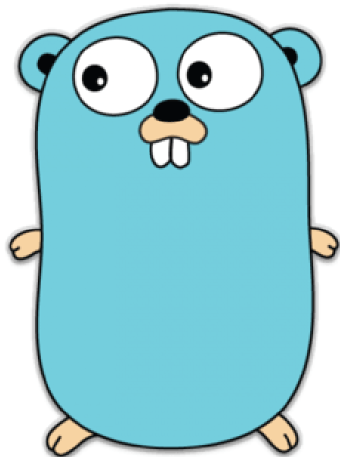
Roope Rantasalo

Go-ohjelmointikielen syntaksi ja suorituskyky



Go

Insinööri (AMK)

Tieto- ja viestintätekniikka

Kevät 2020

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä(t): Rantasalo Roope

Työn nimi: Go-ohjelmointikielen syntaksi ja suorituskyky

Tutkintonimike: Insinööri (AMK)

Asiasanat: Go, ohjelmointikielet, ohjelmointikielen syntaksi, ohjelmointikäytännöt, back end, verkkokehitys

Tämän insinöörityön tavoitteena oli syventyä Go-ohjelmointikielen syntaksiin ja käytänteisiin eli poimia tärkeimpiä huomioita eri valinnoista, joita Go:n kehityksessä on tehty liittyen sen kirjoittamiseen, lukemiseen ja suorituskykyyn. Lisäksi oli tarkoitus etsiä esimerkkejä siitä, mitä eri yritykset ovat Go:lla tehneet, sekä pyrkiä tunnistamaan Go:n vahvuuksia ja heikkouksia löydettyjen esimerkkien kautta. Löydettyjen vahvuuksien ja heikkouksien tueksi Go:ta vertailtiin myös kolmeen perinteisempään ohjelmointikieleen; Javaan, C++:aan ja Pythoniin, sekä yleisesti käytössä olevaan JavaScript kehykseen Node.js:ään. Työn tilaaja oli kajaanilainen KajaPro Oy.

Go:n lyhyen, mutta sitäkin tapahtumarikkaamman historian aikana, Go on vakiinnuttanut oman paikkansa verkko- ja palvelinkehityksen parissa. Go:n kehittäjät Rob Pike, Ken Thompson ja Robert Griesemer kehittivät Go:n vastauksena Googlella todettuihin tarpeisiin helposti opittavaan, tehokkaaseen ja rinnakkaisuutta hyödyntävään kieleen. Go:ssa on yhdistetty useiden vanhempien kielien ominaisuuksia. Näitä ominaisuuksia ovat esimerkiksi: C-kieliperheestä tuttu ilmaisusyntaksi, Pascalin deklaraatiosyntaksi ja BCPL-kielen puolipistesääntö. Go on hybridikieli, jossa yhdistyy käännettyjen kielien tehokkuus ja tulkittujen kielien helppous.

Erilaisia käyttötapauksia tarkasteltaessa todettiin, että monet yritykset ovat vaihtaneet osittain tai jopa kokonaan Go:n käyttöön varsinkin verkko- ja palvelinkehityspuolella, joissa Go:n sisäänrakennettu rinnakkaisuus on tehokkaimmillaan. Suuremmat yritykset ovat huomanneet, että varsinkin Go:n yksinkertaisuus on tuonut tehokkuutta niin yksittäisten kehittäjien, kuin kehittäjätiimienkin sisällä. Tätä selitetään Go:n helpolla omaksuttavuudella, sekä sen kiinteällä ja yksiselitteisellä syntaksilla, jolloin kaikki kehittäjät kirjoittavat lähes identtistä koodia. Go:n vahvuuksiksi lueteltiin yllä mainittujen seikkojen lisäksi asioita, kuten Googlen tuki kielen kehityksessä, Go:n monialustatuki, joka helpottaa Go:lla kirjoitettujen ohjelmistojen julkaisua useilla alustoilla samanaikaisesti, sekä muistinsiivous, joka vähentää kehittäjän harteille kasautuvaa taakkaa, kun Go hoitaa muistinsiivouksen automaattisesti.

Vertailtaessa Go:ta perinteisiin ohjelmointikieliin Python, Java ja C++, sekä Node.js kehykseen huomattiin, että Go on erittäin tehokas käännetty kieli. Vaikka Go ei pärjää C++:alle synteettisissä koetintesteissä, Go:n vahvuudet sen yksinkertaisuuden kautta toivat kehittäjien silmissä sille lisäarvoa. Tulkittuihin kieliin kuten Pyhton ja Java verrattuna Go todettiin moninkertaisesti tehokkaammaksi koetintestien avulla. Go:n suurimpina heikkouksina vertailuissa havaittiin sen kehittymättömät työkalut verrattuna jo kymmeniä vuosia käytössä olleisiin kieliin.

Työn tuloksena todettiin, että vaikka Go onkin nuori kieli, on se lyhyessä ajassa saanut kurottua umpeen kymmenien vuosien kehityksen verrattuna vanhempiin kieliin. Varsinkin palvelin- ja verkkokehityksen puolella Go:n sisäänrakennetusta rinnakkaisuudesta on eniten hyötyä. Käytetty data näyttää, että Go on äärimmäisen tehokas ja nopeasti opittava kieli. Työssä tehtyjä huomioita havainnollistettiin yksinkertaisella teknisellä demolla. Demon oli tarkoitus toimia esimerkkinä siitä, kuinka helppoa Go:n avulla on kirjoittaa mikropalveluita kokemattomanakin ohjelmoijana.

KAMK • University of Applied Sciences

Abstract

Author(s): Rantasalo Roope

Title of the Publication: Go programming language's syntax and performance

Degree Title: Bachelor of Engineering, Information and Communications Technology

Keywords: Go, programming languages, programming language syntax, programming paradigms, back end, web development

This engineering thesis dived into the syntax and paradigms of Go and sought to find the most important observation on the choices that had been made during the development of Go. These notices include concepts such as: how Go is written and read and how does Go perform. In addition, there are examples of what different enterprises and businesses have done with Go and why they choose Go over other languages. Doing this also helped recognize the strengths and weaknesses of Go. To support these findings, Go was compared to three traditional and widely used programming languages Java, C++ and Python as well as a well-known JavaScript framework Node.js.

During the short but eventful history of Go it has consolidated itself in both web and server development. Go was developed by Rob Pike, Ken Thompson and Robert Griesemer to answer the internal need at Google to have an efficient and concurrent yet easy-to-learn programming language. Go aims to combine some of the best features of other languages such as C-language's expression syntax, Pascal's declaration syntax and BCPL-language's semicolon rule. Go is a hybrid language that combines the efficacy of compiled languages and the ease of interpreted languages.

Studying a scale of different use cases for Go it was noted that multiple companies and enterprises had transferred some or at some cases all their codebases to use Go. This was true especially in web and server development. Some of the bigger companies had also noticed that the ease of writing and understanding Go code was a huge deal in larger projects where multiple developers worked on the same project simultaneously. In addition to the beforementioned facts Go was praised for its cross-platform support, which makes it easy to deploy Go programs on multiple platforms. One of the biggest debate causes was Go's garbage collection, which is almost latency free, which makes it great for some projects and terrible for others. Garbage collection in Go cannot be switched off and Go only offers one type of GC. For an unexperienced developer GC makes it easier to write code as they don't need to keep track of memory.

While comparing Go to three older and widely used languages Python, Java and C++ as well as Node.js framework it was seen that Go is efficient compiled language. Even when Go loses convincingly to C++ in most of the benchmarks used, Go is still praised as a fast, compiled language. Comparing Go to interpreted languages such as Java and Python shows how big the difference is between the two styles. Go gains praise for its simplicity compared to C++ and effectiveness compared to Java and Python. The biggest downsides of Go were almost all related the underdevelopment of development tools compared to the older languages. Node.js and Go had similar result in benchmark but Node was usually chosen by developers for its capability of handling front end development which Go is not built for in addition to back end development.

The results of this study suggested that even though Go is a young programming language it has closed the gap significantly between itself and other widely used languages, especially in server and web development where Go´s unrivalled concurrency is at its best. According to the data gathered from developers, Go is an extremely efficient and easy-to-learn language. To illustrate the points made in this thesis a small technical demo was included to show how easy it is to write a microservice with Go even with little experience with Go or microservices.

Sisällysluettelo

1 Johdanto 1

2 Go 2

2.1 Go:n historia 2

2.2 Go ohjelmointikielenä 6

2.2.1 Syntaksi ja käytänteet 7

2.2.2 Tärkeimmät ominaisuudet 15

3 Go:n käyttötapaukset 19

3.1 Esimerkkejä Go:lla tehdyistä projekteista 19

3.2 Go:n kirjastot ja kehykset 22

3.3 Go:n vahvuudet ja heikkoudet 24

4 Go:n vertailu perinteisempiin ohjelmointikieliin 27

4.1 Go verrattuna Node.js:ään 27

4.2 Go verrattuna Pythoniin 28

4.3 Go verrattuna Javaan 30

4.4 Go verrattuna C++:aan 31

4.5 Kielten vertailun yhteenveto 32

5 Teknisen työn toteutus 34

6 Pohdinta 42

7 Yhteenveto 43

Lähteet 45

Symboliluettelo

API = Application Programming Interface. Ohjelmointirajapinta, määrittää sen, miten ohjelmiston muut osat voivat käyttää rajapintaan sidottua komponenttia.

Append-metodi = Go:n slice-tietotyypin sisäänrakennettu metodi, jolla lisätään sliceen elementtejä.

BCPL-ohjelmointikieli = Basic Combined Programming Language. Vuonna 1967 ilmestynyt kieli, josta on otettu mallia kielissä, kuten C ja Go.

Defer = Suomeksi lykätä. Go:n oma avainsana, jolla voidaan lykätä funktiokutsuja funktioiden sisällä.

DNS = Domain Name System. Nimeämiskäytäntö verkkoon yhteydessä oleville tietokoneille ja palveluille.

Docker = Suurin kontituspalveluita tarjoava yritys. Kontitus on tapa, jossa yksi suurempi palvelu tai ohjelma virtuaalisesti jaetaan useiksi pieniksi ja itsenäisiksi osiksi.

Dynaaminen kieli = engl. Dynamic Language. Ohjelmointikieli, joka voi suorittaa osan operaatioistaan ajon aikaisesti.

Gorutiini = engl. Goroutine. Go:n äärimmäisen kevytrakenteiset säikeet, jotka mahdollistavat Go:n tehokkaan rinnakkaisuuden.

GUI = Graphical User Interface. Graafinen käyttöliittymä.

Hajautetut järjestelmät = engl. Distributed systems. Järjestelmät, joiden osat tai komponentit ovat erillään toisistaan. Ne keskustelevat keskenään eräänlaisessa verkostossa.

IDE = Integrated Development environment. Kehitysympäristö, joka sisältää kattavan valikoiman ohjelmistokehityksessä tarvittavia työkaluja (esimerkiksi Microsoft VisualStudio).

JSON = JavaScript Object Notation. Yleisesti datan säilömiseen käytettävä tiedostomuoto

Järjestelmäkieli = Yleisesti monikäyttöinen ohjelmointikieli, joka on suunniteltu järjestelmien, kuten käyttöjärjestelmien kehittämiseen.

Kanava = engl. Channel. Go:n tapa mahdollistaa Gorutiinien välinen kommunikaatio.

Kokonaisluku = engl. Integer. Tietotyyppi, joka sisältää kokonaislukuarvoja.

Käännetty kieli = engl. Compiled Language. Ohjelmointikieli, joka täytyy kääntää lähdekoodista konekoodiksi ennen ajoa. Esimerkiksi Go, C++.

LIFO = Last in First Out. "Viimeisenä sisään, ensimmäisenä ulos". Tietorakenteiden, kuten slicen ja taulukon tapa käsitellä uusia elementtejä. Uudet elementit menevät jonon kärkeen.

Liukuluku = engl. Floating point number eli float. Tietotyyppi, joka sisältää reaalilukuarvoja.

Merkkijono = engl. String. Tietotyyppi, joka sisältää jonossa olevia peräkkäisiä merkkejä.

Rinnakkaisuus = engl. Concurrency. Mahdollistaa useiden eri operaatioiden samanaikaisen suorituksen ohjelman sisällä.

Slice = Go:ssa käytettävä dynaaminen taulukko-tietotyyppi.

SQL = Structured Query Language. Tietokantojen sisäiseen kommunikaatioon kirjoitettu kieli.

Säännöllinen lauseke = engl. Regular expression. Hakurakenne, jonka avulla voidaan etsiä tiettyjä osia esimerkiksi käyttäjän syötteestä.

Tietorakenne = engl. Struct. Tietotyyppi, joka määrittää ryhmitetyn listan muuttujia

TIOBE = The Importance of Being Earnest. Indeksi ohjelmointikielille, jossa vertaillaan hakukoneiden tulosten lukumäärää, jotka sisältävät haetun ohjelmointikielen nimen.

Tulkittu kieli = engl. Interpreted Language. Ohjelmointikieli, jota ei käännetä konekielelle, vaan se voi suorittaa suurimman osan operaatioistaan suoraan lähdekoodista.

UTF-8 = Unicode Transofrmation Format. Rob Piken ja Ken Thompsonin kehittämä koodausjärjestelmä (engl. encoding). Käytössä yli 90% kaikista verkkosivuista.

Väliohjelmisto = engl. Middleware. Rajapintana erilaisten ohjelmistojen välillä toimiva ohjelmisto.

Esimerkkikoodit

1. Go-ohjelman pohjapiirros
2. Muuttujien alustus
3. Muuttujan arvon asetus
4. Esimerkki defer avainsanan käytöstä
5. Slicen ja taulukon alustus ja muokkaaminen
6. Tyhjän interfacen käyttäminen
7. Esimerkki interfacen käytöstä
8. Esimerkki virheiden käsittelystä
9. Esimerkki gorutiinien ja kanavien käytöstä. Ajojen 1-3 tulostukset näkyvät rinnakkain
10. File_Microservicen main.go-tiedoston sisältö
11. HTTPServe-metodi httpservice.go-tiedostossa
12. File-tietotyyppi ja fileList-tietokanta
13. ToJSON- ja FromJSON-funktiot
14. AddFile-funktion toiminta
15. DeleteFile- ja RemoveIndex-funktioiden toiminta
16. GetFile-funktion toiminta

1 Johdanto

Työn toimeksiantaja KajaPro Oy on kajaanilainen ohjelmistoalan yritys, joka tarjoaa ohjelmistosuunnitteluun ja -kehitykseen alihankintaa ja valmiita ratkaisuja. KajaPro haluaa teoriapainotteisen selvityksen Googlen Go-ohjelmointikielestä ja sen käyttötapauksista. Go:sta ei nuoren ikänsä takia ole tehty Suomessa AMK-tutkimustyötä, joten tämä työ on siinä mielessä uniikki, että tehdään pioneerityötä.

Ohjelmointikielien kehittäminen ei ole yksinkertaista tai helppoa, mikä on suurin syy sille, miksi uusia ohjelmointikieliä ilmestyy harvoin. Kun uusia kieliä kehitetään, on tärkeää pitää kielen tarkoitus koko kehityksen ajan ydinasiana. Suurin osa käytetyimmistä kielistä, kuten Java, C-perheen kielet, Python ovat jo kymmeniä vuosia vanhoja. Vaikka kieliä päivitetään, on kielten pohja kuitenkin rakennettu kytköksiin vanhemman teknologian, kuten yksiytimisten prosessorien tai alhaisten muistitaajuuksien kanssa. Näitä perustavalaatuisia ongelmia ei voida korjata päivittämällä. Uudet kielet, kuten Go ja Rust poimivat parhaita paloja vanhoista kielistä. Lisäksi ne on kehitetty vastamaan moderneja kehitysympäristöjä. jotta kehittäjät saavat mahdollisimman paljon irti nykyajan moniydin prosessoreista ja prosessoriklustereista.

Ohjelmointikielen valinta on olennainen osa sovelluskehitystä. Ja ohjelmiston suunnitteluvaiheessa onkin tärkeää määrittää, mitä työkaluja käytetään. Väärän työkalun valinta johtaa usein turhiin kuluihin ja vaikeasti ylläpidettävään koodikantaan. Oikean työkalun valinta taas helpottaa kehitystyötä ja tukee kehitettävän ohjelmiston rakennetta sisäisesti erilaisten ominaisuuksien avulla. Go:n tapauksessa näitä ominaisuuksia ovat muun muassa rinnakkaisuus ja se, että Go on käännetty kieli.

Työn tavoitteena on syventyä Go-ohjelmointikieleen, sekä sen vahvuuksiin ja heikkouksiin että syntaksiin . Työssä keskitytään Go:n ominaisuuksiin sekä esimerkkien kautta Go:n käyttötarkoituksiin. Työ sisältää teoriatasolla Go:n vertailua muiden perinteisempien ja vanhempien ohjelmointikielien välillä. Teoriaa tukemaan kehitetään pieni tekninen demo, jota toimeksiantaja voi käyttää esimerkkinä tulevaisuudessa.

2 Go:n taustaa

Go on Googlen kehittämä monikäyttöinen ja moniparadigmainen avoimen lähdekoodin ohjelmointikieli, jonka kehitys alkoi vuonna 2007 Rob Piken, Ken Thompsonin ja Robert Griesemerin toimesta. Kaikki kolme ovat Googlen kehitysinsinöörejä, jotka kyllästyivät perinteisiin valtakieliin, kuten C++, Java ja Python, joista on vuosien saatossa kasvanut valtavia kirjastoriippuvuuksia vaativia kieliä. He halusivat kehittää yksinkertaisen, tehokkaan ja turvallisen ohjelmointikielen, joka on nopea kirjoittaa ja kääntää. [1.] [2.]

2.1 Go:n historia

Go on ohjelmointikielien mittakaavassa verrattain nuori kieli. Sen ensimmäinen julkaisu tapahtui marraskuussa 2009, kun taas esimerkiksi C-kielen ensimmäinen julkaisu oli vuonna 1972. Go:n kehittäjillä on valtava määrä kokemusta ohjelmointikielistä ja niiden kehityksestä monien vuosikymmenten ajalta, mikä heijastuu Go:n rakenteeseen ja toiminnallisuuteen.

Rob Pike tunnetaan Go:n lisäksi parhaiten kontribuutiostaan Unix- , Plan 9- ja Inferno-käyttöjärjestelmien kehityksessä sekä UTF-8-formaatin suunnittelusta yhdessä Ken Thompsonin kanssa [3].

Ken Thompson on kehittäjistä varmasti tunnetuin, sillä hän toimi yhtenä pääkehittäjistä Unixin, UTF-8 formaatin, Plan 9:n ja Infernon kehityksessä. Thompson on myös kehittänyt C-kielen edeltäjän B-ohjelmointikielen [4].

Robert Griesemer tunnetaan parhaiten Go:n kehityksestä, mutta hän on ollut mukana kehittämässä esimerkiksi kääntäjää Java HotSpot -virtuaalikoneelle [5].

Koska Go on niin uusi kieli, voitiin sen kehityksessä yhdistellä monien muiden kielien ominaisuuksia ja välttää aikaisempien kielien virheitä. Piken mukaan Go:n kehitykseen eniten vaikuttaneita kieliä ovat C-kieli, josta tärkein lainaominaisuus on Go:n ilmaisusyntaksi. Modula2-kielen pakettien käyttö, Pascalin deklaraatiosyntaksi, BCPL-kielen puolipistesääntö ja Smalltalkin metodit ovat hyviä esimerkkejä lainatuista ominaisuuksista. Go:ta varten kehitettyjä ominaisuuksia ovat muun

muassa *defer*-käytäntö (suom. lykätä) ja vakiomuuttujat. Näihin ja muihin ominaisuuksiin syvennyn tarkemmin luvussa Go:n käyttötapaukset [1.] [2.]

Kuvissa 1 ja 2 näkyy Go:n kehitystä vuosi ennen julkaisua ja kuukausi julkaisun jälkeen. Suurimpina huomioina julkaisun jälkeen pakettien käyttäminen, puolipisteiden puuttuminen, sulkujen käyttö sekä muuttujien ja funktioiden näkyvyyden merkkaaminen sen alkukirjaimen koon mukaan.

hello.go, June 6, 2008

```
package main

func main() int {
    print "hello, world\n";
    return 0;
}
```

Kuva 1. Go:n syntaksi vuonna 2008

hello.go, Dec 11, 2009

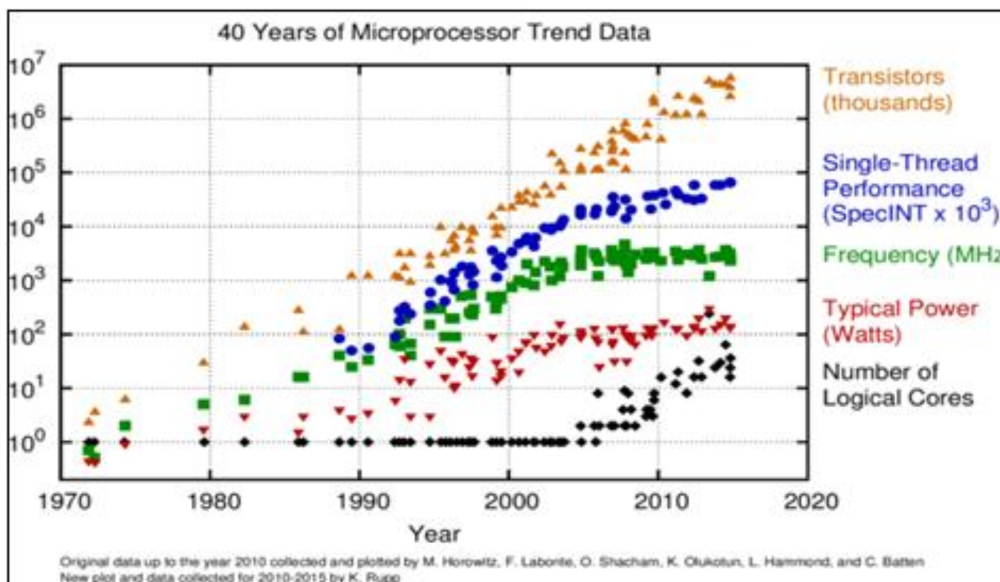
```
package main

import "fmt"

func main() {
    fmt.Printf("hello, world\n")
}
```

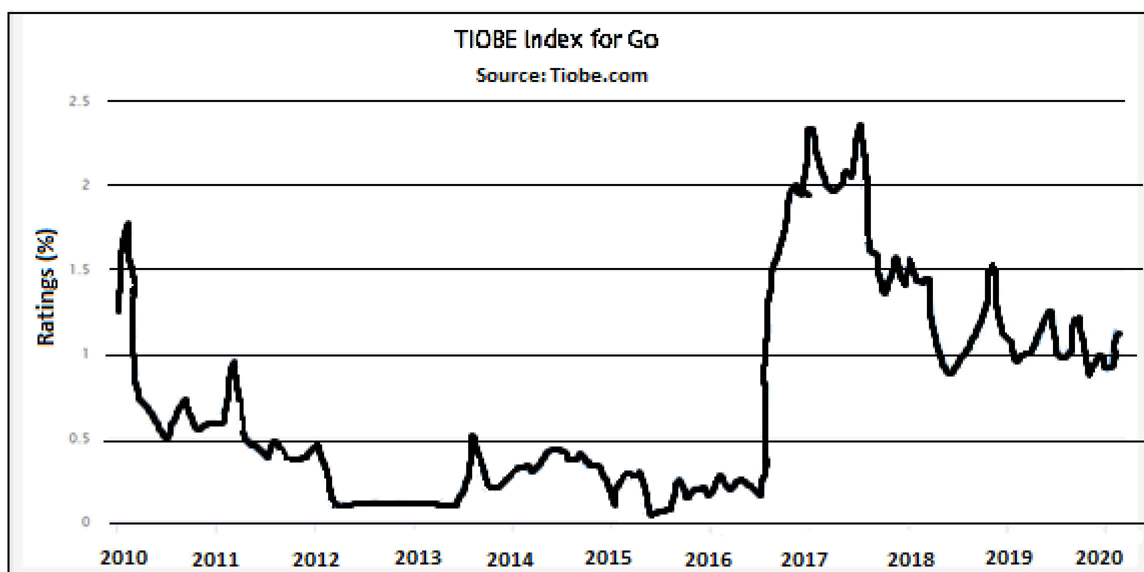
Kuva 2. Go:n syntaksi kuukausi julkaisun jälkeen

Go:sta haluttiin tehdä eräänlainen hybridikieli, jossa yhdistyisi perinteisten käännettyjen kielien, kuten C:n ja C++:n tehokkuus, ja nopeus, sekä dynaamisten kielten, kuten Javan ja Pythonin helppous. Go:n kehittäjät tajusivat, että perinteiset ohjelmointikielet, kuten C++, Python ja Java, on kehitetty oman aikansa laitteistojen pohjalta, jolloin yksiytimiset prosessorit olivat ainoa vaihtoehto. Juuri Go:n kehityksen alkuvaiheessa vuonna 2007 ja siitä eteenpäin alkoi prosessorien ytimten määrä kasvaa, kuten kuvasta 3 huomataan. Ohjelmien rinnakkaisuus, josta kerrotaan myöhemmin lisää luvussa 2.2.2, onkin Go:n tärkein ominaisuus. [1.]



Kuva 3. Mikroprosessorien kehitys vuosikymmenten aikana [6]. Ydinten lukumäärä merkattu mustalla alimpana.

Käyttäen esimerkkinä TIOBE:n tietokantaa, joka kattaa yli 150:n eri ohjelmointikielen tilastotietoja perustuen 25:n eri hakukoneen kautta haettuun tietoon [7]. Kuvasta 4 huomataan, että Go:n suosio on ollut vaihtelevaa koko sen olemassaolon ajan. Tästä huolimatta Go oli TIOBE:n ”vuoden ohjelmointikieli” vuosina 2009 ja 2016, mikä tarkoittaa, että kaikista mitatuista kielistä Go nousi tilastossa eniten kyseisinä vuosina.



Kuva 4. Go:n suosio vuosina 2010–2019 TIOBE:n mukaan [8]

Toinen alue, jossa Go:n suosiota voidaan vertailla muihin kieliin, on GitHub versionhallintapilvestä löytyvät varastokirjastot (engl. *repositories*). Vuodesta 2014 eteenpäin ylläpidetty GitHut 2.0 projekti [9] pitää kirjaa tietyllä kielellä merkittyjen varastokirjastojen määrästä. Taulukoista 1 ja 2 huomataan, että Go-varastokirjastojen määrä on vuosien 2014 ja 2019 välillä lähes kolminkertaistunut. Go:n suosio on merkittävää etenkin, kun otetaan huomioon se, että viidestä suosituimmasta kielestä kaikki muut ovat vähintään 24 vuotta vanhoja.

Taulukko 1. Vuoden 2014 tilaston suosituimmat kielet GitHutissa [9]

# Sija	Ohjelmointikieli	Prosentuaalinen osuus
1	JavaScript	20,891 %
2	Python	15,674 %
3	Ruby	12,831 %
4	Java	10,387 %
5	PHP	9,997 %
6	C++	6,262 %
7	C++	3,919 %
8	Go	3,048 %
9	C#	2,305 %
10	Shell	1,921 %
11	Objective-C	1,472 %
12	Scala	1,312 %
13	CoffeeScript	1,229 %
14	Rust	1,078 %
15	Pers	0,794 %

Taulukko 2. Vuoden 2019 tilaston suosituimmat kielet GitHutissa [9]

# Sija	Ohjelmointikieli	Prosentuaalinen osuus
1	JavaScript	20,266 %
2	Python	17,577 %
3	Java	10,177 %
4	Go	8,287 %
5	C++	6,868 %
6	Ruby	6,802 %
7	TypeScript	6,249 %
8	PHP	5,279 %
9	C#	3,629 %
10	C	2,952 %
11	Shell	1,979 %
12	Scala	1,653 %
13	Rust	0,935 %
14	Swift	0,815 %
15	Dart	0,745 %

2.2 Go ohjelmointikielenä

Go on C-kieleen pohjautuva monikäyttöinen järjestelmäkieli, jossa on yhdistelmä monia käännettävien ja dynaamisten kielten ominaisuuksia. Go:n kehittäjät halusivat kielen, jonka käyttö helpottaisi kehitystä varsinkin Googlen mittakaavalla tehdyissä projekteissa, joissa voi työskennellä jopa satoja tai tuhansia insinöörejä kerralla. Päästäkseen haluttuun tulokseen Go on pysynyt äärimmäisen tiiviinä kokonaisuutena koko elinkaarensa ajan, eikä siihen ole lisätty uusia ominaisuuksia, ellei kaikki kolme pääkehittäjää ole puoltaneet lisäystä. Go:ssa on vain 25 avainsanaa [10] (vrt. C++ 80+ avainsanaa [11] ja Java 51 avainsanaa [12]) ja ytimekäs sekä siisti syntaksi, mikä ohjaa käyttäjiä kirjoittamaan siistiä, ytimekästä ja helppolukuista koodia.

Yksi tärkeimpiä perusperiaatteita Go:n suunnittelussa on sen yksisuuntaisuus. Koko Go:n konsepti on suunniteltu yksisuuntaisuuden ympärille ja Pike toteaakin, että tarkoituksena kehityksessä oli saada mahdollisimman vähällä määrällä ominaisuuksia mahdollisimman paljon aikaan. [1.]

Go on kielenä vakiintunut ja yksinkertainen. Sen rajattu, mutta erittäin käytännöllinen standardikirjasto antaa työkalut lähtökohtaisesti mihin tahansa ilman mitään ylimääräisiä riippuvuuksia. Standardikirjasto sisältää paketteja, joita tuomalla (engl. *import*) saadaan käyttöön kyseessä olevan paketin sisältö. Paketeista löytyy työkalut esimerkiksi erilaisiin salauksiin, verkko-ohjelmistojen kehitykseen, yksikkötestaukseen ja tietokantojen kanssa työskentelyyn. Standardikirjaston lisäksi Go:n käyttäjä- ja kehittäjäyhteisö on rakentanut monia erilaisia paketteja helpottamaan esimerkiksi virheenkäsittelyä tai tunnistautumista ja oikeuttamista.

Kuten Pike toteaa, C ja C++ -kielien riippuvuuskäytännöt eivät mahdollista optimaalisia käännösaikoja, koska niissä voidaan sisällyttää (engl. *include*) samoja tiedostoja useissa muissa tiedostoissa, synnyttäen eräänlaisen verkoston riippuvuuksista, joita ei välttämättä edes tarvittaisi [13]. Go taas ottaa täysin erilaisen lähestymistavan riippuvuuksiin, hakemalla ainoastaan tarvittavat tiedot kohdetiedostosta, ei koko tiedostoa. [14.]

Havainnollistetaan tätä lyhyellä esimerkillä, jossa oletetaan Go-ohjelma, johon on lisätty kolme pakettia: A, B ja C.

- Paketti A sisältää paketin B.

- Paketti B sisältää paketin C.

Vaikka paketti A sisältää B:n, joka sisältää C:n, ei paketti A sisällä paketti C:tä. Riippuvaiset paketit on käännettävä ennen niistä itsestään riippuvaisia paketteja. Kun esimerkkiohjelma käännetään, ensin käännetään paketti C, ja sitten B ja A edellä mainitussa järjestyksessä. Pakettien tuomisen haluttiin olevan joustavaa, joten *import* ottaa argumenttinsa merkkijonona, mikä mahdollistaa pakettien helpon lataamisen esimerkiksi GitHub-varastokirjastoista. Tästä ominaisuudesta kerrotaan enemmän luvussa 3.

2.2.1 Syntaksi ja käytänteet

Go:n syntaksi pohjautuu C-kieleen, mutta on kuitenkin saanut vaikutteita muistakin kielistä. Koska Googlella suurin osa kehittäjistä on nuoria ja heidän ohjelmointikokemuksensa on usein C tai C++. Go:n kehityksessä tärkeää, että sen syntaksi olisi mahdollisimman samankaltainen C-perheen kanssa, jotta uusien Go-käyttäjien olisi helppo omaksua se. Tässä luvussa käydään koodiesimerkkien avulla läpi olennaisimmat osat Go:n syntaksista ja sen eri käytännöistä.

Go-tiedoston pohjapiirros koostuu kolmesta osasta; paketeista, tuonneista ja itse koodista. Esimerkkikoodissa 1 näkyy Go-ohjelman pohjapiirros.

```
1  pacakage main
2
3  import(
4      "fmt"
5  )
6
7  func main(){
8      fmt.Println("Hello World")
9  }
```

Esimerkkikoodi 1. Go-ohjelman pohjapiirros

Esimerkkikoodissa 2 näkyy Go:n muuttujien nimeämiskäytäntö. Ensin annetaan muuttujan nimi, sitten tyyppi. Muuttujien arvot voidaan alustaa kahdella tavalla, ensimmäinen on perinteinen pidempi tapa, jossa käyttäjä alustaa itse muuttujan tyypin. Toinen on lyhyempi tapa, jossa muuttu-

jan tyyppi tulee implisiittisesti sen arvosta [15]. Nämä tavat on esitetty esimerkkikoodissa 3. Lyhyessä alustuksessa käytetään operaattoria ':=', joka on kopioitu Rob Piken kehittämästä Newsqueak-kielestä [1].

```
1
2  //Muuttujien alustus Go-kielessä
3  x int
4  p *int
5  a [3]int
```

Esimerkkikoodi 2. Muuttujien alustus

```
1   pacakage main
2
3   import("fmt")
4
5   func main(){
6       // Pidempi perinteinen tapa
7       var x int = 1
8       // Lyhempi implisiittinen tapa
9       y := 2
10  }
```

Esimerkkikoodi 3. Muuttujan arvon asetus. Molemmat muuttujat ovat int-tyyppisiä

Yksi Go:n avainsanoista on *defer. Defer*-avainsanan käyttö tarkoittaa, että lykättyä funktiota kutsutaan vasta, kun sen sisältävä funktio on suoritettu. *Deferin* käyttöä on havainnollistettu esimerkkikoodissa 4. *Defer* toimii LIFO-periaatteella, mikä tarkoittaa sitä, että jos *deferiä* käytetään useammin kuin kerran, ensimmäisenä koodissa oleva *defer*-funktio toteutetaan viimeisenä. *Deferiä* voidaan käyttää esimerkiksi varmistamaan, että kaikki avatut tiedostot suljetaan käytön jälkeen. [16, s. 143–148.] [17.] Esimerkissä 4 tulostetaan sanat Hei ja Maailma omille riveilleen.

```
package main

import "fmt"

func main(){
    // Defer avainsana lykkää Maailma-sanan tulostuksen main-funktion loppuun
    defer fmt.Println( a...: "Maailma")
    // Tämä rivi tulostuu siis ennen Mailmaa
    fmt.Println( a...: "Hei")
C:\Users\roope\go\src\esimerkit>go run esimerkki4.go
Hei
Maailma
```

Esimerkkikoodi 4. Esimerkki defer-avainsanan käytöstä

Maailman tulostus sijoitetaan ennen Hei:n tulostusta, mutta Maailman tulostuksessa käytetään *defer*-avainsanaa, jolloin se toteutetaan vasta *main*-funktion lopussa. Tämän takia sanat tulostuvat oikeassa järjestyksessä.

Go:n käyttämä *slice* on taulukon pohjalta rakennettu dynaaminen taulukko. Esimerkkikoodista 5 nähdään *slicen* ja taulukon alustuksen erot. *Slice* on dynaaminen tyyppi, joka varaa itselleen muistia käyttönsä mukaan, kun taas taulukoille varataan alustuksessa haluttu määrä muistia. [16, s. 84–93.]

Esimerkissä luodaan ensin kolmen elementin pituinen taulukko x ja *slice* z, jolle annetaan viisi arvoa. *Sliceen* z: lisätään kolme arvoa *append*-funktiolla, jolloin sen pituus kasvaa automaattisesti kahdeksaan. Taulukko x taas on staattinen, joten siihen ei voida *append*-funktiolla lisätä elementtejä, mutta olemassa olevia elementtejä voidaan muokata normaalisti.

```go
package main

import "fmt"

func main() {
    // Array yaatii pituuden määrittämisen alustuksesa
    var x = [3]int{1, 2, 3}
    // Slice on on dynaaminen ja varaa itselleen tarvittavan määran muistia
    var z = []int{1, 2, 3, 4, 5}
    fmt.Println( a...: "Tämä on array x, ennen muokkaamista:", x)
    fmt.Println( a...: "Tämä on slice z, ennen muokkaamista:", z)

    // Koska slice on dynaaminen on siihen heippo lisätä elementtejä
    z = append(z,  elems...: 6, 7, 8)
    // Array on staattinen, joten siihen ei voida lisätä uusia elementtejä
    // Ainoastaan elementtien muokkaaminen on mahdollista
    for i:=0; i < len(x); i++{
        x[i] = x[i]+3
    }

    fmt.Println( a...: "Tämä on array x muokkauksen jalkeen: ", x)
    fmt.Println( a...: "Tämä on slice z muokkauksen jalkeen: ", z)
}
C:\Users\roope\go\src\esimerkit>go run esimerkki5.go
Tämä on array x, ennen muokkaamista: [1 2 3]
Tämä on slice z, ennen muokkaamista: [1 2 3 4 5]
Tämä on array x muokkauksen jalkeen:  [4 5 6]
Tämä on slice z muokkauksen jalkeen:  [1 2 3 4 5 6 7 8]
```

Esimerkkikoodi 5. *Slicen* ja taulukon alustus ja muokkaaminen

Yksi Go:n puhutuimmista ominaisuuksista on sen tapa yhdistää tietotyyppejä tai objekteja toisiinsa ilman perintää, jota käyttävät suurimmat kielet, kuten Java, ja C++. Go käyttää perinnän sijaan niin kutsuttua muodostusmenetelmää (engl. *composition*), jota pidetään yleisesti monikäyttöisempänä kuin perintää. Molempien tapojen tarkoitus on lisätä koodin uudelleenkäytettävyyttä. Go käyttää tietotyyppien ja objektien yhdistämiseen *interfaceja* (suom. liitäntä), jotka ovat kokoelmia metodeja kyseiselle interfacelle. *Interfacejen* avulla on helppoa jakaa ainoastaan tarvittava tieto funktioiden ja metodien välillä. [18.]

Oletetaan *interface*, jolla on kolme metodia, joiden nimi ja tyypit ovat kokonaisluku, liukuluku ja merkkijono. Tätä *interfacea* voi käyttää siis kaikki tyypit, joilla on myös kokonaisluku-, liukuluku-

ja merkkijono-nimiset ja -tyyppiset metodit. Tätä havainnollistetaan esimerkkikoodin 7 avulla. *Interfaceja* voidaan käyttää myös tyhjänä eli ilman yhtäkään metodia. Tällöin kaikki tyypit voivat käyttää sitä. Esimerkkikoodissa 6 on esitetty tyhjän *interfacen* käyttäminen.

```
package main

import "fmt"

// Funktio, jolle annetaan parametrinä tyhjä interface.
// Funktio tulostaa annetun parametrin tyypin ja arvon
func funktio1(i interface{}){
    fmt.Printf("Tyyppi = #{i}, Arvo = #{i}\n")
}

func main(){
    // Merkkijono-tyyppinen mmuuttuja
    _string := "Hei Maailma"
    funktio1(_string)
    // Kokonaisluku-tyyppinen muuttuja
    _integer := 1234
    funktio1(_integer)
    // Tietorakenne-tyyppinen muuttuja
    _struct := struct {
        nimi string
    }{
        nimi: "Esimerkki Erkki",
    }
    funktio1(_struct)
}
C:\Users\roope\go\src\esimerkit>go run esimerkki6.go
Tyyppi = string, Arvo = Hei Maailma
Tyyppi = int, Arvo = 1234
Tyyppi = struct { nimi string }, Arvo = {Esimerkki Erkki}
```

Esimerkkikoodi 6. Tyhjän *interfacen* käyttäminen

Esimerkissä luodaan tyhjä *interface*-funktio, joka tulostaa annetun tyypin ja arvon. *Main*-funktiossa luodaan merkkijono-, kokonaisluku- ja tietorakenne-tyyppiset muuttujat ja asetetaan ne luodun funktion parametreiksi, jolloin nähdään, että funktio tulostaa tyypin ja arvon kaikille luoduille tyypeille oikein.

```
//Geometria interface, jolla metodit pintaala ja piiri
//Taten, kaikki tyypit, joilla on pintaala ja piiri metodit, voidaan soveltaa geometria inte
type geometria interface {
    pintaala() float64
    piiri() float64
}
// Nelikulmio tyyppi, jolla leveys ja korkeus
type nelikulmio struct{
    leveys, korkeus float64
}
// Ympyrä tyyppi, jolla sade
type ympyra struct {
    säde float64
}
//Funktio nelikulmion pinta-alan laskemiseen
func (n nelikulmio) pintaala() float64 {
    return n.leveys * n.korkeus
}
//Funktio nelikulmion piirin laskemiseen
func (n nelikulmio) piiri() float64 {
    return 2*n.leveys + 2*n.korkeus
}
// Funktio ympyran pinta-alan laskemiseen
func (y ympyra) pintaala() float64 {
    return math.Pi * y.säde * y.säde
}
// Funktio ympyran piirin laskemiseen
func (y ympyra) piiri() float64 {
    return 2 * math.Pi * y.säde
}

// Funktio, joka ottaa geometria-tyyppisen parametrin ja laskee sille pinta-alan ja piirin
func mittaa(g geometria) {
    fmt.Println( a...: "Leveys ja korkeus / säde", g)
    fmt.Println( a...: "Pinta-ala", g.pintaala())
    fmt.Println( a...: "Piiri", g.piiri())
}

 func main(){
        n := nelikulmio{leveys: 2, korkeus: 5}
        y := ympyra{säde: 5}
        // Sekä nelikulmio, etta ympyra sisältävät tarvittavat pintaala- ja piiri -metodit,
        // joten ne täyttävät geometriainterfacen ehdot.
        mittaa(n)
        mittaa(y)
    }
C:\Users\roope\go\src\esimerkit>go run esimerkki7.go
Leveys ja korkeus / säde {2 5}
Pinta-ala 10
Piiri 14
Leveys ja korkeus / säde {5}
Pinta-ala 78.53981633974483
Piiri 31.41592653589793
```

Esimerkkikoodi 7. Esimerkki *interfacen* käytöstä

Esimerkissä luodaan *geometria*-niminen *interface*, jolla on kaksi metodia pinta-alan ja piirin laskemiseen. Lisäksi luodaan kaksi uutta tyyppiä, nelikulmio ja ympyrä, ja molemmille funktiot pinta-alan ja piirin laskemiseen. Viimeiseksi luodaan funktio, joka ottaa parametrina *geometria*-tyypin ja tulostaa annetun tyypin mitat, pinta-alan ja piirin. *Main*-funktiossa luodaan nelikulmio ja ympyrä, koska molemmat tyypit täyttävät *geometriainterfacen* ehdot, voidaan niitä käyttää *mittaa*-funktion parametreina.

Virheiden käsittely on yksi ohjelmointikielien tärkeimmistä ominaisuuksista, sillä sen avulla ohjelmoija voi varmistaa ja parantaa koodin toimivuutta jo sen kirjoitusvaiheessa. Go:n tapa virheiden hallintaan pyrkii saamaan ohjelmoijan ajattelemaan mahdollisia virheitä ennen kuin niitä edes tapahtuu. Standardikirjaston virheidenhallinta on Go:ssa kuitenkin melko rajallinen ja vaatiikin suurimmassa osassa tapauksista käyttäjän kirjoittaman implementaation virheille [19]. Go:ssa virheet ovat arvoja siinä missä kokonaisluvut tai merkkijonot, mikä helpottaa niiden käsittelyä [20]. Virheiden käsittely vaatii kehittäjältä melko paljon ylimääräisen koodin kirjoittamista, koska lähes jokainen funktio kannattaa rakentaa ajatellen virheiden tarkastamista. [21.][22.]

Go:n *errors*-paketti sisältää toiminnallisuuden yksinkertaisten virheiden käsittelyyn. Virheiden käsittely tapahtuu yksinkertaisimmillaan if-ehtolauseella, ennen funktion varsinaista toteutusta, kuten esimerkkikoodista 8 nähdään.

```
import (
    "errors"
    "fmt"
)

// Funktiolla on kaksi paluuarvoa, int ja error
// errors.New rakentaa uuden error arvon ja antaa sille halutun viestin
func virheEsimerkki(arvo int)( int, error){
    if arvo == 123: -1, errors.New("tapahtui virhe kasitelaessa arvoa 123")
    return arvo, nil
}
// Omien tyyppien kayttaminen virheina on mahdollista, jos niihin
// Implementoidaan Error() metodi
// Luodaan oma tyyppi, Jolla on kaksi parametria argumentti ja virheviesti
type argumenttiVirhe struct{
    argumentti int
    virhe string
}

//Implementoidaan Error()-metodi luotuun tyyppiin
func (e* argumenttiVirhe) Error() string{
    return fmt.Sprintf("#{e.argumentti} - #{e.virhe}")
}

// Virheen paluuarvoksi annetaan pointteri luotuun argumenttiVirheeseen
//argumenttiVirheelle annetaan halutut parametrit
func virheEsimerkki2(arvo int)(int, error) {
    if arvo == 123 : -1, &argumenttiVirhe{arvo, "arvo ei kelpaa"} ↵
    return arvo, nil
}

// Käytetään luotuja funktioita ja käydään läpi slicen arvot
//ja tulostetaan virheilmoitus tarvittaessa
func main() {
    for _, i := range[]int{15,123}{
        if arvo, virhe := virheEsimerkki(i); virhe != nil {
            fmt.Println( a...: "funktio epäonnistui: " , virhe)
        }else{
            fmt.Println( a...: "funktiol onnistui:", arvo)
        }
    }

    for _, i := range[]int{15,123}{
        if arvo, virhe := virheEsimerkki2(i); virhe != nil{
            fmt.Println( a...: "funktio2 epaonnistui:", virhe)
        }else{ fmt.Println( a...: "funktio2 onnistui:", arvo) }
    }
}
C:\Users\roope\go\src\esimerkit>go run esimerkki8.go
funktiol onnistui: 15
funktio epäonnistui:  tapahtui virhe kasitelaessa arvoa 123
funktio2 onnistui: 15
funktio2 epaonnistui: 123 - arvo ei kelpaa
```

Esimerkkikoodi 8. Esimerkki virheiden käsittelystä

Esimerkissä luodaan kaksi funktiota, jotka tulostavat joko kokonaisluvun tai virheen. Virheen tulostuksessa käytetään kahta eri tapaa. Ensimmäisessä luodaan *errors.New()*-funktiolla vapaavalintainen merkkijono-tyyppinen virheviesti ja toisessa luodaan oma virhetyyppi, joka tulostaa virheellisen arvon ja virheviestin. Itse luotu virhetyyppi sisältää *errors*-paketin implementaation virheille.

2.2.2 Tärkeimmät ominaisuudet

Go:n tärkein tai ainakin puhutuin yksittäinen ominaisuus on sen tapa käsitellä rinnakkaisuutta (engl. *concurrency*) eli ohjelman kykyä käsitellä useita asioita samanaikaisesti. Rinnakkaisuus on ollut osana Go:ta kehityksen alusta asti. Vaikka ominaisuus ei olekaan täysin alkuperäinen idea, on se toteutettu Go:ssa muista olemassa olevista kielistä poikkeavalla tavalla. Go:ssa rinnakkaisuus haluttiin tehdä mahdollisimman helpoksi, ilman erilaisia muista kielistä tuttuja lukkoja, säikeitä tai esteitä, joiden käyttäminen sujuvasti vaatii kohtuullisen paljon ohjelmointikokemusta. *Gorutiinit* ovat Go:n tapa yksinkertaistaa rinnakkaisuuden tuomat ongelmat. Ne ovat rinnakkain ajettavia funktioita tai metodeja ja niitä voidaan ajatella erittäin halpoina versioina säikeistä. *Gorutiinit* suoritetaan lisäämällä funktion tai metodin eteen avainsana *go*, kuten esimerkkikoodissa 9 nähdään. *Gorutiineilla* on oma dynaaminen muistipino, joka kasvaa ja pienenee oman tarpeensa mukaan. Vaikka *gorutiinit* eivät itsessään ole säikeitä, ne limitetään tarpeen mukaan käyttöjärjestelmän säikeisiin. Tällä varmistetaan, että kaikki *gorutiinit*, joita voi olla satoja tuhansia yhdessä ohjelmassa, pysyvät ajossa. *Gorutiinit* myös käsitellään ajon aikana automaattisesti, eivätkä ne vaadi käyttäjältä lisähuomioita. [23.]

Gorutiinit tarvitsevat mahdollisuuden kommunikoida toistensa välillä. Tähän Go:sta löytyy ratkaisu kanavien muodossa. Kanavat ovat muuttujia, jotka voivat lähettää ja vastaanottaa arvoja esimerkiksi *gorutiineilta*. Kommunikaation lisäksi kanavat synkronoivat ohjelmaa automaattisesti, sillä lähettäjä ei voi lähettää, ellei vastaanottaja ole valmiina vastaanottamaan ja päinvastoin. Kanavien käyttöä havainnollistetaan esimerkkikoodissa 9. [23.]

```
// Funktio, joka lähettää kanavlle merkkijono-muuttujan
func funktio2 (s string, kanava chan string){
    kanava <- s
}

func main(){
    // Luodaan uusi merkkijon-kanava
    kanava := make(chan string)

    //Luodaan go avainsanalla gorutiinit

    go funktio2( s: "gorutiinit", kanava)
    go funktio2( s: "ovat", kanava)
    go funktio2( s: "erittäin", kanava)
    go funktio2( s: "mukavia", kanava)

    // Vastaanotetaan kanavalta tulevat merkkijono-muuttujat
    // ja asetetaan ne a,b,c,d muuttujiin
    a := <-kanava
    b := <-kanava
    c := <-kanava
    d := <-kanava
    // Tulostetaan arvot järjestyksessä a,b,c,d
    fmt.Println(a,b,c,d)
}
C:\Users\roope\go\src\esimerkit>go run esimerkki9.go
gorutiinit erittäin ovat mukavia
C:\Users\roope\go\src\esimerkit>go run esimerkki9.go
gorutiinit mukavia ovat erittäin
C:\Users\roope\go\src\esimerkit>go run esimerkki9.go
mukavia gorutiinit ovat erittäin
```

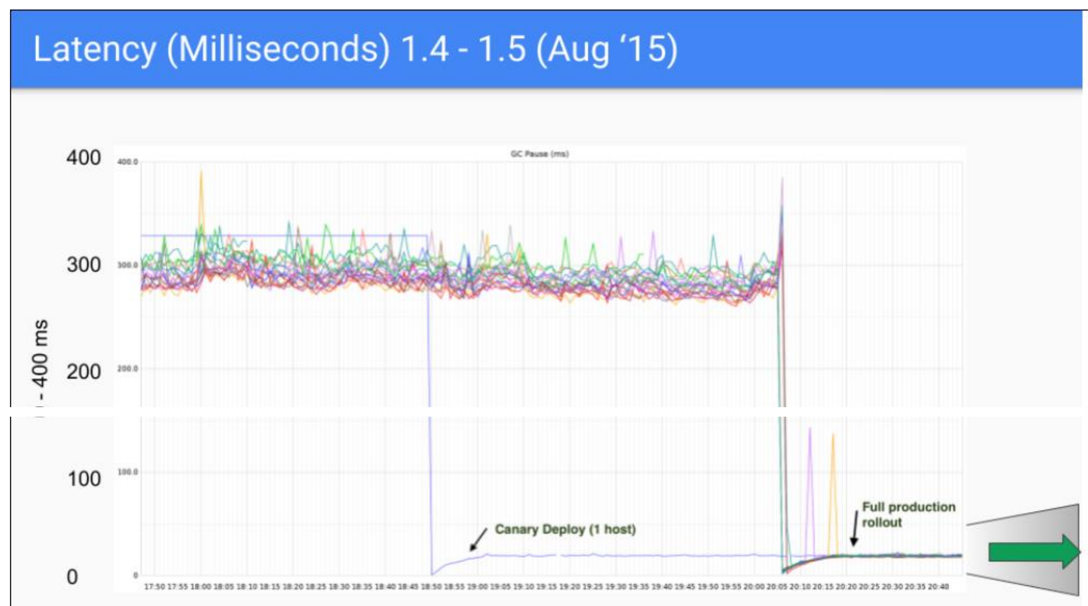
Esimerkkikoodi 9. Esimerkki *gorutiinien* ja kanavien käytöstä. Ajojen 1–3 tulostukset näkyvät allekkain

Esimerkissä luodaan funktio, joka ottaa parametrina merkkijono -muuttujan ja merkkijono-kanavan ja lähettää annetun muuttujan kanavalle ’<-’ operaattorilla. *Main*-funktiossa luodaan uusi merkkijono-kanava ja neljä *gorutiinia* aikaisemmin luoduille funktioille, joille annetaan parametreiksi merkkijono-muuttuja ja kanava. Lopuksi kanavalle lähetetyt arvot vastaanotetaan vuorollaan muuttujiin ja tulostetaan. Koska kanavien lähetys ja vastaanotto on oletuksena estetty, kunnes molemmat osapuolet ovat valmiita eli *gorutiineja* ei ole priorisoitu, on ajon välillä eroja siinä, missä järjestyksessä muuttujat tulostetaan.

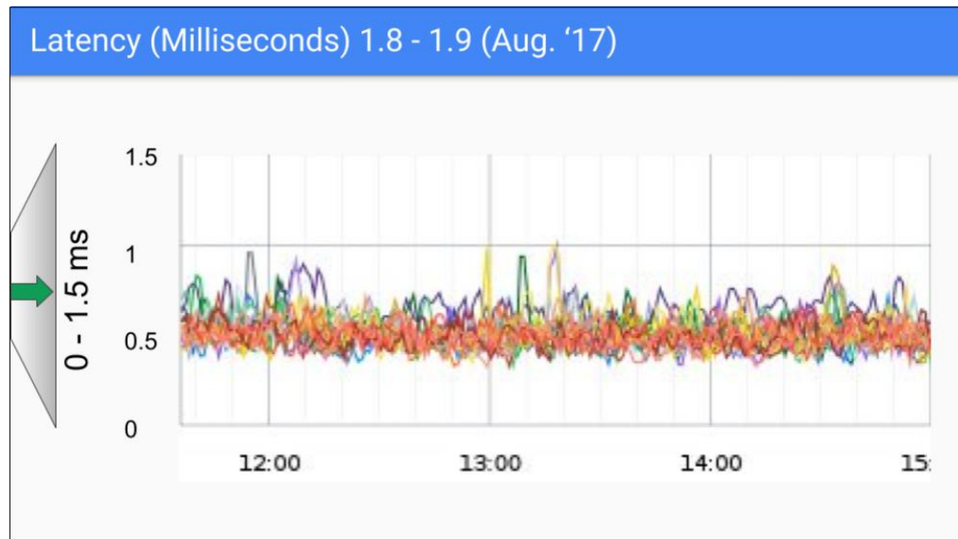
Go on niin sanottu muistisiivottu (engl. *garbage collected*) kieli, eli ohjelmoijan ei tarvitse huolehtia muistinkäytöstä, vaan se tapahtuu Go:ssa ajon aikaisesti samalla tavalla kuin aiemmin puhutut

gorutiinit. Go:n muistinsiivous on kehitetty mahdollisimman matalalatenssiseksi eli ohjelmien pysähdysajat siivouksen aikana ovat mahdollisimman lyhyitä. Matalan latenssin seurauksena on jouduttu karsimaan muistinsiivouksen toimintakyvystä, eli käyttämään siihen enemmän aikaa. [24.]

Go:n muistinsiivous sai täyden uudelleenkirjoituksen vuosien 2014 ja 2018 välillä, kun Googlella tajuttiin, että Go:n muistinsiivouksen latenssit ovat aivan liian korkeita ja ilman parannusta Go ei todennäköisesti tulisi pärjäämään kilpailulle [24]. Vuonna 2014 julkaistussa Go:n versiossa 1.4 latenssit olivat luokkaa 300–400 ms. Siitä asti joka vuosi julkaistiin uusi versio, joka vähensi latenssin noin kymmenesosaan edellisestä versiosta. Loppujen lopuksi vuonna 2017 julkaistussa versiossa 1.7 latenssit olivat enää alle 1 ms luokkaa.



Kuva 5. Go:n muistinsiivouksen latenssin kehitys vuonna 2015 [24].



Kuva 6. Go:n muistinsiivouksen latenssi vuonna 2017 [24].

Go:n aktiivisen yhteisön takia sillä on jo nyt suhteellisen kattava valikoima kolmansien osapuolien kirjoittamia ohjelmistokehyksiä, kirjastoja ja ohjelmistoja. Hyvä ja laaja lista olemassa olevista vaihtoehdoista löytyy esimerkiksi awesome-go.com sivustolta, jonne kerätään jatkuvasti uutta sisältöä. Listalta löytyy työkaluja esimerkiksi grafiikan piirtämiseen, kuvankäsittelyyn, erilaisten bottien rakentamiseen, laitteistojen kanssa työskentelyyn (mm. ARMv5,6,7,8 tuki Linuxille), pelikehitykseen ja tietokantojen käsittelemiseen. [25.] Go:n kirjastoihin ja kehyksiin syvennyn tarkemmin luvussa 3.2.

3 Go:n käyttötapaukset

Tässä luvussa käydään läpi Go:n hyviä ja huonoja puolia erilaisten esimerkkien avulla. Tarkoitus on saada kattava kuva siitä, mitä Go:lla on aikaisemmin tehty ja mitä mieltä käyttäjät ovat siitä olleet. Suurin osa työssä kartoitetuista yrityksistä ovat ottaneet Go:n käyttöön vuosien 2013 ja 2015 välillä, mutta koska Go:n perusajatus ei ole muuttunut viimeisien vuosien aikana, voidaan olettaa, että yritysten väitteet Go:n hyvistä ja huonoista puolista pitävät edelleen paikkansa. Kaikki esimerkit pohjautuvat puhuttavan yrityksen kirjoittamiin blogeihin, artikkeleihin tai esitelmiin.

3.1 Esimerkkejä Go:lla tehdyistä projekteista

Esimerkit on jaoteltu karkeasti viiteen kategoriaan; tietokannat, pilvipalvelut, kontit, yleinen ja skaalautuvuus. Jokaisesta kategoriasta löytyy vähintään yksi esimerkkiyritys ja tiivistelmä siitä, mitä yritys on kertonut Go:n käytöstä.

Tietokantapalveluita tarjoava Cockroach LABS aloitti avoimen lähdekoodin tietokantaprojektin CockroachDB:n kehityksen vuonna 2015 Go:lla, koska yritys koki hajautettujen järjestelmien kehitykseen sopivia kieliä oli tarjolla vain C++, Java ja Go. Java suljettiin pois mahdollisten suorituskykyongelmien takia ja C++ taas sen monimutkaisuuden takia. CockroachDB on suhteellisen suuri projekti, jossa on yli 125 000 riviä koodia, joten koodin pitäminen mahdollisimman yksinkertaisena on tärkeää varsinkin suuremmissa avoimen lähdekoodin projekteissa. Yksinkertaisuutta ja koodin luettavuutta tukee Go:n ytimekäs syntaksi ja tiukka formaatti. [26.]

Erilaisia pilvipalveluita tarjoava Iron omaksui Go:n käytön jo vuonna 2011 ensimmäisten suurempien yritysten joukossa. Iron tarjoaman IronWorker palvelun ensimmäinen versio oli kirjoitettu Rubyllä ja API rakennettu Railsillä (Ruby on Rails). IronWorkerin tarkoitus on helpottaa suurten tehtävien, kuten kuvankäsittelyn ja massadatan käsittelyä Docker-konttien avulla. [27.]

Ennen kun Go otettiin käyttöön, IronWorker käytti jatkuvasti yli 50% palvelimen suoritintehoista ja suoritintehon tarpeen kasvaessa täytyi ostaa lisää palvelimia, mikä on kallista. Lisäksi tietoliikenteen kasvaessa yllättäen kävi usein niin, että Rails-palvelimien suoritinkäyttö nousi nopeasti

100 %:iin, mikä teki niistä reagoimattomia, mikä taas johti seuraavan palvelimen kuormittamiseen ja dominoefektiin, jolloin koko palvelu saattoi kaatua. API kirjoitettiin kokonaan uudestaan Go:lla, sen rinnakkaisuuden, standardikirjaston ja käännösnopeuden takia. Uudelleenkirjoitettu API vähensi palvelimien tarvetta kolmestakymmenestä vain kahteen, ja olemassa olevien kahden palvelimen kuormitukset olivat alle 5 % luokkaa ja muistinkäyttö sovelluksen käynnistyksessä väheni noin viidestäkymmenestä megatavusta muutamaan sataan kilotavuun. [28.]

Pilvipalvelutarjoaja Cloudflare on tehnyt Go:lla jo useampia projekteja *back end* puolellaan. Se kirjoitti esimerkiksi oman DNS-välityspalvelimen rrDNS:n (Resource Record Domain Name System), joka yksinkertaisti palvelun tietokantahakuja vähentämällä edestakaisten hakujen määrää. Uusi järjestelmä laski DNS-hakuihin käytettävän ajan keskiarvoisesti kolmanteen vanhaan järjestelmään verrattuna. Go:n yksinkertaisuus helpottaa työtä ja vähentää kirjoitetun koodin monimutkaisuutta, mikä taas helpottaa koodin käsittelyä ja huoltoa jatkossa. Go:n rinnakkaisuus ja *gorutiinit* nousevat jälleen esille palvelinten välillä, kun hakuja voi olla miljoonia kerrallaan päällä. Cloudflare on kehittänyt myös Railgun nimisen pakkausteknologian, joka nopeuttaa verkkosivujen sisällön toimitusta (engl. *content delivery*). Käytännössä se nopeuttaa verkkosivujen latausta käyttäjän näkökulmasta. Teknologiana Railgun etsii sivustolta välimuistiin tallentumatonta dataa ja monitoroi sen muutoksia. Jos muutoksia on tapahtunut Railgun lähettää vain pakatut muutokset, ei koko sivua uudestaan pakattuna, mikä on pudottanut pakatun datan koon yleisesti käytössä olevista pakkausteknologioista vain murto-osaan. Railgun on saavuttanut testeissä jopa 80 % vähennyksen sivun lataamiseen käytettävään aikaan. [29.] [30.][31.]

Konttipalvelut pinnalle tuonut Docker on myös kirjoitettu Go:lla kokonaisuudessaan. Marraskuussa 2013 pidetyssä Googlen kehittäjien kokoontumisessa Dockerin kehittäjät pitivät esityksen siitä, miksi he valitsivat juuri Go:n Dockerin kehitykseen. Kehittäjät listasivat viisi tärkeintä syytä; (1) staattinen käännös, joka helpottaa asennusta ja testaamista, (2) Go on ”neutraali”, eli se ei ole samanlainen valtakielien C++, Python, Ruby ja Java kanssa, (3) Go:ssa on kaikki tarvittava, kattava standardikirjasto ja rinnakkaisuus ja niin edelleen (4) kattava kehitysympäristö, *go doc*; pakettien dokumentaatio helposti saatavilla, *go get*; pakettien lataaminen esimerkiksi GitHubista, *go fmt* yhtenäinen formaatti kaikelle koodille, *go test*; kaikkien testien ajaminen yhdellä komennolla ja *go run* nopeaan koerakentamiseen ja (5) monialustatuki ilman esikäsittelyohjelmia. [32.]

Kubernetes on Dockerin rinnalle rakennettu konttiklustereiden hallintajärjestelmä, joka mahdollistaa Docker-konttien ajamisen koneklusterissa. Kubernetesin kehittäjät pitivät mahdollisina kehityskielinä myös C:tä, C++:aa, Javaa tai Pythonia, mutta C:tä ja C++:aa he pitivät liian monimutkaisina kirjastoineen ja riippuvuuksineen kokemattomille käyttäjille. Java taas on liian hidas asentaa useille alustoille ja Pythonin dynaaminen kirjoitustyyli on iso riski järjestelmäohjelmistojen kehityksessä. Go:n hyvinä puolina listataan; (1) hyvät ja kattavat kirjastot, (2) äärimmäisen nopeat työkalut kehitykseen ja testaukseen, (3) Go on yksinkertainen, joten suurempien tiimien on helppo käsitellä samaa koodia, (4) hyödyllisiä muita työkaluja formatointiin, testaukseen ja virheiden etsintään, (5) Go:n rinnakkaisuus hajautetuissa järjestelmissä on iso etu, (6) C-tyylinen syntaksi, mutta lisäksi hyviä ominaisuuksia, kuten anonyymit funktiot, (7) muistinsiivous ja (8) tyyppiturvallisuus. [33.]

Tapahtuma- ja markkinointisähköpostipalveluita tarjoava SendGrid käytti aikaisemmin Perliä ja Pythonia, mutta erilaisten rajoitteiden takia sen oli pakko tehdä suurempia muutoksia ja vaihtoehdoiksi ohjelmointikielistä selvisi Scala, Java ja Go. Koska SendGridillä on paljon insinöörejä samojen projektien kimpussa, Scalan ja Javan dynaamisuus koettiin haitaksi, sillä uuden pätevän henkilöstön palkkaaminen on vaikeaa ja vanhan henkilökunnan koulutus aikaa vievää. SendGrid käsittelee päivittäin yli puoli miljoonaa viestiä, joten ohjelmointikielen asynkroninen rinnakkaisuus koettiin valtavaksi hyödyksi, joten Go oli yksinkertaisesti paras vaihtoehto. Suurimpina haittoina nähtiin Go:n vielä vuonna 2014 melko kehittymätön kirjastotuki, ja olemassa olevien ja kokonaan uusien kirjastojen korjaaminen ja kirjoittaminen vie paljon aikaa varsinaiselta ohjelmistokehitykseltä. [34.]

Verkossa toimiva musiikin ja äänen jakelupalvelu SoundCloud omaksui Go:n käytön myös jo varhaisessa vaiheessa vuonna 2012. SoundCloudilla on periaate, jonka mukaan se haluaa insinööriensä olevan mieluummin moniosaajia kuin yhden alan ammattilaisia. Go on ollut tähän täydellinen kieli, se on helppo oppia, koodi on helposti luettavissa, oli se kenen tahansa kirjoittamaa. SoundCloud on säästänyt paljon aikaa esimerkiksi koodikatselmuksissa, kun aikaa ei kulu turhiin teknisiin mielipiteisiin tyyleistä kirjoittaa koodia, kun kaikki kirjoittavat Go:n pakottamalla tavalla. [35.]

Maailman suurin videosuoratoistopalvelu Twitch on kirjoittanut myös vuosien varrella *back endiään* Go:lla. Twitchiä käyttää päivittäin miljoonia ihmisiä, joten palvelimien kuormitus voi olla äärimmäistä, mutta Go:n avulla Twitch on saanut palvelustaan hyvin horisontaalisesti skaalautuvan.

Twitch oli myös aktiivisesti mukana Go:n muistinsiivouksen kehityksessä vuosien 2014-2017 välillä, sillä vaikka Go oli muuten sopiva työkalu, sen valtavat latenssit muistinsiivouksessa aiheuttivat käyttäjille turhauttavia maailman pysähdyksiä jokaisessa muistinsiivoussykslissä (engl. *stop-the-world*). [36.]

3.2 Go:n kirjastot ja kehykset

Tässä luvussa esitellään käytetyimpiä Go:n yhteisön kirjoittamia kirjastoja, ohjelmistokehityspaketteja sekä kehyksiä, jotka helpottavat ja nopeuttavat kehitystyötä.

Googlen itse kehittämä Go *Cloud Development Kit* eli pilvikehityspaketti on 2018 julkaistu työkalupaketti, jonka tarkoitus on helpottaa pilvipalveluiden kehittämistä ja jakelua Go:lla. Se on rakennettu neutraaliksi, eli sitä voidaan käyttää useiden pilvipalvelutarjoajien kanssa. Työkalu tarjoaa yleiskäyttöisiä ohjelmointirajapintoja esimerkiksi MySQL- ja PostgreSQL-tietokantoihin yhdistämiseen, palvelimien pystyttämiseen ja monitoroimiseen ja ajonaikaisesti muuttuvien muuttujien käsittelyyn. [37.]

Gin Web Framework on täysin Go:lla kirjoitettu verkkokehitystyökalu, joka tarjoaa nopeutta ja laajentuvuutta väliohjelmistojen kautta. Sillä on suhteellisen kattava dokumentaatio ja esimerkkikirjasto kehittäjien puolesta. Sillä on samankaltainen ohjelmointirajapinta vanhemman Martini [38] nimisen kehyksen kanssa, mutta Ginin sanotaan olevan jopa 40 kertaa nopeampi. [39.]

Koska koodin testaaminen on niin olennainen osa Go:ta, on siihen myös kirjoitettu hyviä kirjastoja tukemaan standardikirjastoa. Testify on paketti, joka sisältää työkaluja esimerkiksi *mock*-objektien luomiseen ja vahvistukseen (engl. *assert*). Se on pieni, kevytrakenteinen ja helppokäyttöinen, joten se sopii hyvin kaikkiin projekteihin. [40.] Toinen tunnettu testityökalu on http-kuorman testaustyökalu ja kirjasto Vegeta, jonka avulla voidaan kuormittaa http-palveluita tasaisesti ja hallitusti. Vegetalla on hyvä käyttöopas GitHub-sivullaan, josta saa nopeasti käsityksen työkalun käyttötavoista [41].

GORM eli Go:lle rakennettu objekti-relaatio-mappaus-työkalu (engl. *object-relational-mapping*) on kattava kirjasto tietokantojen kanssa työskentelyyn Go:lla. Se sisältää monipuolisia ominai-

suuksia, kuten assosiaatiot, koukut (engl. *hooks*), SQL-rakentajan, automaattiset migraatiot ja lokityökalun. Lisäksi kaikille ominaisuuksille löytyy valmiit testit. GORM:in toinen versio on myös kehitteillä, mikä viittaa kehittäjän aktiivisuuteen ja yhteisön tarpeeseen. [42.]

Go:n geneerisyyden puutteen vuoksi Clipperhouse on kehittänyt koodintuottotyökalu Genin, jonka on tarkoitus implementoida geneerisyyden kaltainen toiminnallisuus Go-koodiin. Sen toiminta perustuu niin kutsuttuihin kirjoituskoneisiin (engl. *typewriter*). Nämä kirjoituskoneet sisältävät itse koodintuoton logiikan. Käyttäjä voi itse lisätä uusia kirjoituskoneita Go-paketteina, mikä tekee Genistä helposti laajennettavan. Genillä voidaan siis luoda kehityksen aikana komentorivin kautta koodia kehittäjän luomille tyypeille, ilman ylimääräisiä riippuvuuksia. [43.]

Verkkokehityksessä tunnistautuminen (engl. *authentication*) on äärimmäisen tärkeää tietoturvan kannalta. Authboss on monista moduuleista koostuva järjestelmä, jota käyttäjä voi muokata omaan käyttöönsä sopivaksi. Moduuleista löytyy toiminnallisuutta esimerkiksi OAuth2 tunnistautumiseen, salasanojen palautukseen sähköpostin välityksellä, kaksivaiheisen tunnistuksen erilaisin keinoin, perinteisen tietokannan kautta tunnistautumisen ja monia muita hyödyllisiä ominaisuuksia. Authboss ei vaadi väliohjelmistojen käyttöä paitsi tietyissä tapauksissa, kuten evästeiden ja istuntojen käsittelyyn sekä istuntojen vanhentumiseen. Sen pitäisi toimia sekä ilman verkkokehystä että minkä verkkokehyksen kanssa tahansa. [44.]

Nuoresta iästään huolimatta Go on aktiivisen yhteisönsä kautta kerännyt suhteellisen suuren määrän erilaisia kirjastoja ja kehyksiä helpottamaan kehitystyötä. Kuten yllä olevista esimerkeistä nähdään, suurin osa kaikista kirjastoista kehyksistä ovat yksityishenkilöiden kehittämiä ja ylläpitämiä. Tämä luonnollisesti vaikuttaa kirjastojen luotettavuuteen ja ajankohtaisuuteen. Kirjastoja käytettäessä kannattaakin ottaa selville, milloin sitä on viimeksi päivitetty ja onko se edelleen relevantti. Potentiaalisia Go:lle kehitettyjä kirjastovaihtoehtoja kartoittaessa tulikin vastaan useista aikaisemmin suosittuja kirjastoja, joiden kehittäjät ovat lopettaneet kirjaston ylläpitämisen syystä tai toisesta.

3.3 Go:n vahvuudet ja heikkoudet

Tässä luvussa tiivistetään aiempia lukuja mukaillen Go:n vahvuuksia ja heikkouksia. Taulukkoon 3 kerättiin yleisimpiä mielipiteitä Go:n käyttäjiltä. Taulukon arvot ovat satunnaisessa järjestyksessä. Seuraavaksi käydään läpi ensin vahvuudet ja sitten heikkoudet siinä järjestyksessä, missä ne ovat taulukossa.

Taulukko 3. Go:n käyttäjien ilmoittamia heikkouksia ja vahvuuksia Go:ssa [29] [31] [33] [34] [35]

Go:n vahvuudet ja heikkoudet	
Vahvuudet	Heikkoudet
Yksinkertaisuus ja luettavuus	Ikä
C-tyylinen syntaksi	Uuden henkilöstön löytäminen
Tyyppiturvallisuus	Virheiden käsittely
Googlen tuki	Turha monisanaisuus
Rinnakkaisuus	Geneerisyyden puute
Standardikirjasto	Reaaliaikaisten tehtävien käsittely
Testaaminen	Kirjastojen ja työkalujen puute
Monialustatuki	Muistinsiivous (ei vaihtoehtoja)
Muistinsiivous	Graafisen käyttöliittymän puute

Suurin osa vahvuuksista on mainittu jo aikaisemmin työssä, joten tämä osio on pidetty mahdollisimman tiiviinä. Go:n kirjoittaminen on tehty mahdollisimman helpoksi ja yksimuotoiseksi, jotta saadaan kaikki Go:n käyttäjät kirjoittamaan lähes identtistä koodia. Koska koodi on kaikkialla saman näköistä, on sitä helppo lukea ja tuoreempienkin kehittäjien on helppo nähdä, mitä koodi tekee. Yksinkertaisuuden lisäksi Go:n C-tyyppinen syntaksi itsessään helpottaa kehitystyötä, jos kehittäjällä on yhtään aikaisempaa kokemusta C-perheen kielistä. C:n tavoin Go on äärimmäisen tyyppiturvallinen kieli, mikä estää kehittäjää tekemästä typeriä virheitä, jotka monissa muissa kielissä menisivät kääntäjästä läpi. Koska Go on Googlen kehittämä, on sillä valtava tukiverkko Googlen puolelta ja Googlen halu kehittää Go:sta paras mahdollinen kieli heidän tarkoituksiinsa takaa, että sen kehitys on tasaista ja jatkuvaa. Rinnakkaisuus on Go:n valttikortti, sillä se on yksi harvoista moderneista kielistä, joihin rinnakkuus on sisäänrakennettuna. Rinnakkaisuus helpottaa

ohjelmistojen kehitystä moniydinprosessoreille, toisin kuin monissa muissa kielissä, joissa monisäikeistäminen voi olla erittäinkin monimutkaista. Go:n standardikirjasto on pienehkö, mutta sitäkin tehokkaampi ja sillä voidaankin tehdä lähes mitä tahansa ilman, että tarvitsee etsiä kolmansien osapuolien kirjoittamia liitännäisiä tai kirjastoja. Koodin testaamisesta on myös tehty mahdollisimman helppoa sisäänrakennettujen *testing*-paketin ja *go test* -komennon avulla. Eli jälleen kehittäjän ei tarvitse etsiä standardikirjaston ulkopuolisia kirjastoja testaamiseen, vaan kaikki tarvittava löytyy suoraan Go:sta itsestään. Go:n tapa kääntää ohjelmistot yhteen binääritiedostoon, joka voidaan jakaa mille tahansa alustalle, tekee siitä erittäin hyvän työkalun työskenneltäessä monien eri alustojen kanssa. Go:n muistinsiivous helpottaa kehittäjiä, kun heidän ei tarvitse miettiä muistinhallintaa kehitystyössä, tämä helpottaa varsinkin kokemattomampia kehittäjiä. Go:n muistinsiivoaja on myös äärimmäisen nopea, joten sen aiheuttamat maailmanpysäytykset eivät juurikaan vaikuta ohjelmistojen käyttäjiin.

Heikkouksia Go:sta löytyy myös ja niistäkin osa on ollut aikaisemmin työssä jo esillä. Ensimmäisenä selvänä heikkoutena on Go:n ikä. Vaikka siihen ei luonnollisesti voi vaikuttaa, on selvää, että nuorempana kielenä se on myös kehittymättömämpi, kuin suosituimmat ja suurimmat valtakielet. Kielen ikä vaikuttaa myös suoraan olemassa olevien osaajien löytämiseen. Tämä taas vaikuttaa yritysten haluun kokeilla Go:n käyttämistä omissa ohjelmistoissaan, sillä osaamattomuus ja tuntemattomuus eivät rohkaise kokeilemaan uutta teknologiaa. Go on saanut myös paljon eriäviä mielipiteitä sen tavasta käsitellä virheitä, ja se onkin johtanut siihen, että suurin osa Go:n käyttäjistä on joutunut kirjoittamaan täysin uuden virheidenkäsittelypaketin omiin tarpeisiinsa. Liittyen myös virheiden käsittelyyn Go:ta pidetään turhan monisanaisena, mikä aiheuttaa sen , että Go:lla kirjoitetut ohjelmistot saattavat sisältää turhan paljon uudelleen kirjoitettua koodia. Monisanaisuuteen vaikuttaa suoraan Go:n geneerisyyden puute. Eli se, kuinka paljon Go:n kehittäjä joutuu kirjoittamaan niin sanottua boilerplate-koodia, eli koodia, joka esiintyy monissa osissa ohjelmistoa, ilman suuria muutoksia. Reaaliaikaista käsittelyä vaativat tehtävät on paras jättää kirjoittamatta Go:lla, sillä sen muistinsiivouksen aiheuttamat maailmanpysäytykset voivat vaikuttaa herkkiin reaaliajassa tehtäviin laskuihin, kuten grafiikan renderöintiin tai erilaisten ajurien ja kontrollerien toimintaan. Muistinsiivouksen lisäksi Go:n käyttämiä *gorutiineja* ei ole priorisoitu, joten niiden käyttäminen voi aiheuttaa renderöinnissä kuvanopeuksien (engl. *framerate*) hidastumista.
Vaikka Go:lla on vahva standardikirjasto, on sen ulkopuolisten kirjastojen määrä suhteellisen pieni ja laatu vaihtelevaa. Go:lla joutuu siten lähtökohtaisesti tekemään suurimman osan asioista itse,

toisin kuin monissa muissa kielissä, joissa kirjastoja ja muita työkaluja on kehitetty parhaimmillaan jo yli kolmekymmentä vuotta. Standardikirjastosta ei myöskään löydy mahdollisuutta rakentaa *GUI*:ta, mitä pidetään isona miinuksena varsinkin verkkokehityksen puolella. *GUI*:n rakentamiseen on tehty ulkopuolisia kirjastoja, mutta nähtäväksi jää, lisätäänkö sitä ikinä viralliseksi paketiksi. Vaikka muistinsiivous on äärimmäisen mukava ominaisuus, häiritsee se varsinkin monia kehittäjiä, jotka ovat aikaisemmin käyttäneet esimerkiksi C++:aa, jossa kehittäjä on itse vastuussa muistinkäytöstä. Sen lisäksi, että muistinsiivousta ei saa Go:ssa pois päältä mitenkään, ei siihen ole myöskään kuin yksi vaihtoehto. Esimerkiksi Java tarjoaa neljä eri muistinsiivoajaa, joista käyttäjä voi valita omaan käyttöön sopivimman.

4 Go:n vertailu perinteisempiin ohjelmointikieliin

Tässä luvussa tarkastellaan Go:ta verrattuna kolmeen perinteiseen ohjelmointikieleen C++, Pyhton ja Java, sekä Node.js ympäristöön. Tarkoitus on selvittää lyhyesti, suurimmat erot ja käyttökohteet. Jokaisessa vertailussa on myös tarkasteltu synteettisten testien avulla kielien tehokkuutta. Synteettisiä testejä vertaillessa on hyvä muistaa kuitenkin se, että ne harvoin soveltuvat oikean elämän ongelmiin, siitä huolimatta antavat ne hyvää viitearvoa koodin nopeudesta ja tehokkuudesta.

4.1 Go verrattuna Node.js:ään

Node.js on tällä hetkellä maailman käytetyin *back end*-sovelluskehys. Se on kirjoitettu JavaScriptillä, joka taas on puolestaan maailman käytetyin ohjelmointikieli. Noden laajuudesta on luonnollisesti hyötyä varsinkin teknologiaa tuntemattomalle kehittäjälle, sillä Nodesta löytää helposti tuhansia erilaisia ohjeita ja neuvoja, kun taas Go:n kanssa joutuu tekemään paljon enemmän itsenäistä taustatyötä. Vaikka Go ja Node tarjoavat melko samanlaisia tuloksia oikean maailman ongelmissa, Go:n vahva rinnakkaisuus tekee siitä äärimmäisen hyvin skaalautuvan isoihinkin projekteihin, verrattuna yksisäikeiseen tapahtumakutsuihin perustuvaan Nodeen. Taulukossa 4 näkyvistä testeistä huomataan, että ainakin synteettisissä testeissä Go tuntuu olevan nopeampi ja muistitehokkaampi. [45.]

Go on Nodea jäljessä myös, kun puhutaan kehitystyökaluista. Vaikka Go:lla alkaa olla jo kattava kokoelma kirjastoja ja kehyksiä, ei se vedä vertoja JavaScriptin valtavan yhteisön laatimille työkaluille. Lisäksi virheiden käsittely Go:ssa, kuten aikaisemmin on todettu, on usein turhan monisanaista ja vaatii usein käyttäjän omia implementaatioita. Node taas käyttää jo useista muista kielistä monille kehittäjille tuttua heitä ja koppaa (engl. *throw* and *catch*) menetelmää.

Stackshare.io sivustolla vertaillessa Go:ta ja Nodea, käyttäjien antamat top5 syyt Go:n käyttöön ovat seuraavat: (1) Korkea tehokkuus, (2) yksinkertaisuus ja minimalistisuus, (3) hauska kirjoittaa, (4) rinnakkaisuuden helppous *gorutiininen* avulla ja (5) nopeat käännösajat.

Vastaavasti Noden käyttöön on annettu syiksi; (1) NPM eli *Node Package Manager*, pakettienhallinta työkalu, (2) JavaScript, (3) kirjastotuki, (4) korkea tehokkuus ja (5) avoin lähdekoodi. [46.]

Samalla tavalla käyttäjät ovat antaneet myös huonoja puolia ja Go:n huonoiksi puoliksi on mainittu; (1) Virheiden hallintaan kuluva aika, (2) Turha monisanaisuus, (3) Pakettien ja niiden riippuvuuksien hallinta. Noden huonoiksi puoliksi kerrotaan; (1) Sidottu yhteen CPU:hun, (2) uusien kehyksien julkaisu päivittäin, (3) huonot malliesimerkit internetissä.

Go ja Node.js ovat molemmat siis päteviä vaihtoehtoja *back end*-kehityksessä, Go:lla on paremmat työkalut rinnakkaisuuden ja skaalautuvuuden kanssa työskentelyyn ja Node taas on kattavampi, helpompi opetella ja *fullstack*-kehyksenä *front end* ei ole myöskään ongelma, toisin kuin Go:n kanssa.

Taulukko 4. Go:n ja Node.js vertailua erilaisten koetintestien kautta [45]

fasta				**mandelbrot**		
lähde	aika (s)	muistinkäyttö(tavuina)		lähde	aika (s)	muistinkäyttö(tavuina)
Go	2,08	3 640		**Go**	5,48	30 912
Node.js	9,8	34 804		**Node.js**	17,72	613 608
k-nucleotide				**spectral-norm**		
lähde	aika (s)	muistinkäyttö(tavuina)		lähde	aika (s)	muistinkäyttö(tavuina)
Go	12,77	148 052		**Go**	3,94	2 344
Node.js	61,24	1 739 992		**Node.js**	15,79	31 900
reverse-complement				**n-body**		
lähde	aika (s)	muistinkäyttö(tavuina)		lähde	aika (s)	muistinkäyttö(tavuina)
Go	4	842 356		**Go**	21,37	1 536
Node.js	16,66	706 256		**Node.js**	26,61	33 152
regex-redux				**fannkuch-redux**		
lähde	aika (s)	muistinkäyttö(tavuina)		lähde	aika (s)	muistinkäyttö(tavuina)
Go	28,21	323 820		**Go**	14,72	1 540
Node.js	4,22	596 420		**Node.js**	89,63	30 960

4.2 Go verrattuna Pythoniin

Python on yksi maailman käytetyimmistä ohjelmointikielistä varsinkin data-analytiikan, sekä tekoälyn ja koneoppimisen saralla. Vaikka Go ja Python ovat monella tavalla täysin erilaisia, on niillä

yllättävän paljon yhtäläisyyksiä. Pythonin vahvuudet ovat ennen kaikkea sen valtava kokoelma kirjastoja ja kehyksiä, sekä sen yksinkertaisuus ja helppolukuisuus, mistä päästäänkin Go:n ja Pythonin yhtäläisyyksiin. Molemmat pyrkivät yksinkertaisuuteen ja helppolukuisuuteen, mikä tekee niistä optimaalisia varsinkin uusille ohjelmoijille. Pythonin kirjoittaminen vaatii useasti vähemmän koodia, kuin täsmälleen samaan Go:lla kirjoitettuun ohjelmaan. [47.]

Suurin ero kielten välillä on se, että Python on niin kutsuttu tulkittu kieli ja Go taas niin sanotusti käännetty kieli. Käännetyt kielet ovat aina paljon nopeampia ja taulukosta 5 huomataan, että tehokkuudessa Go jättää Python 3:n kauas taakseen. Se missä Python ei jää jalkoihin, on käyttäjäyhteisö, joka on Pythonin tilanteessa valtava verrattuna Go:hon. Pythonista löytyy työkaluja käytännössä mihin tahansa pelikehityksestä tieteellisiin laskelmiin, mikä tekee siitä äärimmäisen joustavan työkalun. Go:n työkalut eivät ole yhtä kattavat, mutta niillä voidaan tehdä sitä, mihin Go alun perin on kehitetty eli suuren skaalan palvelin- ja verkkokehitykseen.

Pythonilla ja Go:lla on siis paljon yhteisiä käyttömahdollisuuksia, mutta kielen valitseminen on aina riippuvaista tehtävästä projektista. Go soveltuu tehokkuutta ja nopeutta vaativiin sovelluksiin varsinkin verkkoympäristöissä ja Python taas soveltuu paremmin datan käsittelyyn, tekoälyyn ja muuhun yleisempään verkkokehitykseen.

Taulukko 5. Go:n ja Python 3 vertailua erilaisten koetintestien kautta [48]

fasta				**madelbrot**		
lähde	aika (s)	muistinkäyttö(tavuina)		lähde	aika (s)	muistinkäyttö(tavuina)
Go	2,07	3 744		**Go**	5,48	31 196
Python	63,55	844 180		**Python**	259,5	48 192
k-nucleotide				**spectral-norm**		
lähde	aika (s)	muistinkäyttö(tavuina)		lähde	aika (s)	muistinkäyttö(tavuina)
Go	12,58	150 308		**Go**	3,96	2 692
Python	72,24	199 856		**Python**	169,87	49 188
reverse-complement				**n-body**		
lähde	aika (s)	muistinkäyttö(tavuina)		lähde	aika (s)	muistinkäyttö(tavuina)
Go	3,72	826 396		**Go**	21,25	1 884
Python	16,93	1 777 852		**Python**	865,18	8 176
binary-trees				**fannkuch-redux**		
lähde	aika (s)	muistinkäyttö(tavuina)		lähde	aika (s)	muistinkäyttö(tavuina)
Go	25,68	361 532		**Go**	14,75	3 484
Python	80,3	448 004		**Python**	534,4	47 236

4.3 Go verrattuna Javaan

Java on Pythonin tapaan yksi maailman käytetyimpiä kieliä ja sitä pidettiin pitkään parhaana kielenä *back end* -kehitykseen. Molemmat Go ja Java ovat käännettyjä kieliä, mikä näkyy myös taulukon 6 testien suoritusajoista. Java tarvitsee kääntämiseen aina Java virtuaalikoneen (engl. *Java Virtual Machine, JVM*), joka tekee Java koodista tavukoodia ennen, kuin se voidaan kääntää konekielelle. Molemmat Java ja Go kuuluvat myös C-perheeseen, joten niiden syntaksit ovat samankaltaisia, mutta Go yksinkertaisuutensa kautta tekee siitä helpommin luettavaa. Kummallakin, Javalla ja Go:lla on mahdollista kehittää monelle alustalle. Javan JVM keskustellessa laitteiston kanssa kasaa Go ohjelmat suoraan binääritiedostoon. Tämän takia Go on yleisesti hankalampi toimittaa monille alustoille ilman erilaisia konfiguraatioita. Go:n suurin etu Javaan verrattuna on jälleen kerran sen sisäänrakennettu mahdollisuus rinnakkaisuuteen. Vaikka Javalla voidaan myös rakentaa rinnakkaisia ohjelmia, on se paljon resursseja kuluttavampaa, kun Go:n erittäin kevyet *gorutiinit*. Molemmat kielet ovat muistisiivottuja, mutta Javalla on neljä eri tyyppiä [49]; *Serial, Parallel, CMS ja G1*, kun Go:lla taas vain yksi huonosti muokattavissa oleva vaihtoehto.

Java loistaa monimutkaisuutta vaativissa suurissa projekteissa kehittyneempien kirjastojensa ja suuremman yhteisönsä kautta, mutta Go on erittäin hyvä työkalu pienempiin rinnakkaisuutta vaativiin projekteihin. Lisäksi Go:n helppolukuisuus tekee siitä paremman vaihtoehdon tiimityöskentelyyn.

Taulukko 6. Go:n ja Javan vertailua erilaisten koetintestien kautta [50]

fasta			**mandelbrot**		
lähde	aika (s)	muistinkäyttö(tavuina)	lähde	aika (s)	muistinkäyttö(tavuina)
Go	2,07	3 744	**Go**	5,48	31 196
Java	2,22	45 172	**Java**	6,83	79 108
k-nucleotide			**spectral-norm**		
lähde	aika (s)	muistinkäyttö(tavuina)	lähde	aika (s)	muistinkäyttö(tavuina)
Go	12,58	150 308	**Go**	3,96	2 692
Java	9,33	447 976	**Java**	4,22	36 948
reverse-complement			**n-body**		
lähde	aika (s)	muistinkäyttö(tavuina)	lähde	aika (s)	muistinkäyttö(tavuina)
Go	3,72	826 396	**Go**	21,25	1 884
Java	3,16	712 368	**Java**	21,93	35 408
pidigits			**fannkuch-redux**		
lähde	aika (s)	muistinkäyttö(tavuina)	lähde	aika (s)	muistinkäyttö(tavuina)
Go	2,04	8 732	**Go**	14,75	3 484
Java	3,07	39 320	**Java**	14,33	34 888

Vuonna 2011 Googlen työntekijä Robert Hundt teki vertailua Go:n, Javan, C++:an ja Scalan välillä toistorakenteiden (engl. *loop*) tunnistuksessa. Tutkimuksessa implementoitiin testiä varten rakennettu algoritmi tutkituilla kielillä ja tuloksia arvioitiin koodin koon, käännösaikojen, binäärikokojen, muistinkäytön ja muiden ajonaikaisten mittausten perusteella. Tutkimus osoittautui Go:n kannalta huonoksi, sillä sen tulokset kaikkien muiden kielten taakse. Tutkimuksen julkaisun jälkeen yksi Go:n kehittäjistä oli kuitenkin kertonut, että tutkimuksessa käytetty Go:n koodi ei ollut idiomaattista Go:ta ja testikoodia muokattiin kehittäjän ohjeiden mukaan. Myös muiden kielin koodeja muokattiin, ja ne ajettiin uudestaan, tällä kertaa täysin erilaisin tuloksin. Alkuperäisessä versiossa Go:n suoritusvaihe kesti 56,92s ja käytti 1604MB muistia, kun taas uudessa optimoidussa versiossa Go jätti jopa C++:an taakseen suoritusvaiheen kestäessä vain 3.84s ja käyttäen vain 257MB muistia, eli suoritusvaiheen kesto putosi yli 50s ja muistinkäyttö väheni yli 1300MB. [51.][52.]

4.4 Go verrattuna C++:aan

C++ on Go:n kanssa samaan C-kielten perheeseen kuuluva järjestelmäkieli, joka on ollut yleisesti käytössä jo yli kolmekymmentä vuotta. C++ on äärimmäisen tehokas ja nopea kieli ja sitä käytetäänkin tehokkuutta ja suorituskykyä vaativissa tehtävissä, kuten pelimoottoreissa, hakukoneissa ja selaimissa, pankkijärjestelmissä ja käyttöjärjestelmissä [53]. C++ on niin sanottu keskitason kieli, eli se keskustelee lähempänä laitteistoa, kun korkeamman tason kielet, kuten Go tai Java. C++ on valtava kieli verrattuna Go:hon, mikä vaikuttaa varsinkin siihen, kuinka nopeasti koodia voidaan kirjoittaa. Siinä, missä Go keskittyy yksinkertaisuuteen ja tiiviyteen, C++ keskittyy tehokkuuteen. Suurimmat erot Go:n ja C++:an välillä ovat muistinkäsittely ja kielen taso, jotka molemmat omilta osin johtavat kielten erilaisiin käyttötarkoituksiin. C++ ei ole muistisiivottu kieli, kuten Go vaan ohjelmoijan täytyy itse pitää huolta muistinkäytöstä. Tämä laajentaa ohjelmoijan vapautta, mutta samalla lisää ohjelmoijan taakkaa, varsinkin kokemattomampien käyttäjien kannalta, sillä muistivuodot ovat vakava ongelma ohjelmistokehityksessä. [54.]

Go:ta mainostetaan nopeana kielenä, ja sitä se onkin, kun verrataan dynaamisiin ja tulkittuihin kieliin, kuten aikaisemmin nähtiin esimerkiksi Pythonin kanssa. Go ei ole tästä huolimatta lähelläkään C++:an tarjoamaa tehokkuutta, kuten taulukosta 7 huomataan. C++ jättää Go:n kauas taakseen melkein kaikissa testeissä [55]. Nopeuksien erot eivät kuitenkaan ole yhtä suuria, kuin esimerkiksi Go:n ja Pythonin välillä ja Go sinnitteleekin varsin hyvin C++ perässä ottaen huomioon, että Go:ta hidastaa automaattinen muistinsiivous. C++ on siitä kiitetty kieli, että sitä voidaan käyttää yleisyytensä takia lähes missä tahansa projekteissa ja yli 30 vuotta vanhana kielenä, siltä löytyy Pythonin tavoin laaja valikoima valmiita kirjastoja ja kehyksiä. Ja koska C++ on niin ylivoimainen tehokkuudessaan, ei Go tule ainakaan lähiaikoina astumaan sen reviirille nopeutta vaativissa ohjelmistoissa, mutta verkko- ja palvelinkehityksessä Go tuntuu olevan suositumpi yksinkertaisuutensa ja helpomman rinnakkaisuutensa ansiosta.

Taulukko 7. Go:n ja C++ vertailua erilaisten koetintestien kautta [56]

fasta				mandelbrot		
lähde	aika (s)	muistinkäyttö(tavuina)		lähde	aika (s)	muistinkäyttö(tavuina)
Go	2,07	3 744		**Go**	5,48	31 196
C++	1,46	2 180		**C++**	1,51	25 708
k-nucleotide				**spectral-norm**		
lähde	aika (s)	muistinkäyttö(tavuina)		lähde	aika (s)	muistinkäyttö(tavuina)
Go	12,58	150 308		**Go**	3,96	2 692
C++	3,89	156 148		**C++**	1,98	2 320
reverse-complement				**n-body**		
lähde	aika (s)	muistinkäyttö(tavuina)		lähde	aika (s)	muistinkäyttö(tavuina)
Go	3,72	826 396		**Go**	21,25	1 884
C++	1,88	499 844		**C++**	7,7	1 772
binary-trees				**fannkuch-redux**		
lähde	aika (s)	muistinkäyttö(tavuina)		lähde	aika (s)	muistinkäyttö(tavuina)
Go	25,68	361 532		**Go**	14,75	3 484
C++	3,93	113 768		**C++**	10,7	1 864

4.5 Kielten vertailun yhteenveto

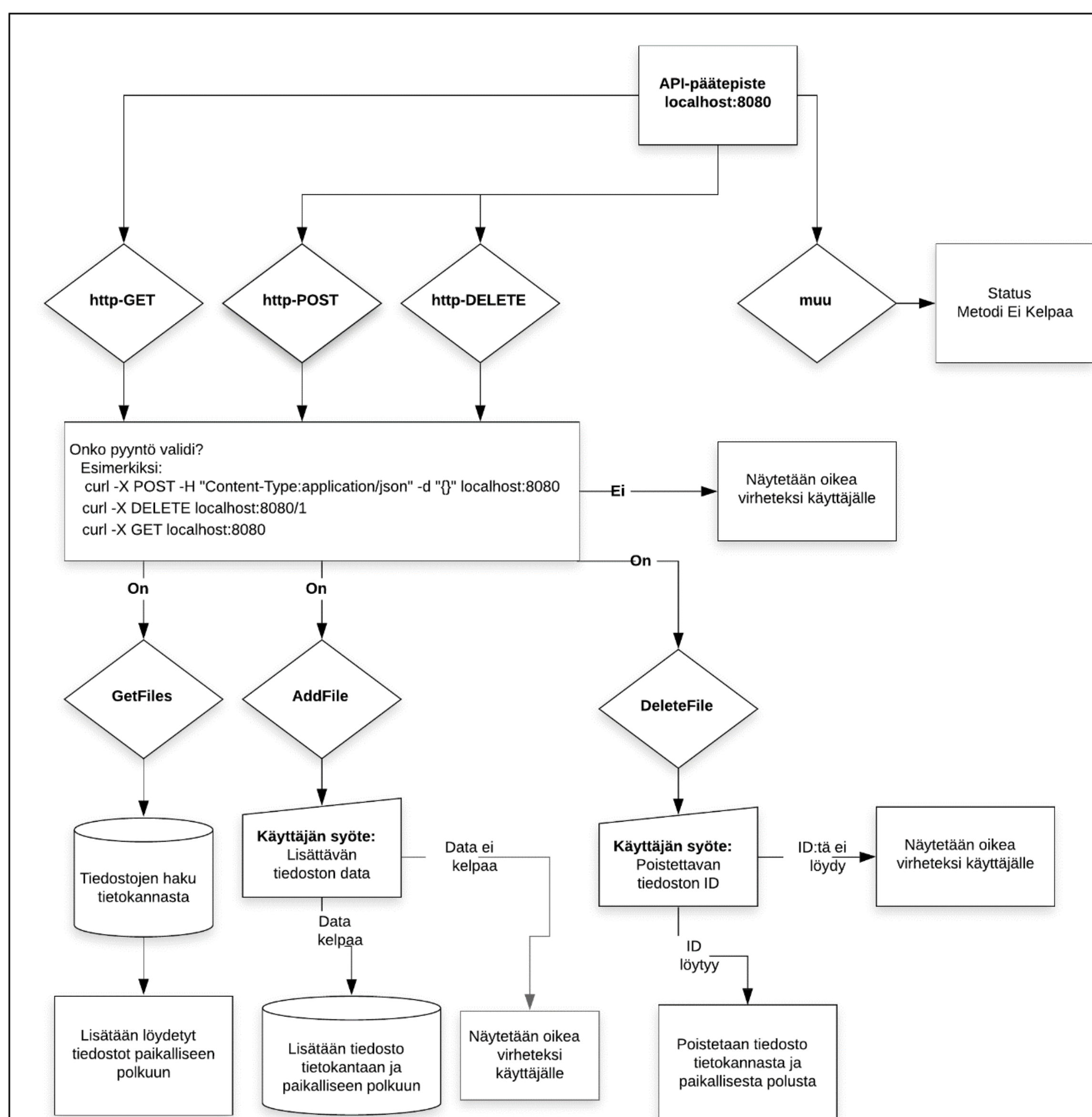
Ohjelmistokehityksessä on äärimmäisen tärkeää valita työhön sopiva työkalu. Go soveltuu parhaiten rinnakkaisuutta vaativiin töihin. Nykyajan prosessorit ovat moniytimisiä, joten niistä saadaan enemmän irti kirjoittamalla ohjelmistot rinnakkaisuutta käyttäen. Go:ssa rinnakkaisuus on

sisäänrakennettuna kieleen toisin, kuin missään muussa vertailussa olevista kielissä. Tämä antaa sille modernissa moniydinprosessorien maailmassa etua, kun ohjelmoijan ei tarvitse taistella itse kieltä vastaan.

Go on kaikkia vertailtuja kieliä ja kehyksiä jäljessä luonnollista kasvua ja kehitystä vaativilla alueilla, kuten kolmansien osapuolien kehitystyökalujen määrässä ja laadussa. Vaikka Go:lla on jo nyt suhteutettuna ikäänsä laaja kirjasto työkaluja, ei se vedä vertoja useita vuosikymmeniä edellä oleville valtakielille.

5 Teknisen työn toteutus

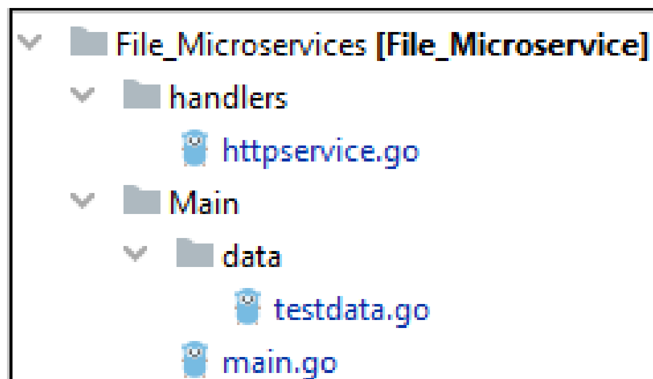
Tämän insinöörityön teknisen osan on tarkoitus toimia toimeksiantajalle esimerkkinä yksinkertaisen mikropalvelun rakentamisesta Go:lla. Mikropalveluna toimii yksinkertainen *POST*, *GET* ja *DELETE* http-metodeja käyttävä palvelu, jolla voidaan lisätä, hakea ja poistaa tiedostoja paikalliseen polkuun tietyn ID:n avulla. Ohjelman kulkua havainnollistetaan kuvan 7 avulla. Työn toteutukseen käytetään JetBrainsin GoLand IDE:tä. Versionhallintaan käytettiin GitHubia ja sen työpöytäsovellusta. Työssä ei käytetty mitään kolmannen osapuolen kirjastoa, kehystä tai työkalua.



Kuva 7. Vuokaavio ohjelman toiminnasta

Kuten kaaviosta nähdään, on ohjelman toiminta yksinkertainen. Palvelin kuuntelee porttia 8080 ja odottaa käyttäjän lähettämiä http-pyyntöjä. Pyyntöjen saapuessa palvelin käsittelee pyynnöt määrätyllä tavalla. *GET*-pyyntö hakee tietokannasta kaikki tiedostot ja lisää ne paikalliseen polkuun, jos ne puuttuvat. *POST*-pyyntö hakee tietokannasta viimeisen tiedoston ID:n ja antaa luotavalle tiedostolle automaattisesti seuraavan vapaan ID:n ja tallentaa tiedoston tietokantaan ja paikalliseen polkuun. *DELETE*-pyyntö ottaa parametrina poistettavan tiedoston ID:n ja tarkastaa löytyykö ID:tä vastaava tiedosto. Jos tiedosto löytyy, poistetaan se tietokannasta ja paikallisesta polusta.

Projektin on tarkoitus toimia hyvin kommentoituna esimerkkinä Go-ohjelman rakentamisesta, joten yksinkertaisuus ja helppolukuisuus ovat avain asemassa työn toteutuksessa. Projekti koostuu kolmesta .go tiedostosta: *httpservice.go*, joka vastaa http-kutsujen käsittelystä, *testdata.go*, joka sisältää funktiot *JSON*:in käsittelyyn, sekä projektin testitietokantana toimivan listan tiedostoja, sekä *main.go*, jossa käsitellään käytettävää palvelinta. Kansiorakenne nähdään kuvasta 8.



Kuva 8. Projektin kansiorakenne

Seuraavaksi käydään läpi ohjelman koodirakenne pääpiirteittäin viitaten aiemmin mainituista tiedostoista otettuihin kuvankaappauksiin. Tarkoituksena on antaa lyhyt ja ytimekäs selostus siitä, mitä ohjelman sisällä tapahtuu.

Esimerkkikoodissa 10 näkyy *main.go* tiedoston sisältö tiivistettynä. Kaikki projektin tiedostot on formatoitu käyttäen *go fmt*-työkalua, jotta formaatti on yhtenäinen läpi projektin. *Main*-tiedostossa luodaan ja käynnistetään uusi palvelin halutuilla asetuksilla, sekä toiminallisuus palvelimen

sulavaan (engl. *graceful*) sammuttamiseen, jotta taustalla pyörivät operaatiot saadaan suoritet-tua ennen palvelimen sammumista.

```
// main-funktio on palvelun aloituspiste
func main() {
    // Oma loggeri, joka käyttää standardiflagejä
    l := log.New(os.Stdout,  prefix: "File_Microservice", log.LstdFlags)

    fh := handlers.NewFiles(l) // Luodaan uusi FileHandler fh
    sm := http.NewServeMux()   // Luodaan uusi ServeMux sm
    sm.Handle( pattern: "/", fh)        // Määritetään handlerin patterni "/", esim. pyyntö localhost:8080/X

    s := http.Server{ // Luodaan uusi palvelin
        Addr:              ":8080",            // osoitteena toimii kaikki sisäverkon osoitteen porttina 8080
        ErrorLog:          l,                  // Uusi loggeri
        Handler:           sm,                 // Oma Mux
        ReadTimeout:       5 * time.Second,    // Kuinka kauan lukeminen saa kestää ennen timeouttia
        ReadHeaderTimeout: 0,                  // Kuinka kauan pyyntöjen headereitten lukeminen saa kestää ennen timeouttia
        WriteTimeout:      5 * time.Second,    // Kuinka kauan kirjoittaminen saa kestää ennen timeouttia
        IdleTimeout:       60 * time.Second,   // Kuinka kauan palvelin saa olla idlenä ennen timeouttia
    }

    // Anonyymi funktio, jolla käynnistetään palvelin
    go func() {
        l.Println( v...: "Käynnistetään palvelin porttiin :8080")
        err := http.ListenAndServe( addr: ":8080", sm) // http.ListenAndServe käynnistää palvelimen annettuun osoitteeseen
        if err != nil {                          // Tarkastetaan palvelimen käynnistys virheen varalta
            l.Printf("Virhe käynnistäessä palvelinta:  #{err.Error()}") // Jos virhe löytyy, ilmoitetaan käyttäjälle ja
            os.Exit( code: 1)                                          // Sammutetaan sovellus
        }
    }()

    // Pienimuotoinen esimerkki kanavien käytöstä palvelimien yhteydessä
    c := make(chan os.Signal, 1)   // Luodaan puskuroitu kanava (jotta ei riskeerata signaalien menetystä)
    signal.Notify(c, os.Interrupt) // Notify välittää Interrupt signaalin c:lle = käyttäjä painaa Ctrl + C

    // Blokataan, kunnes saadaan signaali
    sig := <-c
    log.Println( v...: "Saatu signaali:", sig)

    // Sammutetaan palvelin sulavasti 30sec jälkeen, jotta päällä olevat operaatiot saadaan suoritettua
    ctx, _ := context.WithTimeout(context.Background(), 30*time.Second)
    s.Shutdown(ctx)
}
```

Esimerkkikoodi 10. *File_Microservicen main.go*-tiedoston sisältö

Kuten koodista huomataan Go:n avulla palvelimien käynnistäminen ja hallitseminen voidaan tehdä vain muutamalla rivillä koodia. Palvelin käynnistyy, kun koodi käännetään ja sammuu, kun ohjelma sammuu tai käyttäjä painaa *ctrl + c*-yhdistelmää, joka lähettää *interrupt* signaalin kana-valle, jonka kautta palvelin sammutetaan.

Seuraavaksi siirrytään *httpservice.go*-tiedostoon, jossa käsitellään palvelimelle saapuvat http-pyynnöt. Tärkein funktio on *ServeHTTP*, joka tarkastaa mikä pyyntö palvelimelle saapuu ja toimii sen mukaan. *ServeHTTP* sisältää if-ehtolauseen, joka tarkastaa minkä metodin palvelimelle saa-puva pyyntö sisältää ja sen mukaan joko hakee tiedostot, lisää uuden tiedoston, poistaa olemassa olevan tiedoston tai lähettää virheilmoituksen. Sallitut metodit ovat *POST*, *GET* ja *DELETE*.

```
// ServeHTTP täyttää http.Handler interfacen ehdot ja toimii aloituspisteenä handlerille
// ServeHTTP käsittelee saapuvat HTTP kutsut (POST,DELETE,GET)
// f on ServeHTTP vastaanottaja (receiver), eli ServeHTTP:n sisällä päästään käsiksi f:ään.
// Argumenttina ottaa http.ResponseWriterin, joka rakentaa HTTP vastauksen (response)
// ja http.Request pointerin, joka on serverin vastaanottama pyyntö.
// ServeHTTP ei palauta mitään
func (f *Files) ServeHTTP(rw http.ResponseWriter, r *http.Request) {
    if r.Method == http.MethodGet { // Tarkastetaan onko saapuva pyyntö GET
        f.getFiles(rw, r) // Jos pyyntö on GET, kutsutaan getFiles funktiota
        return            // ServeHTTP ei palauta mitään
    }
    if r.Method == http.MethodPost { // Tarkastetaan onko saapuva pyyntö POST
        f.addFile(rw, r) // Jos pyyntö on POST, kutsutaan addFile funktiota
        return           // ServeHTTP ei palauta mitään
    }
    if r.Method == http.MethodDelete { // Tarkastetaan onko saapuva pyyntö DELETE
        f.lgr.Println( v...: "DELETE", r.URL.Path)
        reg := regexp.MustCompile( str: `/([0-9]+)`)          // Luodaan Regular expression
        g := reg.FindAllStringSubmatch(r.URL.Path,  n: -1) // Etsitään URLista regexin avulla ID

        if len(g) != 1 { // Jos löytyy useampi, kuin yksi ID, kirjataan virhe
            f.lgr.Println( v...: "URI ei kelpaa, liian monta ID:tä")      // Tulostetaan käyttäjälle virheteksti
            http.Error(rw,  error: "URI ei kelpaa", http.StatusBadRequest) // Lähetetään BadRequest (status 400) virhe
            return                                                          // ServeHTTP ei palauta mitään
        }
        if len(g[0]) != 2 { // Tarkastetaan, ettei regexin capture grouppeja ole liikaa
            f.lgr.Println( v...: "URI ei kelpaa, liian monta capture grouppia") // Tulostetaan käyttäjälle virheteksti
            http.Error(rw,  error: "URI ei kelpaa", http.StatusBadRequest)      // Lähetetään BadRequest (status 400) virhe
            return                                                               // ServeHTTP ei palauta mitään
        }
        idString := g[0][1]               // Luodaan IDstä string muuttuja
        id, err := strconv.Atoi(idString) // Muutetaan string intiksi
        if err != nil {                   // Jos muutos ei onnistu
            f.lgr.Println( v...: "URI ei kelpaa, ei voitu muuttaa numeroksi", idString) // Näytetään virheteksti käyttäjälle
            http.Error(rw,  error: "URI ei kelpaa", http.StatusBadRequest)              // Lähetetään BadRequest (status 400) virhe
            return                                                                       // ServeHTTP ei palauta mitään
        }
        f.deleteFile(id, rw, r) // Jos ID löytyy oikein, kutsutaan deleteFile funktiota.
        return                  // ServeHTTP ei palauta mitään
    }
    // Jos metodi on muu kuin POST, GET, DELETE kirjoitetaan virhe
    rw.WriteHeader(http.StatusMethodNotAllowed)
}
```

Esimerkkikoodi 11. *HTTPServe*-metodi *httpservice.go*-tiedostossa

GET- ja *POST*-metodien tarkastaminen on helppoa, mutta *DELETE*-metodin käyttöä varten joudutaan tekemään säännöllinen lauseke, joka tarkastaa käyttäjän lähettämän pyynnön URL:n virheiden varalta. *HTTPServen* lisäksi tiedostosta löytyy funktiot *getFiles*, *addFiles* ja *deleteFiles*, joita kutsutaan *HTTPServen* sisältä, nimiensä mukaisesti ne hakevat, lisäävät ja poistavat tiedostoja.

Viimeisenä käydään läpi *testdata.go*-tiedosto. Se sisältää niin itse testitietokannan, kuin myös funktiot tietokantaan tiedostojen lisäämiseen, poistamiseen ja hakemiseen. Kaikki tietokannassa olevat tiedostot ovat esimerkkikoodissa 12 esiteltävää *File*-tietotyyppiä, jolla on ID, Nimi, Tyyppi ja Sisältö, koodissa näkyy myös käytettävä tietokanta *fileList*.

```
type File struct {
    ID      int    `json:"id"`      // Luodun tiedoston ID
    Name    string `json:"name"`    // Luodun tiedoston nimi
    Type    string `json:"type"`    // Luodun tiedoston tyyppi
    Content string `json:"content"` // Luodun tiedoston sisältö
}
// Testidatana toimii lista kovakoodatuista Fileistä.
var fileList = []*File{
    &File{
        ID:      1,
        Name:    "testitiedosto",
        Type:    ".txt",
        Content: "Lorem ipsum dolor sit amet.",
    },

    &File{
        ID:      2,
        Name:    "testitiedosto2",
        Type:    ".txt",
        Content: "Lorem ipsum dolor sit amet.",
    },

    &File{
        ID:      3,
        Name:    "jsontiedosto1",
        Type:    ".json",
        Content: "Lorem ipsum dolor sit amet.",
    },
}
```

Esimerkkikoodi 12. *File*-tietotyyppi ja *fileList*-tietokanta

Testdata.go sisältää funktiot *JSON*-tyyppisen datan käsittelyyn molempiin suuntiin, eli *JSON*:sta haluttuun tiedostoon ja halutusta tiedostosta *JSON*:iin. Nämä funktiot on esitetty esimerkkikoodissa 13.

```
// Funktio, joka decodaa saadun Json datan.
// f on fromJSONin vastaanottaja (receiver), eli FromJSONin sisällä päästään käsiksi f:ään.
// Argumenttina annetaan io.Reader.
// io.Reader on interface, joka wrappaa Read metodin
// Käytetään httpserice.go:n puolella, kun lisätään uusia tiedostoja
func (f *File) FromJSON(r io.Reader) error {
    fileDecoder := json.NewDecoder(r) // fileDecoder on decoder, joka lukee Json datan Readeriltä r
    return fileDecoder.Decode(f)      // Funktio palauttaa decoodatun tiedoston f sisällön
}

// Files on slice muodossa oleva kokoelma luotuja File-tyypisiä 'objekteja'
type Files []*File

// ToJson funktio sarjastaa (serialize) Files kokoelman Json muotoon.
// f on toJSONin vastaanottaja (receiver), eli toJSONin sisällä päästään käsiksi f:ään.
// Argumenttina annetaan io.Writer.
// io. Writer on interface, joka wrappaa Write metodin
// Käytetään httpserice.go:n puolella, kun haetaan olemassa olevia tiedostoja
func (f *Files) ToJSON(w io.Writer) error {
    fileEncoder := json.NewEncoder(w) // fileEncoder on encoder, joka kirjoittaa encodaatun Jsonin Writteriin w
    return fileEncoder.Encode(f)      // Funktio palauttaa encoodatun datan
}
```

Esimerkkikoodi 13. *ToJSON*- ja *FromJSON*-funktiot

FromJson-funktio lukee *JSON*-dataa ja palauttaa dekoodatun tiedoston sisällön. *ToJSON* on sen käänteinen funktio, eli se lukee tiedoston ja kirjoittaa sieltä löydetyn datan *JSON*:ksi ja palauttaa kooditetut datan.

Seuraavaksi käydään läpi kaikki kolme tiedostojen käsittelyyn käytettävää funktiota *AddFile*, *DeleteFile* ja *GetFiles*. Esimerkkikoodista 14 nähdään *AddFile*-funktion toiminta, esimerkkikoodista 15 *DeleteFile*- ja *RemoveIndex*-funktioiden toiminta ja esimerkkikodista 16 *GetFiles*-funktion toiminta.

```
// AddFile luo uuden tiedoston annetulla datalla
// Argumenttina on luotava tiedosto f
func AddFile(f *File) {
    f.ID = getNextID()   // Haetaan seuraava ID, filesListin mukaan
    if fileList != nil { // Jos fileList ei ole tyhjä, lisätään f appendilla listaan
        fileList = append(fileList, f)
    } else { // Jos fileList on tyhjä (null), luodaan tyhjä lista ja lisätään f listaan appendilla
        println(fileList)
        fileList = []*File{}
        fileList = append(fileList, f)
    }
    contentString := "ID: " + strconv.Itoa(f.ID) + "\n" + "name: " + f.Name + // Luodaan string muuttuja
        "\n" + "type: " + f.Type + "\n" + "content: " + f.Content
    println(f.Name, f.Type, f.ID)
    content := []byte(contentString)                      // Muokataan luotu string muuttuja byteiksi
    testFile, err := os.Create(filePath + f.Name + f.Type) // Luodaan os moduulin avulla uusi tiedosto
    defer testFile.Close()                                 // Defer avainsanalla varmistetaan, että luotu tiedosto suljetaan
    if err != nil {
        println(err.Error())
    }
    _, _ = testFile.Write(content) // Kirjoitetaan data luotuun tiedostoon
}
```

Esimerkkikoodi 14. AddFile-funktion toiminta

AddFilen ainoa parametri on luotava tiedosto f. Funktio hakee ensin *getNextID*-funktion avulla seuraavan vapaan ID:n tietokannasta. Jos tietokanta on tyhjä eli *nil*, luodaan uusi lista, johon uusi tiedosto *append*-funktiolla pusketaan, jos tietokana ei ole tyhjä ei ole tarvetta luoda uutta ja tiedosto voidaan puskea suoraan olemassa olevaan kantaan. Uuden tiedoston data muokataan merkkijono-tyypistä tavuiksi, jotta se voidaan kirjoittaa luotuun tiedostoon *Write*-metodilla, kun se on luotu *os.Create*-metodilla.

DeleteFile-funktiolla on kaksi parametria, poistettavan tiedoston ID, sekä pointteri tiedostoon. Ensin *findFile*-funktiolla etsitään annetulla ID:llä tietokannasta tiedostoa, jos ei löydy palautetaan tyhjä ja kirjataan virhe. Jos tiedosto löytyy, muodostetaan poistettavan tiedoston paikallinen polku ja *os.Remove*-metodilla poistetaan tiedosto. Tietokannasta tiedosto poistetaan *RemoveIndex*-funktiolla, joka kopioi vanhan listan päälle uuden listan ilman poistettua tiedostoa.

```
// DeleteFile poistaa tiedoston paikallisista tiedostoista ja fileLististä
// Käytetään httpservice.go:ssa
// Argumenttina annetaan poistettavan tiedoston ID ja tiedosto f
func DeleteFile(id int, f *File) error {
    ff, pos, err := findFile(id) // Etsitään findFile funktiolla haluttu tiedosto ID:n avulla   ff = löydetty tiedosto, pos = positio fileListissä
    if err != nil {              // Tarkistetaan etsintä virheiden varalta
        println(err.Error())
        return err
    }
    var pathToDelete = filePath + ff.Name + ff.Type // Poistettavan tiedoston polku

    if _, err := os.Stat(pathToDelete); err == nil { // os.Stat palauttaa FileInfo kuvauksen tiedostosta
        println( args...: "Poistettu: ", pathToDelete)
        var er = os.Remove(pathToDelete) // Poistetaan paikallinen tiedosto os.Remove funktiolla
        if er != nil {                   // Tarkastetaan poistaminen virheiden varalta
            println(er.Error())
            return er
        }
    } else {
        println( args...: "Tiedostoa ei löytynyt polusta: ", pathToDelete) // Jos tiedostoa ei löydy, tulostetaan virhe
    }

    fileList[pos] = f                    // Määritetään poistettavan tiedoston paikka listassa
    fileList = RemoveIndex(fileList, id) // Poistetaan elementti listasta

    return nil
}
```

Esimerkkikoodi 15. *DeleteFile* ja *RemoveIndex*-funktioiden toiminta

GetFiles-funktio yksinkertaisesti käy läpi tietokannan sisällön, tulostaa sen konsoliin, sekä lisää kaikki löydetyt tiedostot paikalliseen polkuun, jos ne eivät siellä jo ole. Virhetilanteessa virhe tulostetaan konsoliin.

```
// GetFiles hakee kaikki tiedostot 'filesList' listasta ja luo ne paikallisesti 'data' kansioon
// Käytetään httpservice.go:ssa
func GetFiles() Files {
    if len(fileList) != 0 { // Tarkastetaan ettei lista ole tyhjä
        for _, f := range fileList { // Loopataan fileListin sisällön läpi
            contentString := "ID: " + strconv.Itoa(f.ID) + "\n" + "name: " + f.Name + "\n" +
                "type: " + f.Type + "\n" + "content: " + f.Content // Luodaan string muuttuja
            content := []byte(contentString)                      // Muokataan luotu string muuttuja byteiksi
            testFile, cErr := os.Create(filePath + f.Name + f.Type) // Luodaan os moduulin avulla uusi tiedosto, haluttuun polkuun
            defer testFile.Close()                                // Defer avainsanalla varmistetaan, että luotu tiedosto suljetaan
            if cErr != nil {
                println(cErr.Error())
            }
            _, _ = testFile.Write(content) // Kirjoitetaan data luotuun tiedostoon
        }
    } else {
        fileList = nil // Jos lista on tyhjä asetetaan se nulliksi virhetilojen välttämiseksi
    }

    println( args...: "Amount of files: ", len(fileList))
    return fileList // Palautetaan lista löydetyistä tiedostoista
}
```

Esimerkkikoodi 16. *GetFiles*-funktion toiminta

6 Teknisen työn pohdinta

Teknisestä osiosta tuli varsin onnistunut esimerkki yksinkertaisuudestaan huolimatta. Go osoittautui taustatutkimusta mukaillen helposti opittavaksi, sekä yksinkertaiseksi kirjoittaa. Palvelimen ja asiakaskoneen välinen keskustelu saatiin toimimaan vain muutamalla rivillä koodia ja ilman mitään kolmannen osapuolen työkaluja. Koodista tuli myös hyvin yhteneväistä, kun käytettiin *go fmt*-toimintoa, joka muokkaa kaikki projektin tiedostot noudattamaan samaa formaattia.

Projektin suunnitteluun käytettiin aikaa pari päivää, jonka jälkeen kehitystyö sujuikin varsin kivuttomasti alusta loppuun, eikä suuria ongelmia tullut lainkaan vastaan. Suurimman päänvaivan tuotti yhden tietyn elementin poistaminen *slicestä*, koska sille ei Go:sta löydy suoraan metodia. Tämäkin oli kuitenkin varsin pieni ongelma, joka selvisi muutaman tunnin tutustumisella Go:n dokumentaatioon.

Esimerkkikoodin laajalla kommentoinnilla pyrittiin minimoimaan mahdollisia väärinkäsityksiä ja turhaa ajan hukkaa kehittäjälle, joka sitä lukee. Tekninen työ hyväksytettiin toimeksiantajalla, jotta varmistettiin sen soveltuvuus mahdolliseen jatkokäyttöön.

7 Yhteenveto

Työn tarkoituksena oli saada kattava käsitys siitä, mitä Go-ohjelmointikielellä kannattaa tehdä, mitä sillä on jo tehty ja miksi käyttää Go:ta muiden ohjelmointikielien sijaan. Koska kyseessä oli teoriapainotteinen tutkimustyö, oli selvää, että työn suurin haaste olisi löytää tarvittava määrä lähdemateriaalia, koska kyseessä oli verrattain nuori ohjelmointikieli. Työn ensimmäiset viikot kuluivat suunnitellessa sisällön rakennetta ja etsiessä lähdemateriaalia. Kun sisällön rakenteesta oli saatu ensimmäinen selvä versio, lähdettiin haravoimaan löydettyä lähdemateriaalia. Lähdemateriaalia löytyikin yllättävän paljon ja sitä jouduttiin jopa karsimaan matkan varrella.

Työn ensimmäisen osion oli tarkoitus kertoa taustaa Go:n kehityskaaresta ja avata kehittäjien tekemiä ratkaisuja Go:n sisällöstä. Go:n historia onkin lyhykäisyydestään huolimatta varsin nousujohteinen ja Go:ta tullaan varmasti näkemään enemmän ja enemmän vuosien saatossa. Go:n kehittäjät ovat pitäneet tarkasti kiinni asettamistaan rajoitteista ja tavoitteista Go:n suhteen, eikä Go:hon lisätä ominaisuuksia ilman tarkkaa harkintaa. Go:n ominaisuuksiin syventyminen oli varsin mielenkiintoista, koska oma ohjelmointikokemukseni nojautui lähes täysin C++-ohjelmointiin. Go:n tapa tehdä asioita oli silmiä avaavaa ja teknistä työtä tehdessäni sain konkretiaa tukemaan Go:n ajatusmaailmaa.

Toimeksiantajalle oli tärkeää saada tietoa siitä, mitä Go:n avulla on tähän asti tehty ja syventyä syihin miksi yritykset ovat valinneet juuri Go:n työkalukseen. Tutkiessa yritysten kehittämiä ohjelmistoja oli selvää, että Go:n käyttäminen painottui vahvasti palvelin- ja verkkokehitykseen. Go:n rinnakkaisuus tuo eniten hyötyä varsinkin moniytimisten prosessorien ja valtavien prosessiklustereiden käytössä. Go:n käyttämät kevyet säikeet *gorutiinit* ovat tehokas tapa luoda rinnakkain ajettavaa koodia. Lisäksi suuremmat kehitystiimit hyötyvät Go:n tiukasta formaatista, jonka vuoksi koodin luettavuus kehittäjältä kehittäjälle paranee merkittävästi.

Suurin osa yrityksistä ovat vaihtaneet Go:hon jostakin muusta kielestä. Tämä oli tärkeä huomio ja eri ohjelmointikieliä vertailemalla saatiin arvokasta tietoa siitä, mihin Go soveltuu parhaiten. Go:ta vertailtiin Javaan, C++:aan, Pythoniin ja Node.js kehykseen. Vertailuissa todettiin, että käännettynä kielenä Go on monin verroin nopeampi kuin dynaamiset kielet Java ja Python. Kuitenkaan Go ei ollut koetintesteissä lähelläkään C++:an tehokkuutta. Node.js ja Go olivat erittäin tasaver-

taisia koetintestien perusteella, mutta kuitenkin ne soveltuvat parhaiten erityyppisiin projekteihin. Siinä missä Go loistaa palvelinpuolella Node.js on kokonaisvaltainen *fullstack*-kehys, jolla onnistuu yleisempi verkkokehitys. Node.js sisältää myös työkalut *front endin* kehitykseen toisin kuin Go.

Työn tekeminen oli palkitsevaa ja sainkin laajennettua omaa tietämystäni ohjelmointikielistä niin yleisellä, kuin syvemmälläkin tasolla. Vaikka keskityinkin nimenomaan ohjelmointikieliin, pääsin samalla tutustumaan pilvi- ja mikropalveluiden rakentamiseen, muistinsiivoukseen ja muihin spesifeihin osa-alueisiin. Haastetta työhön toi sen sisällön pitäminen mahdollisimman tiiviinä ja käytännöllisenä. Työhön pyrittiin sisältämään vain toimeksiantajalle lisäarvoa tuovat oleelliset ominaisuudet, mutta tutkimustyön luonteen vuoksi sisältöä tuli aiottua enemmän.

Lähteet

(1) Pike R (24.4.2014). Hello, Gophers! [PowerPoint-esitys]. Saatavilla osoitteesta: https://talks.golang.org/2014/hellogophers.slide#1. Haettu 4.2.2020.

(2) GoogleTechTalks (10.11.2009). The Go Programming Language. Saatavilla osoitteesta: https://www.youtube.com/watch?v=rKnDgT73v8s.

(3) Pike R (2016). Saatavilla osoitteesta: https://research.google/people/r/. Haettu 10.2.2020.

(4) Ken Thompson (2020). Saatavilla osoitteesta: https://computerhistory.org/profile/ken-thompson/. Haettu 10.2.2020.

(5) Computer Hope staff (2017). Robert Griesemer. Saatavilla osoitteesta: https://www.computerhope.com/people/robert_griesemer.htm. Haettu 10.2.2020.

(6) Rupp K (25.6.2015). 40 Years of Microprocessor Trend Data | Karl Rupp. Haettu 9.2.2020.

(7) TIOBE authors (2019). Go | TIOBE - The Software Quality Company. Saatavilla osoitteesta: https://tiobe.com. Haettu 10.2.2020.

(8) TIOBE authors (2019). Go | TIOBE - The Software Quality Company. Saatavilla osoitteesta: https://tiobe.com/tiobe-index/go. Haettu 10.2.2020.

(9) Beuke F. GitHut 2.0 (2020). Saatavilla osoitteesta: https://madnight.github.io/githut/#/pull_requests/2018/2. Haettu 10.2.2020.

(10) Geek Authors (2019). Go Keywords. Saatavilla osoitteesta: https://www.geeksforgeeks.org/go-keywords/. Haettu 21.2.2020 sivustolta Geeksforgeeks.

(11) CppReference authors (2019). C++ keywords. Saatavilla osoitteesta: https://en.cppreference.com/w/cpp/keyword. Haettu 21.2.2020 sivustolta Cppreference.

(12) Geek Authors (2018). List of all Java Keywords. Available at https//www.geeksforgeeks.org/list-of-all-java-keywords/. Haettu 21.2.2020 sivustolta Geeksforgeeks.

(13) O'Reilly (12.9.2012). Why Learn Go? Saatavilla osoitteesta: https://www.youtube.com/watch?v=FTl0tl9BGdc. Haettu 11.2.2020.

(14) Pike R (25.10.2012). Go at Google: Language Design in the Service of Software Engineering. [PowerPoint-esitys]. Saatavilla osoitteesta: https://talks.golang.org/2012/splash.article. Haettu 13.2.2020.

(15) Pike R (7.7.2010). Go's Declaration Syntax. Saatavilla osoitteesta: https://blog.golang.org/gos-declaration-syntax. Haettu 13.2.2020.

(16) Donovan Alan A.A., Kernighan Brian W. (05.11.2015) The Go Programming Language.

(17) Gopher Guides (27.9.2019). Understanding defer in Go. Saatavilla osoitteesta: https://www.digitalocean.com/community/tutorials/understanding-defer-in-go. Haettu 13.2.2020 sivustolta DigitalOcean.

(18) Gopher Guides (6.11.2019). How To Use Interfaces in Go. Saatavilla osoitteesta: https://www.digitalocean.com/community/tutorials/how-to-use-interfaces-in-go. Haettu 14.2.2020 sivustolta DigitalOcean.

(19) Gopher Guides (4.9.2019). Creating Custom Errors in Go. Saatavilla osoitteesta: https://www.digitalocean.com/community/tutorials/creating-custom-errors-in-go. Haettu 14.2.2020 sivustolta DigitalOcean.

(20) Pike R (12.1.2015). Errors are values. Saatavilla osoitteesta: https://blog.golang.org/errors-are-values. Haettu 14.2.2020.

(21) Gerrand A (12.7.2011). Error handling and Go - The Go Blog. Saatavilla osoitteesta: https://blog.golang.org/error-handling-and-go. Haettu 14.2.2020.

(22) Neil D, Amsterdam J (17.10.2019). Working with Errors in Go 1.13. Saatavilla osoitteesta: https://blog.golang.org/go1.13-errors. Haettu 14.2.2020.

(23) Pike R (2012). Go Concurrency Patterns. [PowerPoint-esitys]. Saatavilla osoitteesta: https://talks.golang.org/2012/concurrency.slide#1. Haettu 17.2.2020.

(24) Hudosn R (12.7.2018). The Journey to Go’s Garbage Collector. Saatavilla osoitteesta: https://blog.golang.org/ismmkeynote. Haettu 17.2.2020.

(25) Awesome Go contributors (2019). Saatavilla osoitteesta: https://awesome-go.com/. Haettu 21.2.2020 sivustolta Awesome-go.

(26) Edwards J (3.11.2015). Why Go Was the Right Choice for CockroachDB. Haettu 25.2.2020 sivustolta Cockroach LABS.

(27) Iron Authors (2019). Saatavilla osoitteesta: https://www.iron.io/worker. Haettu 25.2.2020 sivustolta Iron.

(28) Stamat D (12.3.2013). How We Went from 30 Servers to 2: Go. Saatavilla osoitteesta: https://blog.iron.io/how-we-went-from-30-servers-to-2-go/. Haettu 25.2.2020.

(29) Graham-Cumming J (11.11.2013). What we've been doing with Go. Saatavilla osoitteesta: https://blog.cloudflare.com/what-weve-been-doing-with-go/. Haettu 27.2.2020.

(30) Gallagher S (26.2.2013). CloudFlare blows hole in laws of Web physics with Go and Railgun. Saatavilla osoitteesta: https://arstechnica.com/information-technology/2013/02/cloudflare-blows-hole-in-laws-of-web-physics-with-go-and-railgun/. Haettu 27.2.2020.

(31) Graham-Cumming J (21.12.2012). Railgun in the real world: faster web page load times. Saatavilla osoitteesta: https://blog.cloudflare.com/railgun-in-the-real-world/. Haettu 27.2.2020.

(32) Petazzoni J (7.11.2013). Why Go? [PowerPoint-esitys]. Haettu 26.2.2020.

(33) Beda J (11.11.2014). Kubernetes + Go = Crazy Delicious. Saatavilla osoitteesta: https://blog.gopheracademy.com/birthday-bash-2014/kubernetes-go-crazy-delicious/. Haettu 25.2.2020.

(34) Jenkins T (6.3.2014). How to Convince Your Company to Go With Golang. Saatavilla osoitteesta: https://sendgrid.com/blog/convince-company-go-golang/. Haettu 25.2.2020.

(35) Bourgon P (24.7.2012). Go at SoundCloud. Saatavilla osoitteesta: https://developers.soundcloud.com/blog/go-at-soundcloud. Haettu 25.2.2020.

(36) Twitch Authors (5.7.2016). Go’s march to low-latency GC. Saatavilla osoitteesta: https://blog.twitch.tv/en/2016/07/05/gos-march-to-low-latency-gc-a6fa96f06eb7/. Haettu 25.2.2020.

(37) Compton E, Salisbury C (2018). The Go Cloud Development Kit. Saatavilla osoitteesta: https://github.com/google/go-cloud. Haettu 3.3.2020 sivustolta GitHub.

(38) Molina M (2017). go-martini. Saatavilla osoitteesta: https://github.com/go-martini/martini. Haettu: 3.3.2020 sivustolta GitHub.

(39) Gin Team. Gin Web Framework. 2019; Saatavilla osoitteesta: https://gin-gonic.com/. Haettu: 3.3.2020 sivustolta Gin-Gonic.

(40) Soubachov B (2020). Testify. Saatavilla osoitteesta: https://github.com/stretchr/testify. Haettu: 3.3.2020 sivustolta GitHub.

(41) Tsenart (2020). Vegeta. Saatavilla osoitteesta: https://github.com/tsenart/vegeta. Haettu: 3.3.2020 sivustolta GitHub.

(42) Jinzhu (2020). ORM library for Golang. Saatavilla osoitteesta: http://gorm.io/. Haettu: 3.3.2020 sivustolta Gorm.

(43) Sherman M (2020). Gen. Saatavilla osoitteesta: https://github.com/clipperhouse/gen. Haettu: 3.3.2020 sivustolta GitHub.

(44) Volatiletech (2020). Authboss. Saatavilla osoitteesta: https://github.com/volatiletech/authboss. Haettu: 3.3.2020 sivustolta GitHub.

(45) Intellectsoft authors (28.6.2016). NodeJs vs Golang: Which is Best for Backend Development? Saatavilla osoitteesta: https://www.intellectsoft.net/blog/nodejs-vs-golang/. Haettu 27.2.2020.

(46) stackshare (2020). Go vs Node.js | What are the differences? Saatavilla osoitteesta: https://stackshare.io/stackups/go-vs-nodejs. Haettu 27.2.2020 sivustolta Stackshare.

(47) Bayburtsyan T (30.3.2017). 5 Reasons Why We switched from Python To Go. Saatavilla osoitteesta: https://hackernoon.com/5-reasons-why-we-switched-from-python-to-go-4414d5f42690. Haettu 27.2.2020.

(48) Benchmark Game (2019). Go versus Python 3 fastest programs. Saatavilla osoitteesta: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/go-python3.html. Haettu 26.2.2020 sivustolta BenchmarksGame.

(49) Joe (2017) Types of Java Garbage Collectors. Saatavilla osoitteesta: https://javapapers.com/java/types-of-java-garbage-collectors/. Haettu 27.2.2020.

(50) Benchmark Game (2019). Go vs Java - Which programs are fastest. Saatavilla osoitteesta: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/go.html. Haettu 26.2.2020 sivustolta BenchmarksGame

(51)Hundt R (2011). Loop Recognition in C++/Java/Go/Scala. Saatavilla osoitteesta: https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf. Haettu 27.2.2020.

(52) Metz C (1.7.2011). Google Go strikes back with C++ bake-off. Saatavilla osoitteesta: https://www.theregister.co.uk/2011/07/01/go_v_cpluplus_redux/. Haettu 27.2.2020.

(53) EDUCBA Authors (14.10.2018). Uses of C++ | 10 Reasons Why You Should Use C++. Saatavilla osoitteesta: https://www.educba.com/uses-of-c-plus-plus/. Haettu 2.3.2020 sivustalta EDUCBA.

(54) Scully E (11.1.2020). Go vs C++: A Comparison. Saatavilla osoitteesta: https://careerkarma.com/blog/go-vs-c-plus-plus/. Haettu 2.3.2020.

(55) Benchmark Game (2019). fannkuch-redux description. Saatavilla osoitteesta: https://benchmarksgame-team.pages.debian.net/benchmarksgame/description/fannkuchredux.html#fannkuchredux. Haettu 2.3.2020 sivustolta BenchmarksGame.

(56) Benchmark Game (2019). Go versus C++. Saatavilla osoitteesta: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/go-gpp.html. Haettu 2.3.2020 sivustolta BenchmarksGame.