaamattomaan muotoon, jolloin IO-operaatiot eivät enää pysäytäkään koodia vaan palaavat heti mahdollistaen rinnakkaisuuden toteuttamisen IO-multipleksauksen avulla. [2] Tähän syvennytään luvuissa kolme ja neljä.

2.2 IO-operaatiot käyttöjärjestelmän näkökulmasta

IO-operaatiot ovat käyttöjärjestelmän tarjoamia komentoja ja ne yleensä toteutetaan järjestelmäkutsuina. Järjestelmäkutsut eroavat olennaisesti tavallisista funktiokutsuista, ja ne ovat paljon hitaampia. Järjestelmäkutsuissa ohjelmakoodin suoritus pysähtyy ja prosessointi jatkuu käyttöjärjestelmäytimen puolella.

Käyttöjärjestelmäydin on käyttöjärjestelmän ydintehtävistä vastaava komponentti, joka on aina toiminnassa. Käyttöjärjestelmäydin on turvallisuussyistä täysin eristetty ohjelmamuistista, eikä ohjelma pääse suoraan käsiksi käyttöjärjestelmäytimen muistiin. Kun IO-operaation kautta halutaan välittää muistia (esim. lukuoperaation avulla) tämä muisti joudutaan kopioimaan ohjelmamuistin ja ydinmuistin välillä, mikä vie aikaa. [3] Lisäksi muistieristyksen vuoksi prosessorin välimuistin lokalisointi kärsii, koska välimuisti täyttyy ytimen muistilla, jolloin ohjelman suoritus hidastuu IO-operaation jälkeen. Viimein kontekstinvaihto (context switch) eli siirtymä ohjelma- ja ydinkoodin suorittamisen välillä on itsestään hidasta, sillä prosessori joutuu tallentamaan ohjelmakoodin tilan ja muuttamaan prosessorin turvallisuustilaa ennen ydinkoodiin siirtymistä. [4]

Koska verkko-ohjelmointi on hyvin IO-painotteista, verkko-operaatioiden nopeudella on olennainen merkitys ohjelman suorituskyvyn kannalta. Käyttöjärjestelmän verkkorajapintoja onkin optimoitu nimenomaan järjestelmäkutsujen osalta useaan otteeseen muun muassa vähentämällä muistin kopioimista sekä minimoimalla verkkoyhteyksien hallitsemiseksi tarvittavien järjestelmäkutsujen määrää. Linuxissa ja Windowsissa tehtyjä optimointeja avataan lisää luvussa 4.

3 Rinnakkaisuus verkko-ohjelmoinnissa

Rinnakkaisuus on verkko-ohjelmoinnin kenties tärkein peruspilari. Sen toteutukseen on olemassa kuitenkin erilaisia teoreettisia malleja, joita käyttöjärjestelmät ja ohjelmointiympäristöt sekä -kirjastot hyödyntävät. Alhaisen abstraktiotason rinnakkais-IO:n malleissa rinnakkaistaminen tapahtuu yksittäisten verkko-operaatioiden tasolla, kun taas korkeamman abstraktiotason rinnakkaisajon malleissa pyritään samanaikaistamaan kokonaisten tehtävien suorittaminen. Tässä luvussa avataan vielä hieman rinnakkaisuuden tärkeyttä, syvennytään rinnakkais-IO:n ja rinnakkaisajon malleihin ja lopuksi pohditaan rinnakkaisuuden haasteita.

3.1 Rinnakkaisuuden tärkeys

Rinnakkaisuuden voidaan nähdä hyödyntävän verkko-ohjelmia ainakin kahdella eri tavalla: Ensinäkin se parantaa ohjelmien suorituskykyä, sillä ohjelma pystyy hyödyntämään verkko-operaatiossa tapahtuvaa odotusta muiden tehtävien suorittamiseen. Lisäksi rinnakkaistaminen tekee ohjelmasta vakaamman, sillä yhden verkkoyhteyden hitaus ei viivytä muiden verkkoyhteyksien prosessointeja, kun verkkoyhteyksiä hoidetaan toisistaan riippumattomasti.

Verkkopalvelujen yleistyminen vuosien saatossa on huomattavasti nostanut rinnakkaisuuden vaativuustasoa etenkin skaalautuvuuden osalta. Jo vuonna 1999 tunnustettu ohjelmistoinsinööri Dan Kegel [5] määritteli C10k kriteerin, josta on sittemmin tullut tietynlainen hyvän nettipalvelimen suorituskyvyn kriteeri. Kriteerin mukaan palvelinohjelman on kyettävä ylläpitämään kymmentä tuhatta verkkoyhteyttä samanaikaisesti. Kuitenkin jo vuonna 2013 verkkopalveluja tarjoava yritys MigratoryData [6] julisti ylittäneensä 10 miljoonan samanaikaisen WebSocket-tyyppisen verkkoyhteyden ylläpidon yhdellä palvelintietokoneella. Vaikka tämän asteinen rinnakkaisuus toteutetaan yleisemmin kuormantasauksen avulla useamman palvelinkoneen kesken, kasvavat rinnakkaisuuden paineet ovat johtaneet IO-rajapintojen tehokkuuden paranemiseen.

Toki ohjelmasta riippuen rinnakkaisajon tarve voi olla hyvinkin erilainen: esimerkiksi konsolipohjaiset asiakasohjelmistot voivat pärjätä käytännössä ilman mitään IO:n rinnakkaistamista, kun tavoitteena yhdistyä vain yhteen palvelimeen samanaikaisesti

eikä ohjelmalla ole interaktiivista käyttöliittymää. Muissa tapauksissa samanaikaisten yhteyksien määrän odotusarvolla ja ohjelman laskennallisella monimutkaisuudella on suuri vaikutus sopivan rinnakkaisajomallin valinnassa: Rinnakkaisuus voidaan toteuttaa käyttämällä yksinkertaisesti useita käyttöjärjestelmän säikeitä tai se voidaan toteuttaa yhden säikeen sisällä käyttäen käyttöjärjestelmän tarjoamia rinnakkaisen IO:n rajapintoja ja asynkronista ohjelmointia. Vaativimmissa tapauksissa joudutaan hyödyntämään molempia.

3.2 Rinnakkais-IO:n mallit

Alhaisella tasolla verkko-IO voidaan jakaa muutamaan perusmalliin, jotka ovat synkroninen IO, synkroninen IO säikeiden kanssa, tapahtumapohjainen blokkaamaton IO, sekä asynkroninen IO.

Kaikkein yksinkertaisin ohjelmointimalli hyödyntää synkronisia tai blokkaavia IO-rajapintoja. Tällöin jokainen IO-operaatio pysäyttää ohjelman ajon, kunnes operaatio suoriutuu loppuun asti. Esimerkiksi etäyhteyslinkistä lukeminen kestää niin kauan, että data on luettavissa eli saapuu koneelle. Synkroninen IO ei siis mahdollista verkkoyhteyksien rinnakkaistamista, sillä yksittäinen IO-operaatio voi pysäyttää koko ohjelman kulun. Synkronista ohjelmointia voidaan kuitenkin yhdistää säikeiden kanssa. [6] Säikeet ovat käyttöjärjestelmän tarjoama ominaisuus, joka mahdollistaa useamman aliohjelman ajon samanaikaisesti ja toisistaan riippumattomasti. Käyttöjärjestelmä vaihtelee automaattisesti eri säikeiden ajon välillä tarjoten kullekin säikeelle tasavertaisen osuuden prosessointiresursseista. Ohjelma pystyy tällöin käynnistämään yhden IO-operaation kutakin säiettä kohden, ja jos jokaiselle verkkoyhteydelle luodaan oma säie, blokkaavan IO:n käyttö ei ole ongelma. [7]

Kuitenkin samanaikaisten verkkoyhteyksien kasvaessa säikeiden käyttö on ongelmallista: Kukin säie vie huomattavasti muistiresursseja: käyttöjärjestelmäydin allokoi kullekin säikeelle oman säiekontekstin (thread context), minkä lisäksi kukin säie tarvitsee itselleen ainakin oman pinomuistinsa (stack memory). Näin ollen pelkän verkkoyhteyksien hallinnasta johtuvan muistinkasvun lisäksi säikeiden käytöstä johtuva muistinkuormitus on huomattavaa, mikä vaikeuttaa esimerkiksi C10k kriteerin saavuttamisen, kun tietokoneen kokonaismuisti on rajallinen. [7] Lisäksi säikeiden

välillä vaihtaminen eli kontekstinvaihto (tästä lisää luvussa 2) on hidastaa ohjelmanajoa.

Tällöin ratkaisuna voidaan nähdä blokkaamaton IO, joka mahdollistaa verkko-IO:n rinnakkaistamisen yhden säikeen sisällä. Tällöin säikeiden muistikuormasta päästään eroon. Yleisesti puhuttu vertailukohde on nginx- ja Apache-palvelinohjelmistojen välinen arkkitehtuuri- ja suorituskykyero: nginx, joka hyödyntää blokkaamatonta IO:ta suoriutuu monessa vertailussa huomattavasti paremmin kuin blokkaavaa IO:ta ja monisäikeisyyttä hyödyntävä Apache-palvelin [8].

Blokkaamattomassa IO:ssa verkko-operaatiot nimensä mukaisesti eivät pysäytä ohjelman kulkua. Blokkaamattomassa IO:ssa verkko-operaatio onnistuu vain, jos se on toteutettavissa heti: esimerkiksi lukuoperaatio onnistuu, jos data on jo saapunut luettavaksi. Jos taas operaatio vaatii odottamista, IO-operaatio palauttaa virheen. IO on tapahtumapohjaista (event based), mikä tarkoittaa, että ohjelma voi tutkia yksittäisen verkkoyhteyden luettavuus- ja kirjoitettavuustilaa sekä odottaa näitten tilojen muutosta. Rinnakkaistamisen näkökulmasta tapahtumapohjaisen IO:n vahva puoli on se, että IO-rajapinnat tarjoavat tavan odottaa useamman verkkoyhteyden tilanmuutosta samanaikaisesti. Tällöin ohjelma kykenee reagoimaan heti, kun yksikin verkkoyhteys on luettavassa tai kirjoitettavassa tilassa. Useimmiten tämä johtaa tapahtumasilmukan näköiseen rakenteeseen, jossa ohjelma odottaa pääsilmukassa jatkuvasti eri verkkoyhteyksien tilojen muuttumista ja reagoi näihin muutoksiin edistämällä aina sopivan verkkoyhteyden prosessointia, kun sen tila muuttuu luettavaksi tai kirjoitettavaksi. [1]

Tapahtumapohjaista IO:ta kutsutaan usein myös reaktiiviseksi IO-malliksi, koska siinä reagoidaan erilaisiin tapahtumiin ja ajetaan blokkaamattomia IO-operaatioita näitten tapahtumien johdosta. Blokkaamaton IO voidaan kuitenkin toteuttaa käyttäen myös proaktiivista ohjelmointimallia. Siinäkään IO-operaatiot eivät pysäytä ohjelmanajoa, mutta jos operaatio ei ole suoritettavissa heti, operaatiokutsu ei epäonnistu vaan aloittaa operaation suorittamisen taustalla. Odotusfunktion avulla monitoroidaan IO-operaation valmistumista. Proaktiivinen malli mukailee läheisesti korkeamman tason asynkronista rinnakkaisajon mallia, minkä vuoksi sitä kutsutaan usein myös asynkronisen IO:n malliksi. Proaktiivista mallia on luonnehdittu reaktiivista mallia edistyneemmäksi, sillä se on konseptuaalisesti yksinkertaisempi eikä vaadi erillistä operaatiokutsua

odotusfunktion lisäksi IO-operaation saattamiseksi loppuun. Lisäksi proaktiivinen malli skaalautuu helpommin monisäikeisiin ohjelmiin. [9]

Monisäikeisyys ja blokkaamaton IO eivät olekaan toisiaan poissulkevia ratkaisuja. Etenkin moniytimisen prosessorin tapauksessa pelkän blokkaamattoman IO:n käyttö ei johda kaiken prosessointitehon käyttöön: koska kaikki prosessointi suoritetaan yhdellä säikeellä, ohjelma pystyy hyödyntämään vain yhtä ydintä. Tämän vuoksi vaativissa verkkopalvelinohjelmissa voidaan hyödyntää sekä monisäikeisyyttä että blokkaamatonta IO:ta samanaikaisesti. Kutakin ydintä (tai tarkemmin laitteistopohjaista säiettä) kohden luodaan muutama käyttöjärjestelmäsäie, ja jokaisessa säikeessä hyödynnetään blokkaamatonta IO:ta. Tällöin verkko-ohjelma pääsee hyödyntämään kunkin ytimen prosessointiresursseja, mutta toisaalta säikeiden määrän rajallisuudesta johtuen muistin kulutus pysyy rajallisempana kuin jos kaikki olisi toteutettu monisäikeisyyden avulla. [7]

3.3 Rinnakkaisajon mallit

Luvussa 3.2 esitettiin kaksi rinnakkaisajon perusmallia: säikeiden käyttö sekä verkko-operaatioiden multipleksaus blokkaamattoman IO:n ja tapahtumasilmukan avulla. Ilman korkeamman tason IO-multipleksauskirjastoja säikeiden käyttö vaikuttaa usein helpommalta ratkaisulta IO-operaatioiden rinnakkaistamiseksi: Monisäikeisessä ohjelmassa, ohjelma joutuu yksinkertaisesti kertomaan käyttöjärjestelmälle mitä funktiota kunkin säikeen tulisi ajaa, minkä jälkeen käyttöjärjestelmä käynnistää uuden säikeen. Kussakin säikeessä voidaan käyttää synkronista IO:ta ja tavanomaisia synkronisen ohjelmoinnin perusperiaatteita, jossa komennot etenevät sarjassa. Pitkäkestoisten silmukkarakenteiden käyttö onnistuu ongelmitta, vaikka ne sisältäisivätkin useitakin IO-kutsuja, sillä käyttäjän näkökulmasta kukin säie toimii itsenäisesti eikä ohjelman tarvitse huolehtia vaihtelusta eri säikeiden välillä. [1] Lisäksi säiemallin synkronisuuden vuoksi kolmannen osapuolen verkkokirjastojen käyttöönotto on usein mutkattomampaa kuin jos käytössä on blokkaamaton IO, sillä tällöin kaikkia verkkokirjaston IO-operaatioita ei tarvitse portata yhteensopivaksi oman multipleksausmallin kanssa [10]. Toki säikeiden kanssa on otettava huomioon säieturvallisuus.

Silti IO-multipleksaus on usein parempi ratkaisu nimenomaan paremman suorituskyvyn ansiosta. Vaikka alhaisen tason rajapintojen käyttö onkin haastavaa, ohjelmoinnin helpottamiseksi on kehitetty useita erilaisia ratkaisuja. [11] Yleisesti eri ohjelmointialustoilla käytetty IO-multipleksauksen abstraktio on asynkroninen IO.

Asynkronisessa ohjelmointimallissa käyttäjä keskittyy määrittelemään eri prosessointiketjujen kulkua. Toisin sanoen kunkin IO-operaation kohdalla ohjelmoija voi määrittää, mitä ohjelman tulee tehdä kyseisen IO-operaation päätyttyä, esimerkiksi että vastaanotettuasi HTTP-pyyntö päätelaitteelta prosessoi se ja lähetä vastaus; vastauksen lähetettyä sulje yhteys. Perinteisesti prosessointiketjut määritellään takaisinkytkentäfunktioiden (callback function) avulla, eli kunkin asynkronisen verkko-operaatiokutsun kohdalla kerrotaan, mitä funktiota tulee kutsua, kun IO-operaatio saatetaan päätökseen. Takaisinkutsufunktio määritellään joko yhtenä IO-funktion parametrina tai vaihtoehtoisesti IO-operaatiokutsu palauttaa lupaustyyppisen objektin, jonka kautta määritys onnistuu. Esimerkiksi JavaScript:llä hypoteettista verkkopalvelinta voitaisiin ohjelmoida seuraavasti käyttäen mielikuvituksellista verkkokirjastoa:

```
server.serveRequest( (client)=>{
    client.readBody().then((body)=>{
        var json = body.toJSON();
        // ...
        client.sendResponse(200, "OK");
    })

});
```

Esimerkkikoodissa määritellään asynkronisesti, mitä palvelimen tulee tehdä kullekin päätelaitteen lähettämälle http pyynnölle. Funktio on asynkroninen, joten se ei jää pyörimään loputtomasti vaan sen avulla ainoastaan määritellään sääntö sille, mitä IO-operaation valmistuttua tulee tehdä. Kun sääntö on määritelty ohjelman ajo jatkuu eteenpäin. Ulomman takaisinkutsufunktion sisällä nähdään, että prosessointia on ketjutettu toisen takaisinkutsufunktion avulla: kun http-pyyntö saapuu, päätelaitteelta luetaan kaikki data (myös asynkronisesti) ja määritellään mitä tulee tehdä, kun kaikki data on luettu. Jälkimmäinen takaisinkutsufunktio on määritelty lupaus-objektin avulla: readBody-funktio ei ota mitään argumenttia, mutta palauttaa lupausobjektin, jonka then-funktion avulla takaisinkutsu määritellään. Lupaus-objekti on eräänlainen käsittelijä (handler), jonka avulla päästään (ohjelmointikielestä ja kirjastosta riippuen)

käsiksi asynkronisen operaation tilaan, ja joka antaa paljon enemmän vapauksia ketjutuslogiikan määrittelylle kuin takaisinkytkentäfunktion määrittely parametrin avulla: lupaus-objektin avulla voidaan mahdollisesti tarkastella operaation tilaa, voidaan määritellä eri takaisinkutsufunktioita sen perusteella, onnistuuko IO-operaatio tai ei ja sen avulla IO-operaatio voidaan myös perua.

Yllä olevassa koodinpätkässä on käytetty useita nykyaikaisia ohjelmointikielien ominaisuuksia helpottamaan asynkronisen ohjelmoinnin kirjoittamista: Käyttämällä lambdoja eli anonyymeja funktioita, prosessointiketju voidaan esittää hyvin kompaktisti, sillä jokaiselle prosessointiketjun osalle ei tarvitse kirjoittaa funktiomääritelmää, joka olisi erillään itse IO-operaatiosta. Tämä helpottaa koodin lukemista, sillä nähdään suoraan mikä koodi seuraa mitäkin, vaikkakin kyseessä ovat erilliset funktiot. Lisäksi kyseisessä esimerkissä on hyödynnetty lambdojen muuttujien sieppaus -ominaisuutta, jonka avulla lambdan sisällä voidaan nähdä ulomman funktion paikallisia muuttujia. Tämä helpottaa prosessointikontekstin viemistä eteenpäin prosessointilohkolta toiseen, ilman että olisi tarvetta eksplisiittisesti tallentaa sitä johonkin tietorakenteeseen. Lambdojen tuki on lähes poikkeuksetonta korkeamman tason kielissä, ja niitä tuetaan myös C++11-standardissa mahdollistaen tehokkaan asynkronisen ohjelmoinnin myös alhaisemman kielen tapauksessa. Kuitenkaan C-ohjelmointikieli ei tue anonyymeja funktioita, joten asynkroninen ohjelmointi on C-kielellä mutkikkaampaa.

Vaikka lambdat tekevätkin asynkronisesta ohjelmoinnista helppoa, niiden avulla ei kuitenkaan voida kirjoittaa koodia, joka näyttää synkroniselta. Esimerkiksi pitkiä IO-operaatioita sisältäviä silmukkarakenteita ei voi kovin helposti kirjoittaa, sillä prosessointi pitää pilkkoa erillisiksi funktioiksi kunkin asynkronisen IO-operaation kohdalla. 2010-luvulla kuitenkin asynkroninen ohjelmointi sai uuden buumin, kun moni ohjelmointikieli alkoi omaksua korutiiniominaisuuksia: Google julkaisi uuden kokonaan korutiinien ympärille rakentuvan asynkronisuuteen erikoistuvan Go-kielen vuonna 2009, Microsoftin NET-ympäristön kieliin korutiinit tulivat vuonna 2012, Pythoniin nykyaikainen korutiinisyntaksi saapui vuonna 2015, JavaScriptiin vuonna 2016 ja C++:aan vuoden 2020 standardiin [12] [13] [14] [15] [16].

Korutiinit ovat tavallisten funktioiden kaltaisia aliohjelmia, mutta niissä voi käyttää asynkronisia funktioita synkroniseen tapaan. Korutiinissa funktion eteneminen

pysähtyy aina asynkronisen IO-kutsun kohdalla, kunnes operaatio saadaan päätökseen mahdollistaen IO-operaatioiden peräkkäisen suorittamisen. Päältäpäin näyttää siltä, että käytettäisiin blokkaavaa IO:ta, mutta todellisuudessa näin ei ole: aina kun korutiini kutsuu asynkronista funktiota, korutiini sammutetaan kokonaan ja sen ajamista jatketaan sen jälkeen, kun asynkroninen funktio ajetaan loppuun. Korutiinin ollessa sammuksissa samalla säikeellä voidaan suorittaa muita asynkronisia tehtäviä.

Korutiinit ovat hyvin monimutkaisia sisäiseltä toteutukseltaan ja vaativat kääntäjältä tai kielitulkilta paljon työtä. Yksinkertaistettuna voidaan ajatella, että kääntäjä pilkkoo korutiinin osiin kunkin IO-operaation kohdalla ja korutiinin pilkottu osanen toimii ikään kuin takaisinkutsufunktiona sitä osaa edeltävälle asynkroniselle funktiolle. Koska asynkroninen funktio voi sisältää erinäisiä paikallisia muuttujia, joutuu kääntäjä muodostamaan korutiinille korutiinikontekstin, johon tallennetaan korutiinin sisäinen ajotila ja muuttujat. Ulkoapäin katsottuna korutiini toimii itsekin kuin asynkroninen funktio palauttaen lupaus-tyyppisen objektin.

Tarkemmin sanottuna korutiineissa odotus tapahtuu ainoastaan silloin, kun siinä käytetään await-operaattoria lupausobjektien yhteydessä: await-operaattori ikään kuin asettaa takaisinkutsufunktion asynkronisen funktion palauttamaan lupausobjektiin.

Alla yksinkertainen esimerkki, jossa asynkronisen ohjelman avulla vuorotellen vastaanotetaan ja lähetetään viestejä WebSocket yhteyden kautta:

```
async function processWSConnection(wsClient){
    var c = 0;
    while(!wsClient.hasError()){
        var msg = await wsClient.readMessage()
        // ... Viestin prosessointi
        var response = process(msg)
        await wsClient.sendMessage(response)
        c++;
    }
    return c;

}
```

Esimerkissä nähdään, että funktio on merkattu korutiiniksi async-avainsanan avulla. Lisäksi huomionarvoista on, että vaikka funktio näyttääkin palauttavan luvun, todellisuudessa funktio palauttaa lupausobjektin, sillä kyseessä on asynkroninen funktio.

Korutiinien käyttö tekee ohjelmoinnista yhtä vaivatonta kuin synkroninen ohjelmointi tai säikeiden käyttö. Kuitenkin korutiini hyödyntää säikeen sisäistä IO-operaatioiden

multipleksausta tehden siitä sisäisesti hyvin erilaisen toisiin menetelmiin verrattuna. Korutiineihin pohjautuva multipleksaus voidaan toteuttaa kokonaan yhden säikeen sisällä, mutta jos prosessointia on paljon, yksi säie ei riitä kaiken prosessoinnin tekemiseen, jolloin korutiinien käyttöä voi yhdistää monisäikeisyyden kanssa samalla tavalla kuin tavallista multipleksaustakin.

Asynkronista funktiota kutsutaan usein myös tehtäväksi. Tehtävä voidaan ajaa samalla säikeellä kuin funktio, joka sen käynnisti tai vaihtoehtoisesti tehtävä voidaan ajaa myös eri säikeellä.

3.4 Rinnakkaisuuden haasteita ja hyvän ohjelmointirajapinnan meriittejä

Vaikka IO:n rinnakkaistaminen lisääkin huomattavasti suorituskykyä, aiheuttaa useamman verkkoyhteyden samanaikainen prosessointi myös erinäisiä haasteita. Koska yhteyksien määrä voi kasvaa tuhansiin, ohjelman tehokkuus ja skaalautuvuus muuttuvat erittäin keskeisiksi. Yllä mainittiin säikeiden muistinkulutuksesta, minkä vuoksi niiden kanssa ollaan varovaisia esimerkiksi rajoittamalla aktiivisten säikeiden määrää. Myös asynkronisissa rajapinnoissa, jotka käyttävät tapahtumapohjaista IO:ta, tehokkuudella on väliä. Tapahtumasilmukan, joka vaihtelee eri tehtävien välillä sitä mukaan, kun eri verkkoyhteyksien tila muuttuu, on kyettävä seurata potentiaalisesti tuhansien verkkoyhteyksien tiloja mahdollisimman tehokkaasti ilman että kunkin verkkoyhteyden tilan tarkastaminen veisi paljon aikaa. Tämä on toki abstrahoitu yleensä asynkronisissa kirjastoissa kirjaston sisälle, mutta kirjastoa kirjoitettaessa tällä on keskeinen vaikutus ohjelman suorituskykyyn.

Toinen päänvaivaa aiheuttava ongelma on synkronisointi ja resurssien jakaminen, mikä on ongelma erityisesti useamman säikeen tapauksessa. [17] Usein verkko-ohjelmoinnissa useampi tehtävä saattaa yrittää muokata ohjelman yhteistä tilaa, esimerkiksi jotain tietokantaa. Jos tämä tapahtuu samanaikaisesti, saattaa yksi tehtävä jättää tilanpäivityksen puolitiehen, kun toinen tehtävä yrittää sitä lukea, jolloin saatu tulos on määrittämätön. Ratkaisuksi on kehitetty mutekseja eli tietynlaisia lukkoja, jotka estävät useamman säikeen pääsemästä käsiksi samaan resurssiin samanaikaisesti. Kuitenkin lukkojen käyttö vaatii huolellisuutta, sillä vääränlainen käyttö saattaa johtaa tilanteeseen, jossa säikeet lukkivat toisensa loputtomiin (deadlock), kun kumpikin

yrittää odottaa resurssia, jonka toinen säie on lukinnut itselleen. [18] Kun käytössä on vain yksi säie, synkronisointiongelma ei ole yhtä oleellinen, sillä silloin synkronisten tilapäivitysten aikana tehtävien välillä ei vaihdeta [11]: mahdolliset vaihtumiset tapahtuvat vain asynkronisten operaatiokutsujen aikana.

Myös debuggaus on usein haasteellista asynkronisessa ohjelmoinnissa. Kun käytössä on esimerkiksi takaisinkutsufunkioita, debuggeri ei näytä takaisinkutsupinoa, jolloin prosessoinnin logiikka ei ilmene debuggerista kunnolla. Myös korutiinien tapauksessa asynkronisten funktioiden kutsupino ei välttämättä näy oikealla tavalla: esim. C++-kielessä gdb ei oletuksella kykene näyttämään kutsupinoa, sillä korutiinien toteutus on pitkälti ohjelmoijan itsensä varassa, joten debuggerilla ei ole yleistä korutiinimallia, josta se kykenisi muodostamaan kutsupinotiedon [19].

Edellä käsiteltyjen asioiden valossa voidaan todeta, että hyvä rajapinta tarjoaa työkaluja näiden haasteiden ylittämiseen: kirjaston tulee olla hyvin skaalautuva, sen tulee hyödyntää blokkaamatonta IO:ta, mutta tarjottava myös tukea monisäikeiselle ajolle. Kirjastossa tulee olla tukea synkronisointilukoille ja mieluusti myös tapoja tunnistaa deadlock-ilmiöitä.

Seuraavassa luvussa käsitellään rinnakkais-IO:n käyttöjärjestelmätukea, jossa erityisesti tehokkuus ja skaalautuvuus ovat keskeisessä asemassa.

4 Rinnakkais-IO:n käyttöjärjestelmätuki

Koska kaikki verkko-operaatiot suoritetaan käyttöjärjestelmien rajapintojen avulla, rajapinnat ovat kriittisessä roolissa tehokkaan ja nopean verkko-ohjelman kehittämiseksi. Koska rinnakkaisohjelmoinnin tarve ja vaatimukset ovat ajan myötä kehittyneet, näkyy sama kehitys myös käyttöjärjestelmien tarjoamissa IO-multipleksaus-rajapinnoissa. Ensimmäisen iteraation rajapinnoissa ei huomioitu rinnakkaisuuden kasvua, joten ne toimivat optimaalisesti vain muutaman samanaikaisen verkkoyhteyden tapauksessa. Myöhemmissä rajapinnoissa skalautuvuus- ja muu optimointi on otettu paremmin huomioon: rajapintoja on pyritty kehittämään niin, että niiden nopeus olisi verkkoyhteysmäärästä riippumatonta. Lisäksi niistä on karsittu ohjelman ja käyttöjärjestelmäytimen välistä muistinsiirtoa sekä pyritty minimoimaan järjestelmäkutsuista aiheutuvia kontekstikytkentöjä.

Alkuperäiset verkko-IO rajapinnat ovat saaneet paljon inspiraatiota POSIX-standardista (jonka Linux toteuttaa myös suoraan). Tämän vuoksi vanhemmat asynkronisen IO:n rajapinnat vaikuttavat Linuxin ja Windowsin tapauksissa aika samankaltaisilta. Kuitenkin koska standardisointiprosessi on aina hidasta, ovat käyttöjärjestelmien myöhemmin kehittämät rajapinnat erkaantuneet toisistaan enemmän.

Seuraavissa alaluvuissa käsitellään tarkemmin Linuxin ja Windowsin IO-multipleksausrajapintoja ja tarkastellaan niiden suorituskykyä.

4.1 Linux-rajapinnat

Eniten käytetyt Linuxin multipleksausrajapinnat ovat tapahtumapohjaisia hyödyntäen blokkaamatona IO:ta. Niissä hyödynnetään BSD-sokettirajapinnan operaatiokutsuja, mutta operaatiot eivät kuitenkaan toimi blokkaavasti vaan palauttavat virhekoodin, jos operaatiota ei voida suorittaa heti, esimerkiksi jos. verkkoyhteysdata ei ole vielä saapunut palvelimelle.

Linuxin tapahtumapohjaisissa rajapinnoissa itse IO-kutsut pysyvät muuttumattomina, mutta rajapintojen välillä muuttuu mekanismi, jonka avulla pistokkeiden tiloja (luettavuutta ja kirjoitettavuutta) voidaan monitoroida. Monitoroinnin avulla voidaan havainnoida useamman pistokkeen tilanmuutosta ja reagoida heti, kun esimerkiksi joku pistokkeista muuttuu luettavaksi.

Kaksi vanhinta multipleksausrajapintaa saapuivat Linuxiin melkein heti ja ovat osa POSIX-standardia: select (Linux 1.0, 1994) ja poll (Linux 2.0, 1996) [20]. Select on myös osa alkuperäistä BSD-sokettirajapintaa [2]. Molemmissa rajapinnoissa funktiolle annetaan lista verkkoyhteystunnisteita, joiden tiloja halutaan monitoroida sekä spesifioidaan kullekin verkkoyhteydelle, halutaanko havainnoida luettavuutta, kirjoitettavuutta vai molempia. Oletusasetuksilla funktiokutsu palaa heti kun yksikin annetuista verkkoyhteyksistä muuttuu luettavaksi tai kirjoitettavaksi. Funktiot palauttavat samantyyppisen listan annetuista verkkoyhteyksistä, jossa listan kukin elementti kertoo kunkin verkkoyhteyden tilan. [1]

Funktioille select ja poll parametrit annetaan hieman eri lailla, mutta olennaista on kuitenkin se, että molemmille funktioille annetaan aina lista kaikista pistokkeista ja

funktio palauttaa saman pituisen listan riippumatta siitä, kuinka moni pistoke on vaihtanut tilaansa. Etenkin, jos kyseessä on tilanne, jossa suurin osa verkkoyhteyksistä on horrostilassa (engl. idle) eli ne eivät lähetä aktiivisesti dataa, palautetussa listassa vain harva pistoke vaihtaa tilaansa. Kuitenkin, koska listan pituus on vakio, prosessointiaika heikkenee lineaarisesti pistokkeiden määrän kasvaessa, sillä ohjelma joutuu läpikäymään aina koko listan. [1]

Toinen ongelma varhaisimpien rajapintojen kanssa on, että ohjelma joutuu lähettämään pistokelistan jokaisen odotuskutsun yhteydessä. [21] Näin siitä huolimatta, että pistokelista muuttuu yleensä vain vähän, jos laisinkaan funktiokutsujen välillä. (Esimerkiksi jos mietitään WebSocket-palvelinta, päätelaitteet, jotka yhdistyvät palvelimeen, yleensä ylläpitävät pitkäaikaisia verkkoyhteyksiä, joten sen jälkeen, kun verkkoyhteys muodostetaan, se pysyy yhdistyneenä pitkään.) Tämä johtaa turhaan muistinsiirtämiseen ohjelman ja käyttöjärjestelmäytimen välillä, mikä on hidasta. Myös käyttöjärjestelmäydin tekee turhaa työtä joutuessaan lähdekoodin perusteella jatkuvasti luomaan ja vapauttamaan sisäisiä monitorointitaulukoita ja rekisteröimään monitorointeja kullekin pistokkeelle [1]. Näin ollen käyttöjärjestelmäydinpuolen prosessointi hidastuu lineaarisesti.

Yllä mainitut skaalautuvuusongelmat johtivat uuden rajapinnan kehittämiseen Linuxissa, joka pyrkii korjaamaan kaikki yllä mainitut ongelmat. Toisin kuin aikaisemmissa rajapinnoissa epoll-rajapinta (Linux 2.6, 2003) [20] ei vastaanota epoll_wait -odotusfunktiossaan mitään listaa kiinnostavista pistokkeista. Pistokkeet konfiguroidaan etukäteen, kukin erikseen, epoll_ctl-funktiokutsun avulla. Kun pistoke on konfiguroitu monitoroitavaksi, se pysyy (asetuksista riippuen) monitoroitavana niin kauan, kunnes yhteys katkeaa. Kun epoll_wait funktio palaa, se palauttaa vain listan niistä verkkoyhteyksistä, joiden tila on muuttunut. Näin ollen epoll-rajapinta vähentää huomattavasti muistin siirtämistä ohjelman ja käyttöjärjestelmäytimen välillä, koska näiden välillä välitetään vain uutta tietoa. Lisäksi käyttöjärjestelmän kuormitus pienenee huomattavasti, sillä jokainen verkkoyhteys rekisteröidään monitoroitavaksi vain kerran. [1]

---

[1] select-komennon lähdekoodi löytyy Githubista: https://github.com/torvalds/linux/blob/master/fs/select.c.

Toki epoll-rajapinnan paremmuus riippuu palvelinohjelman laadusta. Esimerkiksi yhdessä rajapintoja vertaavassa suorituskykytutkimuksessa [22] havaittiin, että yhden tavun rasitustestissä, jossa kukin asiakasyhteys pyysi palvelimelta vain yhtä tavua luettavaksi, epoll-rajapinta suoriutui muutama prosentti heikommin kuin poll ja select. Tämä johtui suuresta määrästä epoll_ctl-kutsuja. Kun mukaan lisättiin 10 000 nukkuvaa verkkoyhteyttä, epoll-rajapinnan vahvuus näkyi kuitenkin hyvin: 30 000 palvelinpyynnöllä sekunnissa, epoll-rajapinta kykeni vastaamaan 3 kertaa suurempaan määrään pyyntöjä kuin muut rajapinnat. Näyttääkin siltä, että epoll-rajapinnalla on korkeammat verkkoyhteyden alustuskustannukset (koska jokaiselle verkkoyhteydelle tulee tehdä erillinen epoll_ctl-kutsu), mutta pitempiaikaisilla verkkoyhteyksillä epoll toimii hyvin.

Toisin kuin aikaisemmat rajapinnat, epoll on suunniteltu toimivan myös monisäikeisillä ohjelmilla. Kaikissa rajapinnoissa toki voidaan monitoroida samoja verkkoyhteyksiä useammalla säikeellä, mutta epoll-rajapinta mahdollista muun muassa sen, että useampi odotusfunktio ei tiedota samasta pistokkeen tilamuutoksesta useampaan kertaan. [23] Tämä on luultavasti myös yksi syy, miksi epoll on edelleen suosittu rajapinta Linuxissa.

Kuitenkin myös poll- ja select- rajapinnat ovat edelleen käytössä monessa vapaalähdekoodin ohjelmassa. Tämä johtuu rajapintojen järjestelmäriippumatto-muudesta, mikä mahdollistaa sen, että ohjelman porttaaminen eri alustoille on helppoa. [1] Tosin käyttämällä korkeamman tason multipleksaus- tai asynkronisen ohjelmoinnin kirjastoja voidaan kirjoittaa tehokkaita järjestelmäriippumattomia palvelinohjelmia, jossa kirjasto hyödyntää kullakin alustalla parasta mahdollista multipleksausrajapintaa.

Vaikka epoll korjasikin edeltäjiensä ongelmia, se kärsii melko korkeasta järjestelmäkutsujen määrästä (muun muassa epoll_ctl-kutsut, joita on perinteisesti pidetty yhtenä suorituskykyä rajoittavana tekijänä (ks. luku 2.2). Tämän vuoksi viime vuosina on kehitetty useita eri menetelmiä, joiden avulla on pyritty minimoimaan järjestelmäkutsujen määrää. [24] [25] Esimerkiksi reitittimiä ja kuormantasapainottajia varten on kehitetty Data Plane Development Kit (DPDK) -ohjelmointialusta, jonka avulla kaikki verkkoprosessointi voidaan siirtää ytimestä ohjelmaan. Ohjelma ohjaa suoraan verkkosovitinta kirjastojen kautta, jolloin tarve järjestelmäkutsuille katoaa. [26] DPDK:n on havaittu nopeuttavan huomattavasti verkkopakettien prosessointia

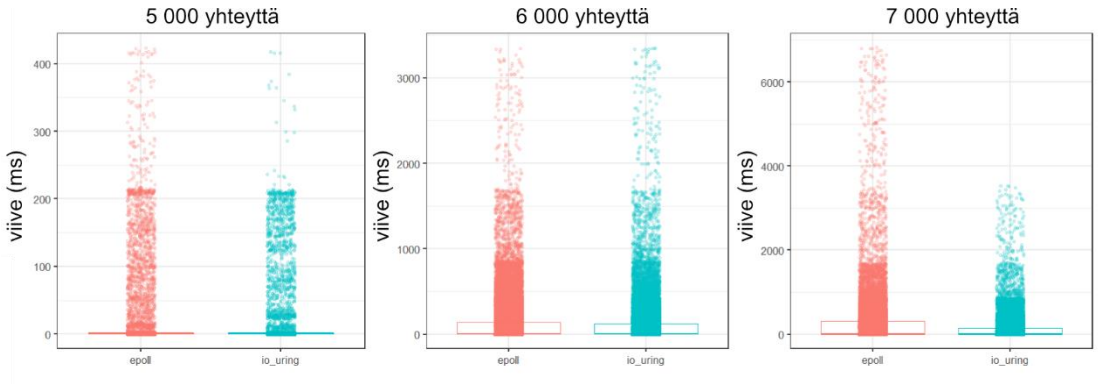
[27]. Koska DPDK ottaa haltuun koko verkkoprotokollapinon, se ei kuitenkaan sovellu tavallisten verkkopalvelinohjelmien rakennuspalikaksi.

Palvelinohjelmien kannalta kiinnostavampi ratkaisu on Linux-versiossa 5.1 (2019) julkaistu uusi multipleksausrajapinta io_uring. Rajapinnan avulla voidaan minimoida järjestelmäkutsujen ja muistinsiirtojen määrää yhdistämällä monta eri käyttöjärjestelmäkutsua yhteen. Rajapinta hyödyntää kahta rengaslistaa, jotka näkyvät sekä ohjelman että käyttöjärjestelmäytimen muistissa. Ohjelma kirjoittaa IO-pyyntöjä yhteen listaan (request queue) ja käyttöjärjestelmä kirjoittaa tulokset toiseen listaan (completion queue) sitä mukaan kuin operaatiot valmistuvat. Ohjelma joutuu vain periodisesti ilmoittamaan käyttöjärjestelmälle, että pyyntölistaan on lisätty uusia IO-pyyntöjä. Lisäksi io_uring tarjoaa toisen funktion, jonka avulla voidaan odottaa, kunnes tuloslistassa on uusia elementtejä. Koska pyyntölista voi olla tuhansienkin elementtien pituinen, voidaan monta pyyntöä hoitaa yhden järjestelmäkutsun avulla. [28]

Lisäksi io_uring ei enää vaadi tapahtumapohjaisen IO:n käyttöä, vaan se hyödyntää luvussa 3.2 esitettyä proaktiivista IO-mallia. Verkko-operaatiokutsut käskevät käyttöjärjestelmää käynnistämään verkko-operaatioiden suorittamisen heti taustalla, ja odotusfunktion avulla odotetaan IO-operaation valmistumista, eikä pistokkeen tilanmuutosta. Tällöin ei uutta erillistä read- tai write- käskyä tarvita odotusfunktion jälkeen. Tämäkin vähentää järjestelmäkutsujen määrää.

Tutkimuksissa on havaittu io_uringin tarjoavan yleisesti parempaa suorituskykyä kuin epoll. Erään tutkimuksen mukaan io_uringin hyöty riippuu pitkälti kontekstinvaihdon nopeudesta, johon vaikuttaa käyttöjärjestelmän konfigurointi erilaisten CPU-haavoittuvuuksien osalta. Kun nämä ovat päällä (ja yleensä ne ovat päällä turvallisuussyistä), kontekstinvaihdot järjestelmäkutsuissa ovat hitaita, jolloin niitten yhdistäminen tekee io_uringistä nopeamman kuin epoll. [29] Yhdessä tutkimuksessa [30] io_uring- ja epoll-rajapintojen suorituskykyjä verrattiin käytännöntasolla toteuttamalla MMO (*Massively Multiplayer Online*) -palvelin käyttäen kumpaakin rajapintaa. MMO-palvelin on onlinepelien pelaamisen mahdollistava palvelin, johon yhdistyy samanaikaisesti suuri määrä pelaajia. Tutkimuksessa havaittiin, että esimerkiksi 7000 pelaajan tapauksessa keskimääräinen viive oli epoll-palvelimella 253 ms ja io_uring palvelimella 148 ms. Lisäksi io_uring:ssa viiveen keskihajonta oli paljon

pienempi, joten viivejakauman häntäpäässä io_uringin parempi suorituskyky näkyy vielä paremmin, kuten näkyy kuvassa Kuva 1 alla.



Kuva 1. MMO-palvelimen viive eri toteutuksilla ja vaihtelevalla yhteyksien määrällä. **[30]** (Oma käännös)

Kuitenkin toistaiseksi io_uring rajapinnan käyttö on ollut melko rajallista, eivätkä esimerkiksi monet kirjastot kuten Node.js tai Tokio hyödynnä niitä (ks. luku 5). Tämä johtunee siitä, että io_uring-rajapinta on melko uusi ja uusia ominaisuuksia on lisätty vielä ainakin vuonna 2023 [31]. Tämän vuoksi io_uring-rajapinnan käyttöönotto rajoittaa huomattavasti, mihin sillä kehitettyjä ohjelmistoja voidaan asentaa. Lisäksi io_uring eroaa proaktiivisella IO-mallillaan huomattavasti edellisten multipleksausrajapintojen reaktiivisesta mallista, jolloin ohjelmistojen porttaaminen saattanee vaatia suuriakin muutoksia. Kuitenkin koska io_uring on kuitenkin nopeampi kuin epoll, voidaan olettaa, että tulevaisuudessa sen käyttö yleistyy.

4.2 Windows-rajapinnat

Kuten Linuxin niin myös Windowsin asynkroninen verkkorajapinta on kehittynyt ajan myötä. Alkuperäinen Winsock-verkkokirjasto (versio 1.1.) julkaistiin vuonna 1993 ja seurasi hyvin läheisesti BSD-sokettispesifikaatiota toteuttaen myös select-multiplekasusrajapinnan [32]. Kuten Linuxinkin tapauksessa Windows-versio tästä rajapinnasta kärsii skaalautuvuusongelmasta, eikä tätä rajapintaa suositella palvelinohjelmien kehittämiseksi [33].

Kehittyneempi nykyään käytetty Winsock 2.2 rajapinta julkaistiin kuitenkin jo vuonna 1996 [34] sisältäen useita skaalautuvuutta mahdollistavia menetelmiä. Toisin kuin Linuxissa Windowsissa siirryttiin heti tuolloin reaktiivisesta IO-mallista proaktiiviseen

IO-malliin, jossa IO-operaatiot ajetaan päätökseen taustalla samalla tavalla kuin io_uringissa.

Ohjelma voi monitoroida IO-operaatioiden valmistumista usealla eri tavalla. Kaksi skaalautuvinta vaihtoehtoa ovat IO-valmistumisrutiinit (engl. IO completion routines) ja IO-valmistumisportit (engl. IO completion port, IOCP). IO-valmistumisrutiinit toimivat kuin takaisinkutsufunktiot. Ohjelma antaa kunkin IO-operaation kohdalla funktion, jota kutsutaan, kun operaatio on valmistunut. Näin ollen Windowsin rajapinta lähentelee jo itsestään asynkronista rinnakkaisajon mallia, sillä ohjelmoija voi yksinkertaisesti määritellä eri IO-operaatioiden välisiä relaatioita. Kuitenkin, jotta takaisinkutsufunktiot tulisivat kutsutuiksi, ohjelman tulee toteuttaa tapahtumasilmukka, jossa se asettaa säikeen jatkuvasti hälytettävään tilaan (alertable state), jolloin käyttöjärjestelmä automaattisesti herättää säikeen jonkun IO-operaation päätyttyä ja kutsuu vastaavan takaisinkutsufunktion. Ohjelma ei kuitenkaan itse joudu tarkistamaan mikä IO-operaatio on kyseessä tai kutsumaan takaisinkutsufunktiota manuaalisesti. [34]

IO-valmistusrutiinit toimivat säiekohtaisesti, ja takaisinkutsufunktio ajetaan aina samalla säikeellä kuin, jolla IO-operaatio on käynnistetty. IOCP toimii samalla tavalla, mutta mahdollistaa tehokkaamman monisäikeisyyden hyödyntämisen, sillä sen avulla takaisinkutsufunktioita voidaan ajaa millä tahansa säikeellä eikä vain IO-operaation aloittaneella säikeellä. [34]

Asynkronisuutensa vuoksi Windowsin IO-valmistusrutiini- IOCP-multipleksausta voidaan pitää kehittyneempänä kuin epoll-rajapintaa, sillä se vaatii vähemmän rajapintakutsuja kuin epoll:n tapahtumapohjainen IO. Kuitenkaan rajapintakutsuista ei päästä kokonaan eroon, joten Windows 8:aan tulikin uusi io_uringin kaltainen rangaspuskureita käyttävä järjestelmäkutsuista vapaa rajapinta Registered IO (RIO), jossa pyynnöt tallennetaan yhteen puskuriin ja tulokset tallennetaan toiseen [30]. Microsoftin mukaan [35] uuden rajapinnan tulisi tehostaa etenkin palvelinohjelmien suorituskykyä.

5 Asynkronisuus ohjelmointialustoissa

Käyttöjärjestelmien tarjoamat rajapinnat mahdollistavat IO-multipleksauksen ja tarjoavat siten hyvän pohjan asynkroniselle ohjelmoinnille. Kuitenkin ilman erillisiä kirjastoja asynkroniseen ohjelmointiin pääseminen on ohjelmoijan kannalta hyvin mutkikasta: ohjelmoija joutuu muun muassa kirjoittamaan tapahtumasilmukan sekä luomaan abstrahointeja, joiden avulla tapahtumapohjainen IO toimii asynkroniseen malliin takaisinkutsufunktioiden avulla. Asynkroniset kirjastot ja ohjelmointialustat piilottavat kaiken tämän kompleksisuuden käyttäjältä siten, että tämä pääsee heti kirjoittamaan asynkronista koodia käyttäen esimerkiksi takaisinkutsufunktioita tai korutiineja.

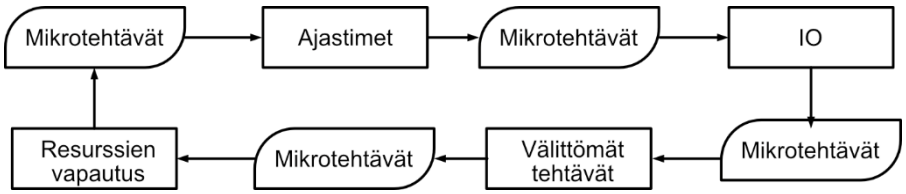
Vaikka eri alustat ja kirjastot tarjoavat paljon samoja työkaluja kuten async/await -tyyppisiä rakenteita, kirjastojen tapa toteuttaa IO-multipleksaus ja rinnakkaisajo vaihtelee välillä suurestikin. Tässä luvussa käsitellään asynkronisuuden toteutusta JavaScriptin Node.js-alustalla sekä Rust-kielen Tokio-ajoympäristössä.

5.1 JavaScript ja Node.js

Node.js on JavaScriptin palvelinpuoleinen ohjelmointiympäristö, joka alkujaan julkaistiin vuonna 2009. Se kehitettiin nimenomaan IO-multipleksausta ja asynkronisuutta ajatellen vastapainona tuon ajan synkronisille malleille ja monisäikeisyydelle rinnakkaisuuden hallinnassa. Kehityksen päämotivaattorina oli nimenomaan asynkronisen IO:n parempi suorituskyky verrattuna blokkaavaan IO:hon, sekä monisäikeisyydestä johtuvat muisti- ja synkronointiongelmat. [11] Node.js-alustan tarjoama asynkroninen tapahtumaperusteinen ohjelmointimalli on nähty sen verran onnistuneeksi, että se johti kyseisen ohjelmointimallin yleistymiseen verkko-ohjelmoinnissa ja mallin adoptoimiseen muissa ohjelmointialustoissa. [36]

Node.js pyörii kokonaisuudessaan vain yhdellä säikeellä suorittaen työtehtävien multipleksauksen tapahtumasilmukan avulla. Kaikki IO-operaatiokutsut ovat asynkronisia siten, että niille annetaan parametriksi takaisinkutsufunktio, joka ajetaan asynkronisen operaation päättyessä. Myöhemmin JavaScriptiin sekä Node.js-alustaan lisättiin korutiiniominaisuus helpottaen ohjelmointia entisestään.

Sisäisesti tapahtumasilmukka koostuu useasta vaiheesta, jotka on esitetty alla kuvassa Kuva 2. Kussakin vaiheessa on oma tapahtumalista (event queue), joka sisältää odotettavien tapahtumien listan sekä kullekin tapahtumalle sitä vastaavan listan takaisinkutsufunktioita ja lupausobjekteja, jotka herätetään, kun tapahtuma tapahtuu. Tärkeisiin päävaiheisiin kuuluvat mm. ajastinvaihe, IO-vaihe, sekä välittömien tehtävien vaihe. Ajastinvaiheessa kutsutaan ajastettuja funktioita, silloin kun aika käy umpeen; IO-vaiheessa tarkistetaan asynkronisten IO-operaatioiden tila ja kutsutaan sopivia takaisinkutsufunktioita aina kun joku IO-operaatio valmistuu. Välittömien tehtävien vaiheessa kutsutaan takaisinkutsufunktioita, jotka on määritetty ajettavaksi välittömästi. Tämä on hyödyllistä silloin kun halutaan pilkkoa pitkään kestävä laskutoimitus pienempiin osiin, jotta muidenkin tehtävien suorittaminen etenisi. Tarkempi eri vaiheiden kuvaus löytyy mm. Sunin ym. tutkimuksesta [36].



Kuva 2. Node.js-alustan tapahtumasilmukan vaiheet. Perustuu Sunin ym. tutkimuksessa **[36]** esitettyyn kaavioon.

Tapahtumasilmukka on toteutettu C-kielellä käyttäen Node.js-alustaa varten kehitettyä libuv-kirjastoa. Näin ollen itse Node.js on ohjelmoitu täysin käyttöjärjestelmäriippumattomasti, mutta libuv hyödyntää käyttöjärjestelmäriippuvia rajapintoja. Linuxissa käytössä on epoll ja Windowsissa IOCP. Ainakaan toistaiseksi io_uring tukea ei ole lisätty. [37]

Vaikka Node.js-alustan tehokas asynkronisuuden käyttö povaakin hyvää suorituskykyä, rinnakkaisuuden rajoittuminen yhteen säikeeseen on yksi alustan heikkouksista, minkä vuoksi toiset alustat kuten Googlen Go, joka hyödyntää sekä IO-multipleksausta että monisäikeisyyttä, suoriutuvat huomattavasti paremmin rasitustesteissä. Node.js:ään on kyllä ajan myötä lisätty tuki monisäikeisyydelle työsäikeet (Worker threads) -ominaisuuden avulla, mutta tuki on melko rajallista: työsäikeet on tarkoitettu vain laskuintensiivisten operaatioiden tekoon ja kukin säie vaatii erillisen instanssin Javascript ympäristöstä, joten säikeiden välinen tiedonvälitys on rajallista [38]. [39]

5.2 Rust ja Tokio

Rust on verrattain uusi vuonna 2015 julkaistu ohjelmointikieli, joka yhdistää alhaisten tason kielien tehokkuuden muistiturvallisuuden kanssa. Rust-kieltä on kehitetty asynkronisuutta silmällä pitäen, joten siihen on sisäänrakennettu esimerkiksi async/await komennot sekä standardisoitu Future- ja Waker- rajapinta asynkronisten tehtävien hallitsemiseksi. Kuitenkin, koska Rust on suunniteltu ajettavaksi erilaisissa ympäristöissä, se ei sisällä sisäänrakennettua ajoympäristöä (runtime) asynkronisen koodin ajamiseksi, vaan sopiva asynkroninen kirjasto tulee käynnistää ohjelmassa manuaalisesti.

Rustissa on toteutettu useita asynkronisia ajoympäristöjä, jotka tarjoavat erilaisia ominaisuuksia ja joiden toimintatapa eroaa IO-multipleksauksen ja monisäikeisyyden osalta huomattavastikin. Yleisin asynkroninen ajoympäristö on kuitenkin Tokio, joka on vakiintuneisuutensa vuoksi saanut eniten tukea kolmannen osapuolen kirjastoista. Tokio myös tarjoaa erilaisia konfigurointimahdollisuuksia rinnakkaisajolle, minkä vuoksi sitä voidaan kustomoida tarpeen mukaan [40].

Future-Waker-rajapintojen ansiosta suuri osa asynkronisista ajoympäristöistä on toteutettu käyttäen samaa perusmallia. Futuuri-objekti (Future) vastaa JavaScriptin lupaus-objektia, joka mahdollistaa asian suorittamista taustalla. Futuuri sisältää poll-metodin, jota kutsuttaessa Futuuri-objekti pyrkii ajamaan tehtäväänsä eteenpäin, kunnes se joutuu taas odotustilaan.

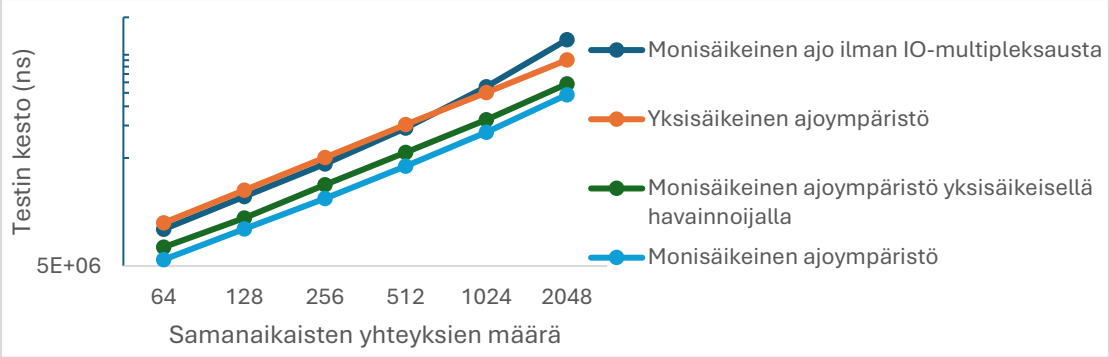
Itse ajoympäristö on jaettu kahteen komponenttiin: suorittajaan (executor) ja havainnoijaan (reactor), jotka toimivat toisistaan riippumattomasti Waker-tietotyypin avulla. Suorittaja hallinnoi eri tehtävien ajoa kutsuen vuorotellen kunkin tehtävän poll-metodia. Jos poll-metodi aiheuttaa Futuuri-objektin menemään odotustilaan, rekisteröi poll-metodi Futuuri-objektiin assosioidun Waker-objektin havainnoijaan. Waker-objektin on yksinkertaisesti rajapinta, joka sisältää wake-metodin. Havainnoija seuraa eri tapahtumia, kuten IO-tilojen muutoksia. Kun havainnoija tekee jonkun havainnon, joka antaa aihetta jatkaa jonkun tehtävän prosessointia, se kutsuu Waker-objektin wake-metodia. Wake-metodi pistää Futuuri-objektin suorittajan odotuslistalle herätettäväksi, jolloin suorittaja voi jälleen kutsua Futuuri-objektin poll metodia ja näin ollen jatkaa tehtävän prosessointia. [7]

Havainnoija ja suorittaja ovat toisistaan täysin riippumattomia, joten suorittajan ei tarvitse esimerkiksi tietää mitä multipleksausrajapintaa havainnoija käyttää. Vastaavasti havainnoijan ei tarvitse tietää mitä rinnakkaisajonmallia käytetään: korutiineja, säikeitä, takaisinkutsufunktioita, tms. Riippumattomuus mahdollistaa mm. sen, että ohjelmassa voi olla useita havainnoijia, jotka havainnoivat eri tapahtumia ja toimivat täysin eri rajapintojen avulla, esim. yksi havainnoija IO:lle ja yksi ajastimelle. [7]

Jotta havainnoija ja suorittaja toimisivat riippumattomasti, niiden kuitenkin on oltava eri säikeillä, sillä suorittaja ei voisi muuten suorittaa mitään tehtäviä, kun havainnoija menee lepotilaan. Kuitenkin optimoidumpaa on ajaa havainnoijaa ja suorittajaa samalla säikeellä, jolloin synkronointeja ja kontekstinvaihtoja säikeiden välillä ei tarvita. Toki tällöin havainnoija ja suorittaja eivät voi olla täysin erillisiä. [7] Mm. Tokio toimii oletuksella tällä tavalla. [40]

Samalla Tokio kuitenkin toimii monisäikeisesti siten, että kullakin säikeellä on oma havainnoija ja suorittaja, jotka kaikki prosessoivat IO-tapahtumia (monisäikeinen ajoympäristö). Tällöin Tokio pystyy hyödyntämään kaikkia prosessorin ytimiä maksimoiden tehokkuutta. Vaihtoehtoisesti Tokio voidaan myös säätää kokonaan yksisäikeiseksi, jolloin se toimii kuin Node.js-ympäristö (yksisäikeinen ajoympäristö). Kolmas tapa on rajoittaa havainnoija yhdelle säikeelle, mutta ajaa suorittajia monisäikeisesti (monisäikeinen ajoympäristö yksisäikeisellä havainnoijalla). [40] [41]

Näitten eri vaihtoehtojen suorituskykyä on vertailtu yhdessä tutkimuksessa [41], jossa Tokion avulla on luotu kaikupalvelin, eli palvelinohjelma, joka peilaa kaiken datan, jota sille lähetetään. Tokio konfiguroitiin käyttämään yhtä yllä kuvailluista kolmesta ajomallista, joita lisäksi verrattiin puhtaasti synkroniseen monisäikeiseen ratkaisuun, jossa jokainen palvelinyhteys hoidettiin erillisellä säikeellä. Samanaikaisten yhteyksien määrää säädettiin 64:n ja 2048:n välillä. Tulokset on esitetty kuvassa Kuva 3 alla. Tuloksista nähdään, että Tokion ajomalleista monisäikeinen toteutus on nopein ja yksisäikeinen on hitain. Lisäksi teorian mukaisesti yksisäikeisen havainnoijan käyttö on hitaampaa kuin jos kaikilla säikeillä on sekä oma havainnoija että suorittaja.



Kuva 3. Tokio-ajoympäristön eri konfiguraatioiden suorituskyky. Kuvaaja luotu raakadatasta. **[42]**

Vaikka tutkimuksessa tätä ei kommentoitu, tuloksista voidaan havaita, että pienillä yhteyksien määrällä säikeiden käyttö ilman IO-multipleksausta ja asynkronista ajoympäristöä on nopeampaa kuin yksisäikeinen ajoympäristö multipleksauksen kanssa, mutta että suuremmilla kuormituksilla yksisäikeinen ajoympäristö muuttuu nopeammaksi. Tämä liittynee siihen, että suurella yhteyksien määrällä säikeiden välinen kontekstinvaihtojen määrä kasvaa räjähdysmäisesti, mikä hidastaa monisäikeistä prosessointia. Yksisäikeisessä rinnakkaisuudessa tällaista ongelmaa ei ole, mikä puhuu asynkronisuuden ja IO-multipleksauksen puolesta.

Kuten Node.js Tokio hyödyntää Linuxissa epoll-rajapintaa ja Windowsissa IOCP-rajapintaa. [43] Tokiolle on kylläkin luotu myös io_uring versio, mutta sen kehitys on vielä alkuvaiheessa eikä kaikkia Tokion ominaisuuksia vielä tueta io_uring-versiossa [44]. Lisäksi io_uring-versiota ei ole kehitetty lähes vuoteen, joten vaikuttaa siltä, että Tokio-kirjasto jää toistaiseksi epoll-pohjaiseksi.

6 Yhteenveto

Tässä kandidaatintyössä tarkasteltiin verkko-operaatioiden ja verkkoyhteyksien prosessoinnin rinnakkaistamista keskittyen erityisesti IO-multipleksausrajapintoihin. Rajapintojen yhteydessä havaittiin huomattava kehitys kolmen vuosikymmenen sisällä. Alkuperäiset hitaat, muutamalle verkkoyhteydelle suunnitellut rajapinnat on korvattu hyvin skaalautuvilla monisäikeisyyttä tukevilla rajapinnoilla, joita on iteratiivisesti optimoitu useasta suunnasta. Linuxissa siirtyminen select/poll -rajapinnasta epoll rajapintaan vähentää muistinsiirtoa ohjelman ja käyttöjärjestelmäytimen välillä, sillä listaa monitoroitavista pistokkeista ei enää tarvitse lähettää kunkin odotusfunktion kohdalla, vaan kunkin pistokkeen pystyy rekisteröidä havainnoitavaksi pistokkeen luomisen yhteydessä erikseen. Tämä vähentää sekä muistin siirtoon menevää aikaa, että ydinpuolista prosessointitaakkaa. Siirtyminen epoll:sta io_uring-rajapintaan taas minimoi järjestelmäkutsujen määrää, mikä vähentää kontekstinvaihdoista aiheutuvaa hitautta. Windowsissa nähdään samantyyppistä kehitystä, jossa poll-tyyppinen rajapinta korvataan ensin IOCP-rajapinnalla ja sittemmin io_uring-kaltaisella RIO-rajapinnalla.

Vaikka uusimmat rajapinnat tarjoavat yleisesti parasta suorituskykyä, niiden käyttöönotto on ollut hidasta. Mielenkiintoisena havaintona moni rinnakkaisuutta hyödyntävä ohjelma tai asynkroninen verkkokirjasto hyödyntää vieläkin epoll ja IOCP-rajapintaa, vaikka uudemmat rajapinnat ovat olleet saatavilla jo vuosia. Osin tätä muutosta hidastavat vanhempien rajapintojen laajempi käyttöjärjestelmätuki, sillä uusimmat rajapinnat (ja etenkin jotkut io_uring-ominaisuudet) vaativat melko uuden käyttöjärjestelmäpaketin. Myös vanhempien rajapintojen laaja suosio ja yhteisötuki ovat osaltaan ehkä vaikuttaneet niiden suosion pysymiseen. Tämä näkyy mm. siinä, että io_uring-rajapinnasta on verrattain vaikea löytää tieteellisiä suorituskykytutkimuksia verkko-ohjelmoinnin näkökulmasta.

Verkko-operaatioiden rinnakkaistaminen blokkaamattoman IO:n ja multipleksauksen kautta on muovannut suuresti verkko-ohjelmoinnin työkaluja. Node.js:n innostamana yleistyivät asynkronisen ohjelmoinnin mallit, jossa ohjelmointi tapahtuu takaisinkutsufunktioiden ja korutiinien avulla. Vaikka asynkronisen mallin perusideat ovatkin samat, kandidaatintyössä esitettyjen esimerkkien – Node.js ja Tokio – avulla

nähdään, että rinnakkaistamisen toteutus voi olla hyvinkin erilainen ohjelmointialustasta riippuen. Alhaisemman abstraktiotason kielissä kuten Rust-kielessä, asynkronisuuden toimintamallia, kuten säikeisyyttä voidaan myös manuaalisesti säätää tarpeiden mukaan mahdollistaen suorituskyvyn, muistinkäytön ja ohjelmoinnin kompleksisuuden räätälöimisen sopivaksi.

Koska kandidaatintyössä tarkastuksessa olivat nimenomaan multipleksausrajapinnat, moni muu rinnakkaisajoon olennaisesti kuuluva asia jäi huomioimatta. Etenkin monisäikeisessä ajossa erityisen tärkeätä on synkronointi ja lukkojen käyttö datan välityksessä säikeiden välillä. Tässäkin käyttöjärjestelmien tuki on välttämätöntä. Verkko-ohjelmointikirjastojen näkökulmasta taas tehokas ja reilu prosessointiresurssien jako tehtävien välillä mahdollistaa viiveiden minimoimisen kaikille verkkopalvelimeen yhdistyneille käyttäjille.

Koska Internetin vaativuus tulee kasvamaan vielä entisestään, kasvavat myös paineet entistä tehokkaammille verkkorajapinnoille. Tulevaisuudessa varmasti nähdään io_uring- ja RIO-rajapintojen laajempi käyttöönotto, minkä jälkeen keksittäneen jälleen uusi rajapintojen optimointitapa johtaen taaskin uuteen rajapintojen iteraatioon.

Lähdeluettelo

[1] Kerrisk M. (2010). *The Linux programming interface : a Linux and UNIX system programming handbook.* USA: No Starch Press, Inc. ISBN: 978-159-327-220-3

[2] Hewlett-Packard Company (1997). *BSD Sockets Interface Programmer's Guide.* USA, 1997.

[3] Stancevic D. (2003). Zero copy I: user-mode perspective. *Linux Journal,* 2003(105), s. 3.

[4] Soares L. ja Stumm M. (2010). FlexSC: flexible system call scheduling with exception-less system calls. *OSDI'10: Proceedings of the 9th USENIX conference on Operating systems design and implementation*, s. 33–46 Vancouver, Kanada. DOI:10.5555/1924943.1924946

[5] Kegel D. (1999) The C10k Problem. Arkistoitu. Haettu *Dan Kegel's Web Hostel* -sivustolta osoitteesta, https://web.archive.org/web/19990508164301/ http://www.kegel.com/c10k.html .

[6] *Scaling to 12 Million Concurrent Connections: How MigratoryData Did It* (2013). MigratoryData. Haettu osoitteesta: https://migratorydata.com/blog/migratorydata-with-12-million-concurrent-websockets

[7] Samson C. F. (2024). *Asynchronous Programming in Rust.* Birmingham, UK: Packt Publishing Ltd. ISBN 978-1-80512-813-7.

[8] Alexeyev A. (2011). nginx. *The Architecture of Open Source Applications, Volume 2: Structure, Scale and a Few More Fearless Hacks*, s. 211–224. ISBN: 978-1-105-57181-7.

[9] Pyarali I., Harrison T., Schmidt D. C.. ja Jordan T. D. (1997). Proactor – An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events. *Pattern Languages of Programmin Conference*, Illinois, USA, 1997.

[10] Fowler M. (2022). *Python Concurrency with asyncio*, USA: Manning Publications Co, 2022. ISBN: 978-161-729-866-0

[11] Dahl R. (2009). Node.js. Esitelmä. *The European JavaScript Conference*, 2009. Berlin, Saksa.

[12] *FAQ* (n.d.). Haettu 13.5.2025 The *Go Programming Language* -sivustolta osoitteesta https://go.dev/doc/faq

[13] *Asynchronous programming with Async and Await (Visual Basic)* (2022). Haettu *Microsoft Learn* -sivustolta osoitteesta: https://learn.microsoft.com/en-us/dotnet/visual-basic/programming-guide/concepts/async/

[14] Selivanov, Y. (2015). *PEP 492 – Coroutines with async and await syntax*. Haettu 15.5.2025, *peps.python.org* -sivustolta osoitteesta https://peps.python.org/pep-0492/

[15] *async function – JavaScript* (2025). Haettu 15.5.2025, *mdn web docs* -sivustolta osoitteesta https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

[16] *C++20* (n.d.). Haettu 6.4.2025, sivustolta *cppreference.com* osoitteesta https://en.cppreference.com/w/cpp/20

[17] Choi S. E. ja Lewis E. C. (2000). A Study of Common Pitfalls in Simple Multi-Threaded Programs. *The proceedings of the thirty-first SIGCSE Technical Symposium on Computer Science Education: SIGCSE 2000, Austin, Texas, USA.*

[18] Holub A. (2000). *Taming Java Threads*. California, USA: Apress Berkeley. ISBN: 189-311-510-0

[19] The Clang Team. (n.d.). *Debugging C++ Coroutines*. Haettu 13. toukokuuta 2025 sivustolta *Clang documentation* osoitteesta https://clang.llvm.org/docs/DebuggingCoroutines.html

[20] *syscalls(2)*. (n.d.). Haettu 5.5. 2025, *Linux manual page* -sivustolta osoitteesta https://man7.org/linux/man-pages/man2/syscalls.2.html

[21] Love, R. (2007). *Linux System Programming*. USA: O'Reilly Media, Inc. ISBN: 144-933-953-0.

[22] Gammo, L., Brecht, T., Shukla, A. & Pariag, D. (2004). Comparing and Evaluating epoll, select, and poll Event Mechanisms. *Linux Symposium 2004*, s. 215–226.

[23] *epoll(7)*. (n.d.). Linux manual page. Haettu 14.5. 2025, *Linux manual pages* -sivustolta, osoitteesta https://man7.org/linux/man-pages/man7/epoll.7.html

[24] Cai, P. & Karstren, M. (2023). Kernel vs. User-Level Networking: Don't Throw Out the Stack with the Interrupts. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(3).

[25] Hruby T., Crivat T., Bos H. ja Tanenbaum A. S. (2014). OnSockets and System Calls – Minimizing Context Switches for the Socket API. *2015 USENIX Annual Technical Conference.* Amsterdam .

[26] *The Linux Foundation Projects*. (n.d.). *About.* Haettu 11.4.2025 sivustolta *DPDK*, osoitteesta https://www.dpdk.org/about/

[27] Belkhiri A., Pepin, M., Bly, M. & Dagenais, M. (2022). Performance analysis of DPDK-based applications through tracing. *Journal of Parallel and Distributed Computing*, 2023(173), s. 1–19. DOI: 10.1016/j.jpdc.2022.10.012

[28] Ahern D. & Mukherjee, S. (2023). Merging the networking worlds. *The Technical Conference on Linux Networking*.

[29] Wang X. (2022). *io_uring vs. epoll – Which Is Better in Network Programming?* Haettu 15.5.2025, *Alibaba Cloud Blog* -sivustolta, osoitteesta https://www.alibabacloud.com/blog/io-uring-vs--epoll-which-is-better-in-network-programming_599544

[30] Seong S. ja Jung N. (2020). Improving performance of MMO game server using io_uring I/O Model. *Journal of Korea Game Society*, 20(6), s. 31–42.

[31] *io_uring_setup(2)* (n.d.). Haettu 3.5.2025, *Linux manual pages* -sivustolta, osoitteesta https://man7.org/linux/man-pages/man2/io_uring_setup.2.html

[32] Towfiq M. (1993). *Windows Sockets – An Open Interface for Network Programming under Windows.* Haettu osoitteesta: https://www.ibiblio.org/pub/micro/pc-stuff/ms-windows/winsock/winsock-1.1/winsock.html

[33] Jone, A. ja Ohlund J. (2002). *Network Programming Under Microsoft Windows*. Microsoft Corporation.

[34] *Windows Sockets 2 – Application Programming Interface. An Interface for Transparent Network Programming Under Microsoft Windows*. (1996).

[35] *What's New for Windows Sockets*. (2022, 19. tammikuuta). Haettu 15. toukokuuta 2025, *Microsoft* -sivustolta, osoitteesta https://learn.microsoft.com/en-gb/windows/win32/winsock/what-s-new-for-windows-sockets-2?redirectedfrom=MSDN

[36] Sun H., Bonetta D., Schiavio F. ja Binder W. (2019). Reasoning about the Node.js Event Loop using Async Graphs. *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Washington, USA*.

[37] *Design overview* (n.d.). Haettu 27. huhtikuuta 2025, *libuv documentation* -sivustolta, osoitteesta https://docs.libuv.org/en/v1.x/design.html

[38] *Worker threads* (n.d.). Haettu 27.4.2025, *Node.js documentation* -sivustolta, osoitteesta https://nodejs.org/api/worker_threads.html

[39] Otynshin, D. (2023). Optimizing Node.js Application performance through main thread offloading. *International Journal of Information and Communication Technologies*, 2(4), s. 82–93.

[40] *tokio::runtime* (n.d.). Haettu 3.5.2025, *Rust documentation* -sivustolta, osoitteesta https://docs.rs/tokio/latest/tokio/runtime/index.html

[41] Vaner M. (2018). *Benchmark of different Async approaches in Rust*. Haettu 2.5.2025, *vorner.github.io* -sivustolta, osoitteesta https://vorner.github.io/async-bench.html

[42] Vaner M. (2018). *Benchmark of different Async approaches in Rust – raw data*. Haettu 3.5.2025, *GitHub* -sivustolta, osoitteesta https://github.com/vorner/corona/tree/master/benchmarks/results/parallel

[43] Lerche C. (2019). *Making the Tokio scheduler 10x faster – Tokio*. Haettu 15. toukokuuta 2025, *tokio.rs* -sivustolta, osoitteesta https://tokio.rs/blog/2019-10-scheduler#conclusion

[44] *tokio-uring* (n.d.). Haettu 3. toukokuuta 2025, *GitHub* -sivustolta, osoitteesta https://github.com/tokio-rs/tokio-uring

[45] Damyanov D. ja Varbanov Z. (2024). Using Asynchronous Programming in C# – Problems, Practical Tips and Scenarios: A Short Review. *5th International Conference on Communications, Information, Electronic and Energy Systems (CIEES), Veliko Tarnovo, Bulgaria*.