Maisterintutkielma

Tietojenkäsittelytiede

Tulkin sulauttaminen monisäikeiseen ohjelmaan

Risto M. Mikkola

26.10.2020

Matematiikka-luonnontieteellinen tiedekunta

Helsingin yliopisto

Ohjaaja(t)

Prof. Tommi Mikkonen, Tri Antti-Pekka Tuovinen

Tarkastaja(t)

Prof. Tommi Mikkonen, Tri Antti-Pekka Tuovinen

Yhteystiedot

PL 68 (Pietari Kalmin katu 5)
00014 Helsingin yliopisto

Sähkopostiosoite: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytiede

Tekijä — Författare — Author

Risto M. Mikkola

Työn nimi — Arbetets titel — Title

Tulkin sulauttaminen monisäikeiseen ohjelmaan

Ohjaajat — Handledare — Supervisors

Prof. Tommi Mikkonen, Tri Antti-Pekka Tuovinen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Maisterintutkielma	26.10.2020	66 sivua

Tiivistelmä — Referat — Abstract

Ohjelmat koostuvat nykypäivänä useista eri kielillä tehdyistä osista. Eri kieliä käytetään niiden tarjoamien erilaisten etujen takia. Samassa ohjelmassa kieliä voidaan sekoittaa sulauttamalla yhden kielen tulkki toisella kielellä toteutettuun ohjelmaan, jolloin tulkin avulla yksi kieli voi suorittaa toisen kielen ohjelmia. Toimiakseen yhdessä kielten välille tulee rakentaa rajapinta, jonka avulla ne voivat kommunikoida. Kielet ovat jo tarkoitusperiensä perusteella eriävät, joten niiden välisen rajapinnan toteutuksessa on erilaisia yhteensopivuusongelmia. Lisäksi useiden sulautettavien kielten tulkit on suunniteltu vain yksisäikeiseen suoritukseen, joten niiden käyttö monisäikeisessä ohjelmassa voi vaarantaa ohjelman tehokkuuden tai eheyden.

Tässä työssä etsimme kielten välisen rajapinnan ja monisäikeisyyden ongelmia tulkin sulautuksessa ja niihin käytettyjä ratkaisuja kirjallisuuden avulla. Sulautamme käytännöntyönä Lua-kielen tulkin C++-kieliseen monisäikeiseen MMORPG-pelin palvelinohjelmaan. Sulautetun kielen käyttötapaus asettaa käytännön rajoja suoritusnopeudelle, joten rinnakkaisuuden säilyttäminen on ensiarvoisen tärkeää. Käytännöntyössä sulautetun kielen tulkista luodaan monta toisistaan eristettyä instanssia, joita voidaan suorittaa rinnakkain eri säikeillä. Käyttötapauksen takia instanssit joutuvat jakamaan tietoa keskenään, joka tapahtuu tiedon sarjallistamisen kautta.

Työn soveltava osa ja siinä käytettyjä ratkaisuja arvioidaan simuloimalla palvelinohjelman rakennetta ja suoritusta. Vertaamme tulkin yksisäikeistä sulautusta rinnakkaiseen toteutukseen ja toteamme, että testitilanteessa pääsemme haluttuihin melkein ideaalisiin suoritusnopeuksiin rinnakkaisuutta tukevan toteutuksen avulla. Nopeutuksen lisäksi toteamme muistinkulutuksen kasvun olevan vähäistä saatuihin hyötyihin nähden.

ACM Computing Classification System (CCS)

Software and its engineering → Software creation and management → Designing software → Software design tradeoffs

Computer systems organization → Real-time systems → Real-time system architecture

Computer systems organization → Architectures → Parallel architectures → Multicore

Avainsanat — Nyckelord — Keywords

tulkki, sulautus, monisäikeinen, palvelin, arkkitehtuuri, C++, Lua, tutkielma

Säilytyspaikka — Förvaringsställe — Where deposited

Helsingin yliopiston kirjasto

Muita tietoja — övriga uppgifter — Additional information

Ohjelmistojärjestelmien erikoistumislinja

Sisällys

1 Johdanto **1**

2 Isäntä- ja vieraskielen välinen rajapinta **3**

2.1 Sulautusmenetelmät 3

2.2 Muistinhallinta 4

2.3 Virheiden käsittely 5

2.4 Tietotyypit 6

2.5 Rajapinnan dokumentointi 7

3 Vieraskielen toiminta monisäikeisessä isäntäohjelmassa **9**

3.1 Johdanto monisäikeisiin ohjelmiin 9

3.2 Vieraskielen sulautus monisäikeiseen ohjelmaan 11

3.3 Tulkki-instanssien välinen tiedonjako 12

4 Isäntä- ja vieraskielen ominaisuudet: C++ ja Lua **14**

4.1 Tietotyypit ja tyypitys 14

4.2 Kielten syntaksi 15

4.3 Lua-kielen C-rajapinta 18

4.4 Muistinhallinta 19

4.5 Virheiden käsittely 20

4.6 Monisäikeisyys 21

5 Tapaustutkimuksen taustaa **22**

5.1 MMORPG 22

5.2 Palvelinohjelman arkkitehtuuri 24

5.3 Palvelinohjelman päivityssykli 26

5.4 Vieraskielen sulauttamisen vaatimukset 29

6 Toteutus **31**

6.1 Arkkitehtuuri 31
6.2 Tapahtumien käsittely 33
6.2.1 Tapahtumakutsun kulku 33
6.2.2 Eri tapahtumatyyppien käsittely 36
6.2.3 Lua-funktion liittäminen tapahtumaan 38
6.3 Tietotyyppien muunnos kielten välillä 40
6.4 Lua-instanssien välinen kommunikointi 42
6.4.1 Sarjallistaminen 43
6.4.2 Viestintäjärjestelmä 45
6.5 Virheiden käsittely 47

7 Arviointi **49**
7.1 Suorituksen vertailu 49
7.2 Vieraskielen sulauttamisen vaatimukset 53
7.3 Puutteet 54

8 Yhteenveto **56**

Kirjallisuus **59**

1 Johdanto

Ohjelmointikielillä on erilaisia ominaisuuksia. Tämä johtuu eroista niiden käyttötarkoituksissa, toteutuksissa, syntaksissa ja semantiikassa. Eri käyttötarkoituksilla on erilaisia rajoituksia ja tarpeita, joiden ympärille ohjelmointikieliä on suunniteltu ja toteutettu [27, 36]. Esimerkiksi on toteutettu aluekohtaisia kieliä (domain specific language, DSL), joiden tarkoitus on mahdollistaa ohjelmien tehokas ja nopea tuottaminen yhdellä ongelma-alueella [24]. Ohjelmointikielten ominaisuuksien, kuten tehokkuuden, takia ohjelmakirjastot on usein luotu tietyllä ohjelmointikielellä [11, s. 315-317]. Haluttuja kirjastoja ei siis voi välttämättä käyttää halutulla kielellä suoraan. Lisäksi yhdellä kielellä toteutetuissa ohjelmissa ongelmaksi voi osoittautua ohjelmointikielen tai ohjelman eri osien käytön rajoittaminen ohjelmoijilta virheellisen käytön estämiseksi [7].

Useaa ohjelmointikieltä yhdessä käyttämällä päästään käsiksi joukkoon ominaisuuksia, joita ohjelmointikielet eivät yksin toteuta ja voidaan valita ongelma-alueeseen sopiva ohjelmointikieli. Modernit ohjelmat koostuvat useista erityyppisistä ohjelmointikielistä, kuten käännetyistä, tulkatuista, funktionaalisista ja imperatiivisista kielistä. Yksi vallitseva tapa mahdollistaa kahden ohjelmointikielen käyttö saman ohjelman sisällä on toteuttaa ohjelma yhdellä kielellä, joka toimii ohjelman isäntäkielenä (host language), ja toteuttaa toisen kielen, eli vieraskielen (guest language), tulkki ohjelman isäntäkielellä [64, 7]. Tällöin vieraskielen tulkki on sulautettu (embedded) isäntäkieliseen ohjelmaan [64], jota kutsumme isäntäohjelmaksi.

Ohjelmointikielten eroavaisuudet aiheuttavat ongelmia kielten käyttöön yhdessä. Kieli voi esimerkiksi omata tietotyyppejä, joita ei toisessa kielessä ole ollenkaan, jolloin tietotyypin siirtäminen yhden kielen kontekstista toiseen ei ole suoraan mahdollista. Erilaisten kielten käyttö yhdessä vaatii ohjelmointikielten yhteentoimivuutta (interoperability) [36]. Ohjelmointikielten yhteentoimivuus määritellään usean ohjelmointikielen kyvyksi toimia osana samaa systeemiä. Yhteentoimivuuteen liittyvien ongelmien lisäksi monisäikeinen ohjelmointi tuo mukanaan monia ongelmia, joita perinteisissä yksisäikeisissä ohjelmissa ei tarvitse huomioida, kuten muistin samanaikainen käyttö useasta säikeestä [32].

Tässä työssä tutkimme kielten yhteentoimivuuden ja yhteiskäytön ongelmia ja ongelmien ratkaisuja tilanteessa, missä isäntäkielellä toteutettu monisäikeinen ohjelma voi suorittaa vieraskielellä toteutettuja ohjelmia saman prosessin sisällä sulautetun tulkin avulla.

Keskitymme seuraaviin tutkimuskysymyksiin.

1. Mitä ongelmia isäntäkielen ja sulautetun kielen vuorovaikutukseen liittyy ja mitä ratkaisuja niihin on käytetty?
2. Mitä ongelmia monisäikeisyys aiheuttaa isäntäkielen ja sulautetun kielen yhteentoimivuuteen ja mitä ratkaisuja niihin on käytetty?

Etsimme tutkimuskysymyksiin vastauksia kirjallisuudesta. Kirjallisuuden perusteella sovellamme löydettyä tietoa tapaustutkimuksen muodossa. Työn soveltavassa osassa sulautamme Lua-kielen tulkin monisäikeiseen C++-kieliseen palvelinohjelmaan. Soveltamisen avulla varmistamme, että keskeiset ongelmat on löydetty. Sen avulla kokeilemme myös joidenkin ratkaisujen käyttöä käytännössä, jolloin voimme arvioida niiden hyötyjä ja haittoja.

Opinnäytetyön rakenne on seuraava. Luvussa 2 käymme läpi kahden kielen välisen rajapinnan taustaa, eri osa-alueita, ongelmia ja ongelmiin käytettyjä ratkaisuja kirjallisuuden kautta. Luvussa 3 kuvailemme monisäikeisyyttä yleisesti, tuomme esiin monisäikeisyyden aiheuttamia ongelmia kahden kielen väliseen rajapintaan ja käymme läpi ongelmiin käytettyjä ratkaisuja. Luvussa 4 esittelemme tapaustutkimuksessa käytetyt kielet. Luvussa 5 esittelemme tapaustutkimuksen kohteen ja siihen tehdyn toteutuksen vaatimukset. Luvussa 6 käymme läpi tapaustutkimuksen toteutuksen arkkitehtuuria ja käytettyjä ratkaisuja. Luvussa 7 esittelemme toteutuksesta tehdyt mittaukset, arvioimme, täyttyivätkö toteutuksen vaatimukset, ja listaamme tutkimuksen puutteet. Luvussa 8 työ päätetään yhteenvedolla.

2 Isäntä- ja vieraskielen välinen rajapinta

Jotta erikieliset ohjelmat voivat toimia yhdessä, tulee niiden voida viestiä jollakin tavalla keskenään. Viestintää varten tulee kielten välille määrittää niille yhteinen rajapinta. Rajapinta voi toteuttaa yhteentoimivuuden eri tasoisesti. Kielet voivat esimerkiksi kyetä vain kutsumaan toistensa aliohjelmia, ne voivat pystyä viittaamaan toistensa arvoihin tai ne voivat muuntaa (translate) [7] arvoja toistensa välillä. Seuraavissa aliluvuissa käsittelemme isäntä- ja vieraskielen välisen rajapinnan toteutusmenetelmiä ja niihin liittyviä ongelmia ja ongelmien ratkaisuja.

2.1 Sulautusmenetelmät

Matthews et al. [37] ovat määrittäneet formaalisti kaksi sulautusmenetelmää. Könttäsulautuksessa (lump embedding) kielet voivat viitata toistensa arvoihin läpinäkymättömillä viitteillä (opaque pointers). Yhden kielen omistamaan arvoon luodaan uniikki viite, esimerkiksi kokonaisluku, jota sitten voidaan käyttää toisesta kielestä. Tällöin arvoja voidaan käyttää esimerkiksi osana funktiokutsuja käyttämällä viitettä arvon itsensä sijaan. Läpinäkymättömyydellä viitataan siihen, että viitteitä käyttävä kieli ei tiedä arvon toteutuksesta tai esitystavasta mitään. Könttäsulautus toteutetaan usein käärimällä (wrap) vieraan kielen olio, jolloin kääreen avulla viitattuun olioon voidaan liittää lisätietoa ja toiminnallisuutta. Arvoihin viittaamisen lisäksi kielten välillä halutaan kuitenkin usein siirtää tietoa, jotta yhden kielen arvoja voidaan käsitellä toisen kielen avulla. Luonnollisessa sulautuksessa (natural embedding) kielten väliseen rajapintaan määritetään muunnoksia kielten tyyppien välillä. Muunnosten avulla yhden kielen arvo muunnetaan toisen kielen arvoksi, jolloin sitä voidaan käyttää toisessa kielessä.

Bergel et al. [34] kuvailevat, että nämä tavat toteuttaa kielten välinen rajapinta sulautuksessa ovat laajasti hyväksyttyjä, mutta niihin liittyy haasteita ja rajoituksia. Kielten välisessä funktiokutsussa funktion parametreille tulee luoda viitteet tai ne tulee muuntaa toisen kielen arvoiksi. Muuntovaiheessa arvo voidaan kopioida kielestä toiseen, jolloin kulutetaan lisää muistia. Lisäksi erilaiset muunnosoperaatiot vievät aikaa, mikä voi ol-

la huomattava intensiivisessä suorituksessa. Esittäessä samaa arvoa eri kielissä voidaan haluta arvon päivittyvän kielten välillä jokaisen operaation seurauksena molempiin kieliin. Tällöin tarvitaan mahdollisesti monimutkaisia ja hitaita synkronointimekanismeja. Kopioinnin ja muunnosoperaatioiden välttämiseksi yhden kielen arvo voidaan kuvata toisessa kielessä todellisen arvon sijaan edustusolion (proxy) [18] avulla, joka sisältää viitteen todelliseen arvoon, johon kaikki edustusoliot viittaavat. Edustusoliot voivat viitata alkuperäisen kielen arvoon, jolloin arvon synkronointi kielten välillä tapahtuu suoraan arvoa muuttamalla. Kahta tai useampaa edustusoliota käytettäessä voidaan menettää viitteiden identiteetti, eli yhtäläisyys. Tämä johtuu siitä, että kieli voi tulkita eri edustusoliot eri arvoiksi, vaikka ne viittaisivat samaan olioon. Samaan olioon viitatessa edustusolioiden identiteetti kielen sisällä voidaan pyrkiä pitämään samana, jonka järjestäminen voi viedä suoritusaikaa ja muistia. Erityisesti aikaa voi kulua usein viitteidenhallintaan käytettyjen heikkojen viitteiden käsittelyyn roskienkerääjän (garbage collector, GC) [53] suorittaessa, joka poistaa arvoja, joihin ei enää viitata. Arvoja muunnettaessa kielestä toiseen on myös huomioitava erilaiset luokkahierarkiat ja niiden mahdollinen paljastaminen.

2.2 Muistinhallinta

Muistinhallinta käsittää muistin varaamisen ja sen vapauttamisen. Kaikki ohjelmien käyttämät muuttujat on sijoitettu muistiin suorituksen aikana. Muistia voidaan yleensä varata halutun kokoisia alueita. Varattuun muistiin voidaan usein liittää jokin luokka tai muu tyyppi, jonka avulla muistia voidaan varata, käyttää ja vapauttaa ohjelmassa kielen määritysten avulla helpommin. Ohjelman varaama muisti tulee jossakin vaiheessa vapauttaa, jotta sitä voidaan käyttää uudelleen. Jo vapautetun muistin käyttäminen on virhetilanne, joka voi johtaa erilaisiin tilanteisiin kuten ohjelman keskeytymiseen tai odottamattomaan ohjelman toimintaan [8][43, s. 78-79].

Muistinhallintamalli määrittää kuinka muistia voidaan varata ja vapauttaa. Eri kielet käyttävät erilaisia muistinhallintamalleja [45, §2.10][43, s. 313-314], joita voi olla hankala sovittaa yhteen. Ierusalimschy et al. [28] käsittelevät artikkelissaan, miten kielten tavanomainen rajapinta koostuu C-kielestä ja miten se aiheuttaa erilaisia rajoituksia sulautetun kielen rajapinnan toteutukselle. C-kieli ei tue monia moderneja konsepteja kuten automaattista muistinhallintaa. C-kielessä ohjelman keosta varaamien muistialueiden muistinhallinta jätetään kokonaan ohjelmoijan vastuulle, jolloin ohjelmoijan tulee ohjelmassaan eksplisiittisesti ohjelmassa varata ja vapauttaa haluamansa kokoiset muistialueet. Monis-

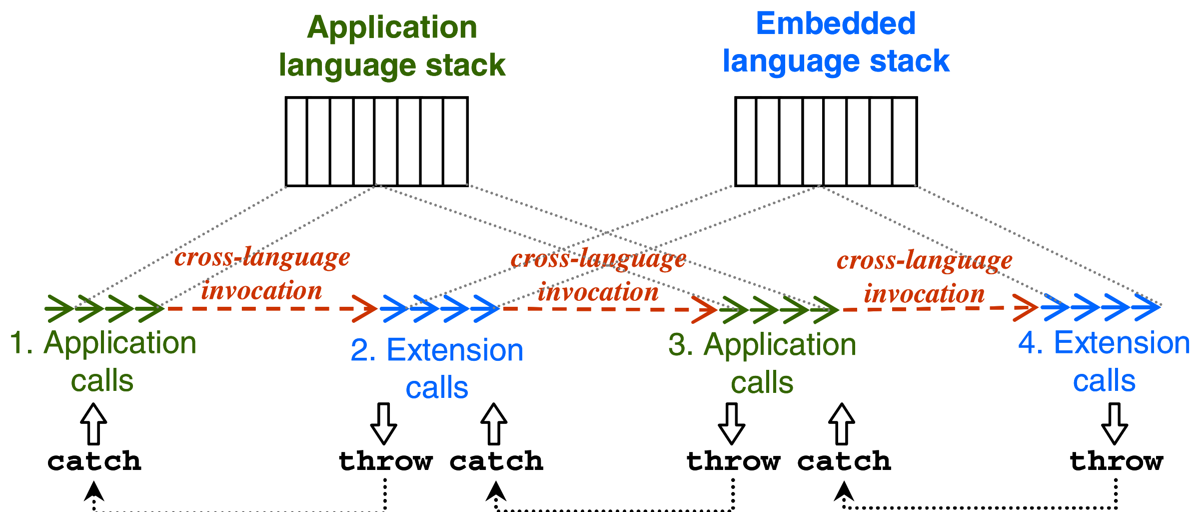
sa tulkatuissa kielissä, kuten Lua [53], muistinhallinta on automaattista. Käytännössä silloin muistia varataan luomalla esimerkiksi uusi muuttuja, joka viittaa tietorakenteeseen. Kielen tulkki varaa muuttujaa varten sen tarvitseman määrän muistia. Kielen tulkki voi tarvittaessa tarkistaa, onko olemassa varattua muistia, johon ohjelman muuttujien avulla ei enää päästä käsiksi ja vapauttaa sen.

Kun kieliä käytetään yhdessä, tulee toisen kielen muistinhallintamalli siis pitää mielessä sen arvoja käytettäessä, ellei sulautuksessa ole varmistettu muistin vapautuksen tapahtuvan vasta kun muistia ei enää tarvita. Muistinhallinta, joka varmistaisi muistin turvallisen käytön kaikissa tilanteissa, voi olla vaikea toteuttaa kielten muistinhallintamallien erojen takia. Automaattisen muistinhallinnan omaavissa kielissä ei yleensä voida viitata jo vapautettuun muistiin, sillä kielen ohjelmassa ei voi olla viitettä vapautettuun muistiin. Kuitenkin esimerkiksi edellä mainitun könttäsulautuksen avulla voidaan viitata toisen kielen omistamaan muistiin, joka voi olla jo vapautettu. Jotta kyseinen tilanne voitaisi estää, tulisi voida havaita milloin kielten ohjelmat eivät enää käytä niiden jakamaa muistia ja lisäksi kyetä vapauttamaan käyttämätön muisti havaitsemisen jälkeen.

2.3 Virheiden käsittely

Ohjelmissa tapahtuu virheitä ja virheiden käsittelylle on eri tapoja. Virheiden havaitseminen ja käsittely on tärkeää, sillä yleensä sulautettujen kielien virheiden ei haluta keskeyttävän koko ohjelman suoritusta [28]. Kielten välinen rajapinta on toteutettu usein C-kielellä, joka ei tue poikkeuksia (exception) [28]. Eri kielien poikkeukset tulee siis esimerkiksi kaapata (catch) ennen C-kieleen siirtymistä ja antaa C-kielen toteutukselle jossakin toisessa muodossa eteenpäin.

Savidis [47] toteaa, että erilaiset virheidenkäsittelymekanismit eivät ole aina yhteensopivia. Kuvassa 2.1 nähdään, miten isäntäkielisen ohjelman ja sulautetun vieraskielen ohjelman funktiokutsut muodostavat pinoja, joissa kieliä käytetään vuorotellen. Hän toteaa myös, ettei vieraskielestä nostettua poikkeusta (kuvan kohdat 2 ja 4) voida useimmissa toteutuksissa käsitellä isäntäkielen ohjelmassa suoraan, vaan se tulee käsitellä esimerkiksi kaappaamalla se kielten välisessä rajapinnassa kuvan mukaisesti. Kaappauksen jälkeen virhe nostetaan uudelleen toisen kielen virheenkäsittelymekanismin avulla, eli kuvassa yhden kielen virheilmoitus muunnetaan toisen kielen virheeksi. Virheet propagoidaan eteenpäin, kunnes ne kaapataan ja käsitellään. Savidis ehdottaa muutoksia sulautettaviin kieliin, jotka mahdollistaisivat poikkeusten käsittelyn eri kielten läpi ilman niiden muuttamista



Kuva 2.1: Isäntäkielen ja vieraskielen välisten kutsujen muodostaman pinon virheidenkäsittelykaavio [47].

kielten välisessä rajapinnassa.

2.4 Tietotyypit

Eri kielet voivat käyttää erilaisia tietorakenteita ja niiden toteutuksia. Kielissä voi olla myös erilaisia tietotyyppejä, kuten bittiesitykseltään erikokoisia kokonaislukuja. Edellä mainitun luonnollisen sulautuksen tavoin voidaan pyrkiä muuntamaan arvoja kielten välillä. Kielten yhteisille tyypeille löytyy usein suora muunnosoperaatio esimerkiksi sulautetun kielen tarjoaman C-rajapinnan kautta. Jotkin tietotyypit voidaan pyrkiä muuntamaan toisen kielen tietotyypiksi, jos ne ovat tarpeeksi lähellä toisiaan. Tietotyypit, joita ei voida tai haluta muuntaa toisen kielen arvoiksi voidaan esittää könttäsulautuksen avulla, jolloin olioihin voidaan viitata ja niiden ympärille voidaan rakentaa mahdollisuuksien mukaan erilaisia abstraktioita kääreiden ja edustusolioiden avulla.

Erilaisten kääreiden, muunnosoperaatioiden ja tyyppien määrittely kielten välille voi johtaa suureen määrään ns. liimakoodia (glue code) [60, 46]. Kielet voivat tukea heijastusta (reflection) [51, 16, 63], missä kieli pystyy tarkastelemaan omia rakenteitaan ja määrityksiään suorituksen aikana. Heijastusta voi käyttää apuna kielten välisen liimakoodista koostuvan rajapinnan toteuttamisessa [7]. Liimakoodi voidaan myös generoida automaattisesti erillisellä generaattorilla [5, 4] tai mallipohjaisen ohjelmoinnin avulla [38]. Erilaisilla generaattoreilla ja kirjastoilla, joilla liimakoodia voidaan luoda, voi kuitenkin

olla rajoituksia esimerkiksi usean paluuarvon käsittelyssä, jota kaikki kielet eivät tue [60].

Tietotyyppien muunnos kielten välillä ei välttämättä aina onnistu ja voidaankin esimerkiksi yrittää käyttää vääräntyyppisiä arvoja funktioiden parametreina. Tilanteessa tarvitaan tyyppien tarkastamista ja virheidenkäsittelyä turvallisen tyypityksen varmistamiseksi. Jacob Matthews et al. [37] esittää, että tyyppien tarkistus voi olla osana muunnosfunktiota tai että tarkistus voitaisi erottaa muunnosoperaatiosta erilliseksi kääreeksi, joka suorittaa tyyppitarkistuksen.

2.5 Rajapinnan dokumentointi

Jotta kehittäjät ja käyttäjät osaavat käyttää ohjelmaa, tulee ohjelman toiminta ja funktiot dokumentoida. Erilaiset tavat kommunikoida kielten välillä ja eri tavat käsitellä muistinhallinta, tietotyypit ja virheiden käsittely, johtavat erilaisiin rajapintoihin ja toimintatapoihin, joten yleisesti ei voida tietää rajapinnan toimintaa ilman dokumentaatiota. Dokumentaation puuttuessa käyttäjä voi joutua lukemaan lähdekoodia selvittääkseen, miten esimerkiksi muistinhallinta toimii tai mitä funktioita rajapinta tarjoaa. Dokumentaation puute voi johtaa rajapinnan väärinkäyttöön tai sen dokumentoimattomien osien vähäiseen käyttöön.

Muutokset rajapinnassa voivat pakottaa tekemään muutoksia rajapintaa käyttäviin ohjelmiin. Jotta ohjelmat toimivat oikein rajapinnan muutosten jälkeen, tulee rajapinnan muutokset tarkistaa ja huomioida sitä käyttävissä ohjelmissa. Rajapinnan eri versioiden dokumentaation avulla tarvitut korjaukset on mahdollista havaita ja tehdä. Dokumentaatio pyritään usein generoimaan automaattisesti. Automaattinen dokumentaation generointi on erityisen arvokas tilanteessa, jossa kielten välinen rajapinta muuttuu usein. Dokumentointi voidaan generoida automaattisesti monella tavalla, ja mahdolliset tavat vaihtelevat ympäristöstä ja kielistä riippuen. Jotkin kielet, kuten Java, tukevat esimerkiksi heijastusta, jolloin rajapinnan funktioita ja olioita voidaan tarkastella ja niistä tuottaa dokumentaatio ohjelman kääntämisen jälkeen.

Heijastuksen puuttuessa dokumentaatio voidaan generoida tulkitsemalla koodia, mutta tämä lähestymistapa on vahvasti sidottu käytettyyn kieleen ja voi olla hankala toteuttaa riippuen kielestä. Dokumentaatiogeneraattorit, kuten Doxygen [15] ja Javadoc [30], hyödyntävät koodiin upotettuja kommentteja ja merkintöjä. Useiden ohjelmointikielten kommentointityylit ovat samanlaiset, jolloin kommenteista ja kommentteihin liitetyistä merkinnöistä dokumentaation generoiva järjestelmä voi tukea useaa kieltä. Do-

kumentaatiogeneraattorit voivat hyödyntää useita edellä mainituista tekniikoista ja voivat tarjota esimerkiksi automaattista graafien generointia olioista ja niiden relaatioista analysoimalla dokumentaatiota ja koodia. Vaikka dokumentaatiogeneraattorit yrittävät soveltua erilaisiin dokumentointitarpeisiin, eivät ne aina sovellu ideaalisesti haluttuun käyttötarkoitukseen.

3 Vieraskielen toiminta monisäikeisessä isäntäohjelmassa

Jotta vieraskielen sulautus voidaan tehdä monisäikeiseen ohjelmaan ja samalla säilyttää monisäikeisyyden tuomat hyödyt, tulee ymmärtää sen hyödyt ja toiminta. Seuraavissa aliluvuissa käymme läpi monisäikeisyyttä yleisesti ja sen aiheuttamia ongelmia vieraskielen sulauttamiseen monisäikeiseen isäntäohjelmaan ja ongelmiin käytettyjä ratkaisuja.

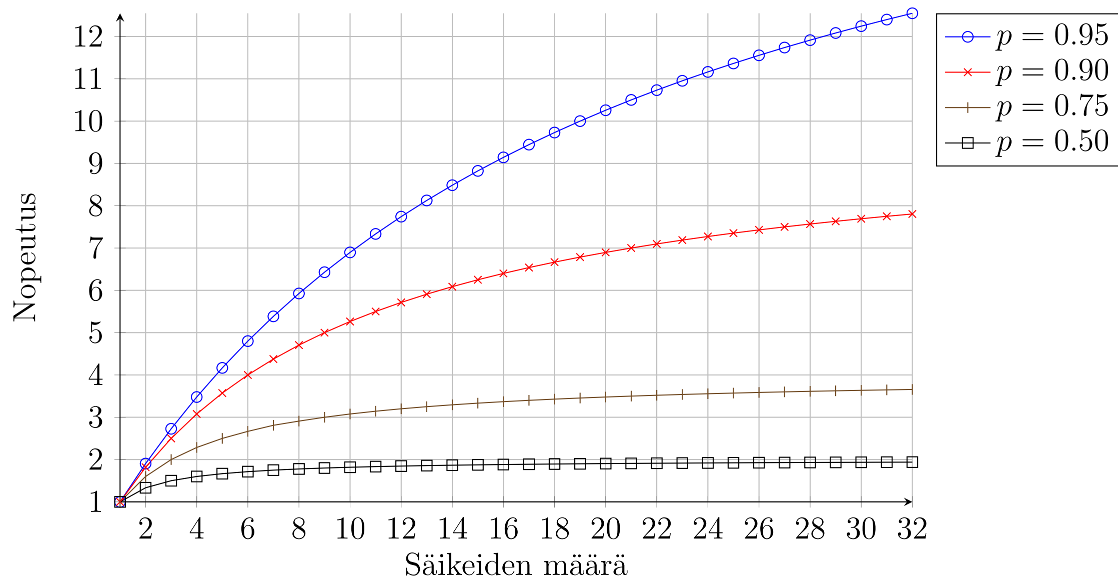
3.1 Johdanto monisäikeisiin ohjelmiin

Mooren lakia [39] mukaillen prosessorien ydinten laskentanopeus kasvaa vuosien kuluessa lineaarisesti, noin kahden kertoimella vuosittain. Suurta laskentatehoa vaativissa ohjelmissa voidaan kuitenkin tarvita esimerkiksi kymmenkertaista nopeutta saatavilla olevan prosessoriytimen nopeuteen nähden, jotta ohjelma toimisi tarvitulla tavalla. Tarvittu laskentateho voidaan saavuttaa jakamalla laskenta useaan osaan, jotka sitten lasketaan eri ytimillä rinnakkain. Teoriassa laskenta voitaisiin siis jakaa n itsenäiseen osaan, jotka kaikki suoritetaan omilla ytimillään, jolloin laskenta saataisiin valmiiksi $\frac{1}{n}$-kertaisessa ajassa. Käytännössä ohjelmia ei kuitenkaan aina voida jakaa itsenäisiin osiin, ja niissä tarvitaan jonkinlaista koordinointia, joka toteutetaan osin sarjallisesti.

Intuitiivisesti voisi ajatella, ettei ohjelman pienen osan yksisäikeinen suoritus välttämättä vaikuta ohjelman suoritusnopeuteen huomattavasti. Amdahlin lain [2] kautta voidaan pyrkiä arvioimaan teoreettisesti yksisäikeisen suorituksen vaikutusta sen ollessa osana monisäikeistä ohjelmaa. Laki voidaan esittää seuraavana kaavana:

$$S(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

Kaavassa $S(s)$ on nopeutus ohjelman rinnakkaisen osuuden ollessa p ja rinnakkaisen osuuden nopeutuksen ollessa s, eli rinnakkaisten säikeiden määrä. Kuvassa 3.1 nähdään, miten lain mukaan ohjelman rinnakkaisen osuuden vaihtelu voi vaikuttaa huomattavasti prosessoriytimistä saatavaan mahdolliseen hyötyyn. Esimerkiksi viiden prosentin sarjallinen osa ohjelmassa tiputtaa kahdeksasta ytimestä saadun nopeutuksen teoreettisesta kahdeksankertaisesta nopeutuksesta kuusinkertaiseksi, eli 25% pienemmäksi. Myös tutkimus toteaa,



Kuva 3.1: Ohjelman suorituksen latenssin teoreettinen maksiminopeutus sitä suorittavien prosessoriytimien lukumäärän funktiona Amdahlin lain mukaan eri rinnakkaisen osuuden p arvoille.

että rinnakkaisuuteen pyrkiminen on kriittistä hyvän nopeutuksen saavuttamiseen [22]. Erilaisten lakien käyttöä tehokkuuden pullonkaulojen tunnistamiseen on kuitenkin kritisoitu, sillä ne tuottavat hyvin erilaisia tuloksia [31]. Amdahlin laki ilmaisee vain teoreettisen maksiminopeutuksen, joten todelliset hyödyt tulee pyrkiä käytännössä mittaamaan.

Monisäikeiseen ohjelmointiin liittyy hyötyjen lisäksi myös monia ongelmia. Tutkimuksen [42] mukaan monisäikeisen ohjelmoinnin suosituimmat kysymyskategoriat olivat: (i) teoreettisten konseptit (ii) käytännön konseptit, (iii) ensiaskeleet, (iv) säikeiden elinkaari, (v) tekniset ongelmat, (vi) samanaikaiset kirjastot ja (vii) ohjelmien oikeellisuus. Jos monisäikeisen ohjelman suoritussäikeet voivat vaikuttaa toisiinsa, voivat ne aiheuttaa sivuvaikutuksia toistensa toimintaan kilpatilanteiden kautta (race condition) [14, 25]. Esimerkiksi säikeiden suorituksen riippuessa muuttujasta, jonka molemmat säikeet asettavat eri arvoksi, tapahtuu kilpailutilanne, jossa ohjelman suoritus riippuu säikeiden suoritusjärjestyksestä. Sivuvaikutukset tulee voida eliminoida. Säikeet voivat käyttää yhteisiä muistialueita ja säilyttää eheyden käyttämällä synkronointimekanismeja, kuten lukkoja, transaktioita ja atomisia muuttujia [19]. Erilaisten synkronointimekanismien käyttö kuitenkin vähentää ohjelman rinnakkaista osuutta, kasvattaa koodin määrää ja lisää sen monimutkaisuutta. Synkronointimekanismien käyttö voi esimerkiksi aiheuttaa uusia ongelmia, kuten ohjelman lukkiutumista (deadlock) [49]. Ohjelma lukkiutuu tilassa, missä esimerkiksi kaksi suoritussäiettä odottavat toistensa pitämien lukkojen vapautumista. Monisäikeisyydestä johtuvia ongelmia on vaikea diagnosoida, koska ne voivat riippua suori-

tuksen tietystä ajoituksesta eri säikeiden kesken. Ohjelmia on myös vaikea validoida, koska säikeiden suorittamien konekäskyjen suoritusjärjestysten määrä kasvaa eksponentiaalisesti säikeiden määrän kasvaessa, jolloin validoinnista tulee laskennallisesti haastava tehtävä [44]. Monisäikeisyydestä johtuvien ohjelmointivirheiden vähentämiseksi on kehitetty erilaisia algoritmeja [49], työkaluja [35], standardeja [12], ja jopa ohjelmointikieliä [65, 21].

3.2 Vieraskielen sulautus monisäikeiseen ohjelmaan

Monet kielet, kuten Lua, JavaScript, Python ja Ruby, voidaan sulauttaa ohjelmiin niiden tarjoamien C-rajapintojen ja eri kielillä toteutettujen tulkkien avulla. Nämä kielet ja siten myös niiden tulkit ovat suunniteltu vain yksisäikeiseen suoritukseen, jolloin ne eivät voi suorittaa ohjelmia rinnakkain montaa prosessoriydintä hyödyntäen [29, 59, 50, 17, 3]. Useasta säikeestä koostuvissa ohjelmissa, joihin on sulautettu yksisäikeiseen suoritukseen tarkoitettuja tulkkeja, tulee tulkilla suoritus suojata monen säikeen rinnakkaiselta suoritukselta. Python ja Ruby ratkaisevat ongelman globaalilla tulkkilukolla (global interpreter lock, GIL) [13], jolla tarkoitetaan lukkoa, joka päästää kerrallaan vain yhden suoritussäikeen suorittamaan tulkattua kieltä [59]. Samalla kuitenkin menetetään monisäikeisyyden tuomat edut prosessoreilla, joilla säikeet voisivat suorittaa tulkattavia ohjelmia rinnakkaisesti. Tulkkilukko suojaa tulkin sisäistä tilaa, joka jaetaan suoritettavien funktioiden kesken.

Tulkkilukkoa voidaan käyttää ratkaisuna monisäikeisyyden tuomaan ongelmaan tilanteissa, joissa tulkattu kieli ei ole pullonkaulana järjestelmän tehokkaalle toiminnalle. Esimerkki tällaisesta tilanteesta ovat ohjelmat, jotka käyttävät suuren osan ajastaan lukien tietoa levyltä laskennallisen ohjelmakoodin suorittamisen sijaan [33]. Esimerkin tilanteessa voidaan käyttää asynkronista tiedonhakua, jossa funktion suoritus keskeytetään tiedonhaun ajaksi. Tiedonhaun aikana muut säikeet voivat suorittaa tulkin avulla ohjelmia. Kyseisessä tilanteessa tulkin suoritus olisi yksisäikeistä, mutta tulkin tekemiä käskyjä suoritettaisi rinnakkain.

Tulkattavat ohjelmat eivät kuitenkaan välttämättä tee toimintoja, jotka hyötyisivät asynkronisuudesta, jolloin sen käyttö ei johda parempaan tehokkuuteen. Ohjelman alkuperäisen arkkitehtuurin ja sen tuomien etujen säilyttämiseksi suorituksen sarjallistamista monisäikeisessä ohjelmassa tulisi välttää. Jokaiselle suorittavalle säikeelle voitasi luoda oma instanssi tulkista, jolloin yksittäistä tulkkia suorittaa kerrallaan vain yksi säie. Ierusalimschy [26, s. 285-292] käyttää kyseistä lähestymistapaa esimerkissään Lua-

kielen käytöstä monisäikeisessä ohjelmassa. Ierusalimschy toteaa, että tulkin instansseilla on omat muistialueensa, jolloin ne eivät voi sotkea toistensa muistia ja pysyvät siten eheinä monisäikeisessäkin suorituksessa. Skyrme et al. [50] toteuttavat samanlaisen järjestelmän, mutta siinä tulkki-instanssit eivät ole säiekohtaisia. Sen sijaan tulkki-instanssit ovat jonossa, josta säikeet ottavat niitä yksi kerrallaan ja suorittavat instanssiin liitettyä ohjelmaa, kunnes se loppuu tai keskeytyy viestintäjärjestelmän seurauksena. Näin säikeiden määrä voi erota tulkki-instanssien määrästä ja jokainen instanssi pääsee lopulta suoritukseen.

Monissa kielissä voidaan asettaa globaalien muuttujien arvoja. Jos suorittava tulkki-instanssi vaihtuu, ei päästä enää käsiksi jo asetettuihin arvoihin. Funktionaaliset kielet ratkaisevat ongelman. Funktionaalisessa kielessä funktiot vain saavat arvoja ja palauttavat uusia arvoja, joten ne eivät muokkaa olemassa olevia arvoja [23]. Funktiot eivät siten myöskään voi muokata minkäänlaista globaalia tilaa. Seurauksena tästä funktioita voidaan suorittaa rinnakkain ja niitä voidaan suorittaa eri tulkki-instansseilla kerrasta toiseen, sillä ne eivät riipu instanssin säilyttämästä tilasta. Funktionaalisten kielten rajoitukset aiheuttavat kuitenkin ongelmia. Monet algoritmit mielletään jonkinlaisen tilan kautta, jolloin ne pitää pystyä muuttamaan funktionaaliseen muotoon, jotta niitä voidaan käyttää.

3.3 Tulkki-instanssien välinen tiedonjako

Usean tulkki-instanssin ongelmana on tiedon jakaminen instanssien välillä. Tulkki-instanssin suorittama ohjelma voi haluta siirtää tietoa toisella säikeellä tai myöhemmin samalla säikeellä suorittavalle tulkki-instanssille. Ierusalimschy [26, s. 285-292] käyttää esimerkissään tiedonsiirtoon synkronoitua viestinvälitystä, jossa eri säikeillä suorittavat tulkki-instanssit jäävät odottamaan viestin lähetettyään sen vastaanottamista ja vastaanottaessaan viestin saapumista. Esimerkissä viestintä tapahtuu listojen avulla, joiden kautta järjestelmä pääsee tallettamaan ja hakemaan viestejä niitä lähetettäessä ja vastaanottaessa. Viestijonot ovat jaettu eri säikeiden kesken, ja ne on suojattu lukolla, jolloin säikeidenvälinen viestintä on osin sarjallista. Ierusalimschyn ja Skyrme et al. [50] toteuttamissa järjestelmissä viestit voivat sisältää vain tekstiä tai tekstiä ja primitiivisiä tietotyyppejä, kuten lukuarvoja.

Skyrme et al. [50] ehdottavat tietotyyppien sarjallistamista tekstiksi siirtoa varten. Sarjallistamisella tarkoitetaan tiedon muuttamista varastoitavaan tai siirrettävissä olevaan muotoon. Tiedon sarjallistaminen, etenkin suuren tietomäärän, aiheuttaa instanssien tiedonvälitykselle ylimääräistä prosessointia. Prosessoinnin minimoimiseksi voidaan tieto kui-

tenkin pyrkiä sarjallistamaan vain kerran siirtoa varten, ja muuntaa vain osa tiedosta takaisin sarjallisesta muodosta halutuiksi tietotyypeiksi [58]. Tiedon osittainen sarjallistaminen asettaa kuitenkin erilaiset vaatimukset sarjallistetun tiedon muodolle, jolla on vaikutusta sarjallistetun tiedon kokoon ja sarjallistamisen nopeuteen, joten todelliset hyödyt tulee mitata ja arvioida käyttötapauksen mukaan. Esimerkiksi jos sarjallistettu tieto käytetään vain kerran ja kokonaan, sitä käytettäessä voi tiedon osittaisen purkamisen toteuttamisesta olla vain haittaa sarjallistetun tiedon viemän tilan, tiedon purkamiseen ja sarjallistamiseen kuluvan ajan ollessa suurempia.

Jos sarjallistettua tietoa yritetään käyttää useasta säikeestä, tulee sarjallistettu tieto suojata, ettei sitä muokattaessa tai luettaessa tapahdu kilpatilannetta. Suojaus voidaan toteuttaa lukoilla, mutta myös lukottomien järjestelmien avulla, kuten käyttämällä ohjelmallista transaktionaalista muistia (software transactional memory) [48], jonka Clojure ohjelmointikieli toteuttaa*. Siinä muutokset tietoon talletetaan erilleen ja sulautetaan tietoon myöhemmässä vaiheessa. Näin voidaan saavuttaa yhtäaikaista lukemista ja kirjoittamista, mutta kirjoittamisen jälkeen luettu arvo toisessa säikeessä ei välttämättä ole heti rakenteeseen kirjoitettu arvo.

*https://clojure.org/reference/refs haettu 30 syyskuuta 2020

4 Isäntä- ja vieraskielen ominaisuudet: C++ ja Lua

Työn soveltavassa osassa sulautamme C-kielellä toteutetun Lua-kielen C++-kieliseen ohjelmaan. Lua-kielestä käytämme kielen versiota 5.1, koska se on yhteensopiva Lua-kielen ajonaikaisen kääntäjän kanssa [41]. Käytämme C++-kielen versiota C++17, koska käytetyt kirjastot vaativat sitä. C++-kieli pohjautuu C-kieleen ja tarjoaa sen toimintojen lisäksi esimerkiksi luokat (class), mallit (template), poikkeukset (exception), viitteet (reference) ja uusia tietotyyppejä [52, s. 1]. Käymme seuraavissa aliluvuissa läpi käytettävien kielten sulauttamisen kannalta keskeiset ominaisuudet ja koodiesimerkkien kannalta lukijalle olennaiset tiedot.

4.1 Tietotyypit ja tyypitys

Lua-kielessä on dynaaminen tyypitys [45, §2.2], jossa muuttujan tyyppi voi vaihdella suorituksen aikana sen pitämän arvon mukaan. Lua-kielessä kaikentyyppisiä arvoja voi talletella muuttujiin ja käyttää funktion parametreina, funktion paluuarvoina, tai assosiaatiotaulun avaimena tai arvona. Lua-kielessä on olemassa tietotyypit `number`, `boolean`, `string`, `table`, `nil`, `function`, `userdata`, `lightuserdata` ja `thread` [45, §2.2, §3.7]. Kaikki lukuarvot esitetään Lua-kielessä 64-bittisenä liukulukuna. Tyypin `nil` arvo on tyhjä arvo ja tarkoittaa arvon puuttumista, `lightuserdata`-tyyppi esittää tyypitöntä osoitinta (void pointer). Tyypin `thread` arvo osoittaa Lua-kielen korutiiniin (coroutine), joka on funktio, jota voi keskeyttää ja jatkaa tarpeen mukaan [45, §2.11]. Tyypin `function` arvo, eli funktio, voi esittää Lua-funktiota tai C-kielen funktiota, jonka viite on talletettu Lua-kielen ympäristöön. Lua-kielen funktiotyypin arvoon voidaan liittää muita arvoja, joihin päästään käsiksi funktiokutsun yhteydessä [45, §3.4], eli se on sulkeuma (closure) [57]. Lua-taulu, eli `table`-tyypin arvo, on Lua-kielen ainoa tietorakenne. Lua-taulu on assosiaatiotaulu (associative array) [6], eli avain-arvo-tietorakenne, jossa avaimena ja arvona voi toimia `nil`-tyypin arvoa lukuun ottamatta mikä tahansa arvo [29]. Lua-tauluja voi käyttää myös taulukon (array) tavoin, jolloin assosiaatiotaulun avaimet ovat numeroita. Lua-kielessä voidaan esittää mielivaltaista C-kielen tietoa `userdata`-tyyppistä arvoa

käyttämällä. Sen avulla voidaan tallettaa esimerkiksi C++-olion osoitin (pointer) tai koko olio itsessään Lua-tulkin hallitsemaan muistiin. Kyseisen tyyppiseen arvoon voidaan liittää metataulu (metatable) [45, §2.8], joka on Lua-taulu ja jonka ennalta määrätyille avaimille talletetut arvot voivat olla funktioita. Näitä funktioita kutsutaan eri operaattorien, kuten laskutoimitus- ja indeksointioperaattorien käytön yhteydessä. Metataulu voidaan liittää myös toisiin Lua-tauluihin.

C++-kieli on vahvasti tyypitetty kieli [52, s. 9, 77–83], jossa kaikilla muuttujilla, funktion parametreilla ja funktion paluuarvoilla on tietty ohjelmoijan antama tyyppi. C++-kielisten ohjelmien koodi käännetään konekieleksi, joka voidaan sitten suorittaa. Kääntämisen aikana kääntäjä tarkastaa ohjelmakoodiin kirjoitettujen tyyppien avulla, että ohjelma on validi. Kääntäjä käyttää tyyppejä myös apuna konekielisen koodin generoinnissa. Tyyppien avulla kääntäjä voi tietää jokaisen muuttujan vievän tilan muistissa ja siten optimoida generoitua koodia ja vaikuttaa lopullisen ohjelman tehokkuuteen. C++-kielen tyypit voidaan jakaa perustyyppeihin (fundamental types) ja yhdistetyyppeihin (compound types). Perustyypit käsittävät erikokoiset kokonais- ja liukulukutyypit, Boolen arvot ja tyhjän tyypin (void). Perustyyppiset lukutyypit ovat kooltaan maksimissaan 64-bittisiä ja kokonaislukutyypit voivat olla lisäksi etumerkillisiä tai etumerkittömiä. Yhdistetyypit voivat koostua useasta perustyypistä ja käsittävät lisäksi funktiot ja erilaiset viitetyypit, kuten viitteen (reference) ja osoittimen. Yhdistetyyppejä ovat esimerkiksi luokat, merkkijonot (string), ja säiliöt (containers), kuten assosiaatiotaulut ja taulukot. C++-kielen tyypit voivat omata `const`-määrityksen, joka tarkoittaa, ettei kyseisen tyyppistä arvoa voi muokata [52, s. 82-83].

4.2 Kielten syntaksi

Lua-kielessä on kahdenlaisia muuttujia: globaaleja ja lokaaleja [45, §2.3]. Kaikki globaalit muuttujat talletetaan todellisuudessa Lua-tauluun, jota indeksoidaan globaalia muuttujaa käytettäessä. Lua-funktioihin liitetään kyseinen taulu, jolloin ne pääsevät globaaleihin muuttujiin käsiksi. Muuttujat, jotka on erikseen merkitty avainsanalla `local`, ovat lokaaleja, eikä niihin voida viitata ennen niiden lokaaliksi määrittämistä eikä niiden näkyvyysalueen (scope) ulkopuolelta.

```
1  t = {"foo", ["key"]="value", bar="baz", 42}
2  t = {[1]="foo", ["key"]="value", ["bar"]="baz", [2]=42}
3
4  t["k"] = v
5  t.k = v
6
7  t.f(t)
8  t:f()
9
10 for k,v in f, s, v do
11 end
12
13 function f(x, y, z)
14     return x, x+y, y+z
15 end
16 a,b,c = f(1,2,3,4,5)
```

Listaus 4.1: Esimerkkejä Lua-kielen notaatiosta.

Lua-tauluja voidaan muodostaa aaltosulkuja käyttämällä [45, §2.5.7]. Aaltosulkujen sisään voidaan listata pilkulla eriyttäen taulun avain-arvo-pareja. Parien keskellä on yhtäsuuruusmerkki, jonka oikealla puolella on arvo ja vasemmalla puolella on joko hakasulkujen sisällä avaimena käytetty arvo tai ilman hakasulkuja avaimena käytetyn merkkijonon merkit. Yhtäsuuruusmerkin ja avaimen voi myös jättää pois, jolloin avaimena käytetään vasemmalta oikealle avaimettomille arvoille kokonaislukujonoa ykkösestä ylöspäin. Listauksen 4.1 riveillä 1 ja 2 on esimerkit kahdesta ekvivalentista taulukon luomisesta, jotka asetetaan muuttujaan t.

Lua-taulua voidaan indeksoida hakasulkuoperaattorin tai pisteoperaattorin avulla listauksen 4.1 rivien 4 ja 5 näyttämällä tavalla [45, §2.2, §2.3]. Listauksessa t on Lua-taulu, jota indeksoidaan merkkijonolla ”k”, ja v on arvo, joka asetetaan tauluun sijoitusoperaattorin avulla. Lisäksi Lua-tauluun talletettuja funktioita voidaan kutsua kaksoispistenotaatiolla, joka helpottaa olio-ohjelmointityylisiä funktiokutsuja [45, §2.5.8]. Listauksen 4.1 riveillä 7 ja 8 näkyvät esimerkit pisteoperaattorilla ja kaksoispistenotaatiolla tehdyistä ekvivalenteista funktiokutsuista. Koodissa t on Lua-taulu, johon on talletettu funktio merkkijonoavaimella ”f”, jota kutsutaan sulkujen avulla. Molemmissa esimerkeissä indeksoitava arvo asetetaan ensimmäiseksi parametriksi kutsuttavaan funktioon, jolloin funktiossa päästään käsiksi olioon. Kaksoispistenotaatiossa se tapahtuu implisiittisesti.

Listauksen 4.1 riveillä 10 ja 11 nähdään *for*-lause [45, §2.4.5], joka suorittaa `do` ja `end` avainsanojen välissä olevan koodin, kunnes funktio f palauttaa `nil`-arvon. Funktio voi

palauttaa jonkin määrän arvoja, jotka voidaan nimetä pilkulla eroretulla listalla muuttujanimiä ja siten niihin voidaan päästä käsiksi *for*-lauseen sisällä. Listauksessa funktion odotetaan palauttavan kaksi arvoa, jotka asetetaan muuttujiin a ja b. Funktiota f kutsutaan aluksi arvoilla s ja v, mutta sittemmin arvoilla s ja funktion ensimmäisellä paluuarvolla. Käytännössä arvojen f, s ja v avulla voidaan toteuttaa iterointifunktio.

Lua-kielen funktiot voivat ottaa sisäänsä ja palauttaa mielivaltaisen määrän parametreja ja paluuarvoja. Listauksen 4.1 riveillä 13-16 nähdään esimerkki funktiosta f, joka ottaa vastaansa kolme parametria ja palauttaa kolme paluuarvoa. Paluuarvot asetetaan sitten muuttujiin a, b ja c. Funktiota kutsutaan esimerkissä viidellä parametrilla, mutta ylimääräiset parametrit jätetään käyttämättä. Jos kutsussa on liian vähän parametreja, korvataan loput parametrit `nil`-tyypin arvoilla.

C++-kielen muuttujien yhteydessä täytyy merkitä niiden tyyppi. Eksplisiittisen tyyppimäärityksen sijaan ohjelmoija voi antaa kääntäjän päätellä tyyppejä `auto`-avainsanan avulla [52, s. 171-174]. C++-kielen viite ja osoitin antavat tavan päästä käsiksi jo varattuun muistiin ja ne eroavat merkintätavoiltaan. Arvoon viittaava muuttuja on merkitty arvon tyypin lisäksi &-merkillä [52, s. 206-207]. Osoitin arvoon on merkitty arvon tyypin lisäksi *-merkillä ja osoittimia voi luoda liittämällä &-operaattori muuttujan vasemmalle puolelle [52, s. 207-208]. Listauksen 4.2 riveillä 1-3 nähdään esimerkki olion luonnista sen kasaajaa kutsumalla, sekä viitteen ja osoittimen luonnista kyseiseen olioon.

```
1  Object o();
2  Object& o_reference = o;
3  Object* o_pointer = &o;
4  
5  o.member();
6  o_reference.member();
7  o_pointer->member();
8  
9  Object::static_function();
10 Object::template_function<int>(1);
11 Object::template_function<float, int>({1,2,3});
```

Listaus 4.2: Esimerkkejä C++-kielen notaatiosta.

Olion jäsenmuuttujia (member variable) käytetään pisteoperaattorin avulla, ja samaa operaattoria voidaan käyttää olioon viittaavan viitteen yhteydessä kutsumaan olion metodeja [52, s. 130-131]. Olioon viittaavan osoittimen kautta metodikutsut tehdään nuolioperaattorin avulla. Esimerkit metodikutsuista nähdään listauksen 4.2 riveillä 5-7. Olion metodin

sisällä `this`-muuttuja on osoitin ja viittaa olioon itseensä [52, s. 96-97]. C++-kielessä voidaan ohjelmassa määrittää muuttujien ja arvojen alustusarvoja aaltosulkujen avulla [52, s. 220-237]. Alustusarvoilla voidaan luoda helposti esimerkiksi listoja alkioineen, kuten voidaan nähdä listauksen 4.2 rivillä 11.

Kielen eri hierarkiat, kuten sisäkkäiset luokat ja nimitilat (namespace) erotetaan toisistaan kahden peräkkäisen kaksoispisteen avulla [52, s. 178-191, 241–252]. Kielen funktioita voidaan ylimääritellä (overload), eli niille voidaan luoda useita erilaisia määrityksiä, jotka riippuvat parametrien tyypeistä [52, s. 311-357]. Myös olioiden operaattoreita voidaan ylimääritellä.

C++-kielessä voidaan käyttää metaohjelmointia (meta programming), jossa ohjelma voi käyttää ohjelmia tai ohjelman palasia osana käsiteltävää tietoa [10]. C++-kielen metaohjelmointi perustuu malleihin (templates) [52, s. 344-424] ja sitä kutsutaan mallipohjaiseksi metaohjelmoinniksi (template metaprogramming) [1]. Mallipohjaisen metaohjelmoinnin avulla ohjelmoija voi määritellä malleja funktioille, luokille ja muuttujille, jotka voivat riippua kyseisten mallien käytön yhteydessä annetuista malliparametreista (template parameters). Kääntäjä voi usein päätellä malliparametrit, mutta ne voi tai täytyy antaa `<` ja `>` merkkien ympäröimänä pilkulla erotettuna listana. Mallien avulla voidaan toteuttaa esimerkiksi funktioita, jotka voivat ottaa sisäänsä mielivaltaisen tyyppisiä ja mielivaltaisia määriä parametreja ja voivat havaita parametrien tyypit ja muuttaa toimintaansa niiden mukaan. Listauksen 4.2 rivillä 9 nähdään esimerkki staattisesta funktiokutsusta, jossa funktioon päästään käsiksi kaksoispistenotaation avulla. Rivillä 10 nähdään staattisen templaattifunktion kutsu, jossa templaattiparametri on annettu eksplisiittisesti. Rivillä 11 nähdään kuinka samanniminen templaattifunktio, jolla on eri templaattiparametrit ja eri parametrit, joten `template_function` on siis ylimääritelty. Lisäksi voidaan huomata, etteivät templaattiparametrit ole sidoksissa funktion parametreihin.

4.3 Lua-kielen C-rajapinta

Lua-kieli tarjoaa C-kielisen rajapinnan [45, §3], jonka avulla muut kielet voivat toimia sen kanssa yhdessä ja jotta kielet voivat viestiä keskenään. Rajapinta paljastaa arvot virtuaalisen pinon kautta [45, §3.1], jonka avulla Lua-kielen arvoihin voi viitata niiden pinoindeksin mukaan. Rajapinnan funktioiden [45, §3.7] avulla voidaan myös tehdä pysyviä Lua-viitteitä pinon arvoihin, vaikka ne eivät olisi enää nähtävissä pinon avulla. Näiden `integer`-tyyppisten viitteiden avulla Lua-kielen muuttujia voidaan tallettaa ja käyttää.

Kaikki Lua-kielen tilatieto talletetaan Lua-kielen tulkin instanssia kuvaavaan tietorakenteeseen eli Lua-instanssiin.

Pinoon voi lisätä tai siitä voi poistaa arvoja rajapinnan funktioiden avulla. Funktiot antavat asettaa pinoon tietyntyyppisiä C-kielen arvoja, jotka muunnetaan suoraan Lua-kielen arvoiksi, tuetut arvot ovat boolean, merkkijono, 64-bittinen liukuluku, 32-bittinen kokonaisluku, tyypitön osoitin (void pointer) ja funktio. Lisäksi pinoon voi asettaa Lua-kielen arvoja, esimerkiksi `nil`-arvon. Pinoon lisätyillä C-kielen funktioilla tulee olla tietty tyyppimuoto. Käytännössä funktioille tulee siis tehdä käärefunktioita, jotta ne voidaan paljastaa Lua-kieleen. Käärefunktiot voidaan tehdä suurelta osin automaattisesti mallipohjaisen ohjelmoinnin avulla [38] tai esimerkiksi luomalla ne ennen ohjelman kääntämistä ohjelmallisesti [5, 4].

Lua-kielen C-rajapinta tarjoaa monia erilaisia funktioita [45, §3.7]. Rajapinta tarjoaa funktion käytännössä kaikille kielen eri operaatioille, kuten funktioiden kutsumiseen, virheiden nostamiseen, Lua-taulujen luomiseen ja käsittelyyn, metataulujen luomiseen, ja tyyppien tarkistamiseen. Lisäksi rajapinta antaa funktiot Lua-instanssien luomiseen ja hajottamiseen, monien C-kielen arvojen muuntamiseen Lua-kielen arvoiksi ja takaisin C-kielen arvoiksi, Lua-viitteiden luomiseen ja vapauttamiseen, ja C-rajapinnan pinon käsittelyyn.

4.4 Muistinhallinta

Lua-kielen ja C++-kielen muistinhallinta eroavat toisistaan. Lua-kielessä käytettävä roskienkeräys (garbage collection) [45, §2.10] etsii ajoittain arvot, joihin ei voida enää viitata ja vapauttaa niiden käyttämän muistin. Kaikki Lua-kielen arvot voidaan vapauttaa roskienkerääjän toimesta. Koska roskienkeräys on ainoa tapa vapauttaa arvoja, voidaan päätellä, ettei kielessä voi koskaan viitata muistiin, mitä ei ole olemassa. Jos arvolla on metataulu, roskienkerääjä kutsuu sen `__gc`-merkkijonolla talletettua funktiota ennen arvon poistoa parametrina arvo itse. Lua-kielessä Lua-taulut, `userdata`-arvot, funktiot ja korutiinit ovat viitteitä, joiden todelliset arvot ovat talletettuna jonnekin muistiin [45, §2.2]. Tästä syystä näiden arvojen käsittely ei koskaan luo arvosta kopiota vaan vain uuden viitteen. Kaikki muut arvot kopioidaan aina niitä käsitellessä, esimerkiksi käytettäessä niitä funktion parametreina tai tallettamalla ne muuttujaan.

C++-kielen standardi määrittää objekteiksi kaikki muistia varaavat arvot, joita kielen avulla kasataan, muokataan, ja hajotetaan [52, s. 9]. Objekti-käsite kattaa kaikki perustyypit ja yhdistetyypit, mutta funktiot eivät ole objekteja varasivat ne muistia tai eivät.

Objekteilla on erilaisia säilytysaikoja (storage duration) ja säilytysaika määräytyy objektin luontitavan perusteella [52, s. 70-71]. Staattisen säilytysajan (static storage duration) omaavat muuttujat ovat olemassa koko ohjelman suorituksen ajan. Säie-säilytysajan (thread storage duration) omaavat muuttujat ovat olemassa vain säikeen olemassaolon ajan. Automaattisen säilytysajan (automatic storage duration) omaavat muuttujat ovat olemassa niiden määrittämän yhdistelauseen (compound statement, block) [52, s. 145] keston ajan, eli käytännössä ne vapautetaan, kun muuttujan näkyvyysalue loppuu. Dynaamisen säilytysajan (dynamic storage duration) omaavia *objekteja* luodaan ja vapautetaan suorituksen aikana `new`- ja `delete`-lausekkeiden avulla. C++-kielen objekteja voidaan siis luoda ja tuhota ohjelmoijan toimesta eksplisiittisesti. Muuttujiin voidaan viitata muuttujanimien ja viitteiden sekä osoittimien kautta, joiden avulla olioon voidaan yrittää viitata sen vapauttamisen jälkeenkin. Jotta muistia ei voida vahingossa jättää vapauttamatta, voidaan käyttää olio-ohjelmoinnin RAII-idiomia (resource acquisition is initialization) [55, s. 383-389]. Kyseisessä idiomissa luodaan olio, joka varaa resurssin ja vapauttaa sen tuhoutuessaan.

4.5 Virheiden käsittely

Lua-kielisen ohjelman virhetilanteessa suoritus palautuu Lua-kieltä kutsuvalle ohjelmalle, jolla on pääsy virheviestiin ja voi toimia ohjelmoijan haluamalla tavalla [45, §2.7]. Lua-ohjelman suoritus kuitenkin keskeytyy, eikä keskeytyneen Lua-funktion suoritukseen voida palata takaisin. Lua-ohjelma voi aiheuttaa virheitä `error`-funktion avulla. Lua-ohjelma voi myös itse käsitellä virheitä käyttämällä `pcall`-funktiota, jonka avulla suoritetut funktiot palauttavat virhetilanteessa kaksi arvoa: `nil`-arvon ja virheviestin [45, §5.1].

Työssä käytetyn Lua-kielen tulkin toteutukseen käytetyssä C-kielessä ei ole poikkeuksia. Tulkin toteutus käyttää sisäisesti C-kielen hyppykirjastoa (jump library) [43, s. 243-245], jonka avulla suoritus voidaan palauttaa aiemmin merkattuun pisteeseen [45, §3.6]. C++-kielen standardin mukaan hypyn käyttö on kuitenkin määrittämätöntä, jos poikkeuksen käyttö hypyn sijaan vapauttaisi esimerkiksi RAII-idiomin olion, jonka tarkoitus on vapauttaa muistia [52, s. 511]. Tulkin toteutuksen voi kääntää myös C++-kielenä, jolloin tuen poikkeuksille voi tarvittaessa lisätä [45, §3.6]. Lua-kieleen voidaan paljastaa C-funktioita, ja sitä kautta myös C++-funktioita, jotka voivat aiheuttaa poikkeuksia. Kielen manuaali ei kerro, mitä tapahtuu, kun kieli on käännetty C++-kielenä poikkeustuen kanssa ja paljastettu funktio heittää poikkeuksen suorituksen aikana.

C++-kielessä virhetilanteessa voidaan heittää (throw) poikkeus [52, s. 424-433]. Poikkeuksen heitto siirtää ohjelman suorituksen ohjelman aiempaan osioon, missä on käytetty poikkeuksen kaappaavaa lauseketta. Kyseiseen lausekkeeseen kuuluu poikkeuksen käsittelijä, johon suoritus siirtyy kaapattaessa poikkeus, ja jossa voidaan päästä käsiksi heitettyyn poikkeukseen. Poikkeus voi sisältää virheviestin ja lisäksi muutakin tietoa virheeseen liittyen riippuen poikkeuksesta. Suorituksen siirron aikana kaikki automaattisen säilytysajan omaavat arvot vapautetaan, ja samalla ne voivat vapauttaa myös dynaamisen säilytysajan omaavia arvoja RAII-idiomin avulla.

4.6 Monisäikeisyys

Lua-kieli ei tarjoa minkäänlaista monisäikeisyyttä, vaan kieli on suunniteltu toimimaan vain yksisäikeisesti [29]. Lua-kielen `thread`-tyyppisiä arvoja ei tule sekoittaa säikeisiin tai rinnakkaisuuteen (parallelism), sillä ne toteuttavat samanaikaisesti (concurrently) suoritettavia korutiineja (coroutine) [40][45, §2.11]. Kielen avulla ei voida tehdä rinnakkaista laskentaa, eikä kielen tulkkia voida käyttää useasta säikeestä samanaikaisesti turvallisesti. Lua-kielen tulkista voidaan kuitenkin tehdä useita toisistaan eristettyjä instansseja, joilla voidaan suorittaa Lua-koodia eri säikeillä rinnakkain [50].

C++-kieli tukee rinnakkaisten säikeiden luontia ja suoritusta, ja määrittelee erilaisia operaatioita, joiden avulla rinnakkaiset säikeet voivat kommunikoida [52, s. 15-20]. C++-kieli ja sen standardikirjasto määrittelevät atomisia operaatioita ja operaatioita erityisille lukko-olioille (mutex). Nämä operaatiot ovat synkronoituja eli sarjallisia operaatioita. Lukkojen ja atomisten operaatioiden avulla voidaan suojautua kilpatilanteilta ja siten jakaa tietoa kahden säikeen välillä.

5 Tapaustutkimuksen taustaa

Työn soveltava osa keskittyy tapaustutkimukseen, joka käsittelee Lua-kielen tulkin lisäämistä C++-kielellä ohjelmoituun palvelinohjelmaan. Seuraavissa aliluvuissa keskitymme tapaustutkimuksen kohteena olevan palvelinohjelman toimintaan. Esittelemme palvelinohjelman tukemien pelien keskeiset käsitteet ja toiminnan, palvelinohjelman arkkitehtuurin, ja käsittelemme palvelinohjelman tehokkuutta tutkielmalle oleellisin osin. Käymme myös läpi Lua-kielen tulkin sulautuksen tarpeen ja vaatimukset.

5.1 MMORPG

Tutkittu palvelinohjelma on tarkoitettu massiivisille monen pelaajan verkkoroolipeleille (Massively multiplayer online role-playing game, MMORPG) [20]. Roolipeli on tarinankerronnan muoto, jossa pelaajat upottautuvat tarinan hahmoihin ja voivat vaikuttaa tarinaan. MMORPG-peleissä pelaajat yhdistävät tietokoneensa tietoverkon välityksellä yhteiseen palvelimeen, jossa suorittava palvelinohjelma ylläpitää pelin tilaa ja välittää sitä kaikille pelaajille. Pelaajat näkevät saman pelitilan, voivat nähdä toisensa ja toimia yhdessä virtuaalisessa maailmassa. Kuvassa 5.1 näkyy pelaajalle tyypillinen näkymä MMORPG-pelistä.

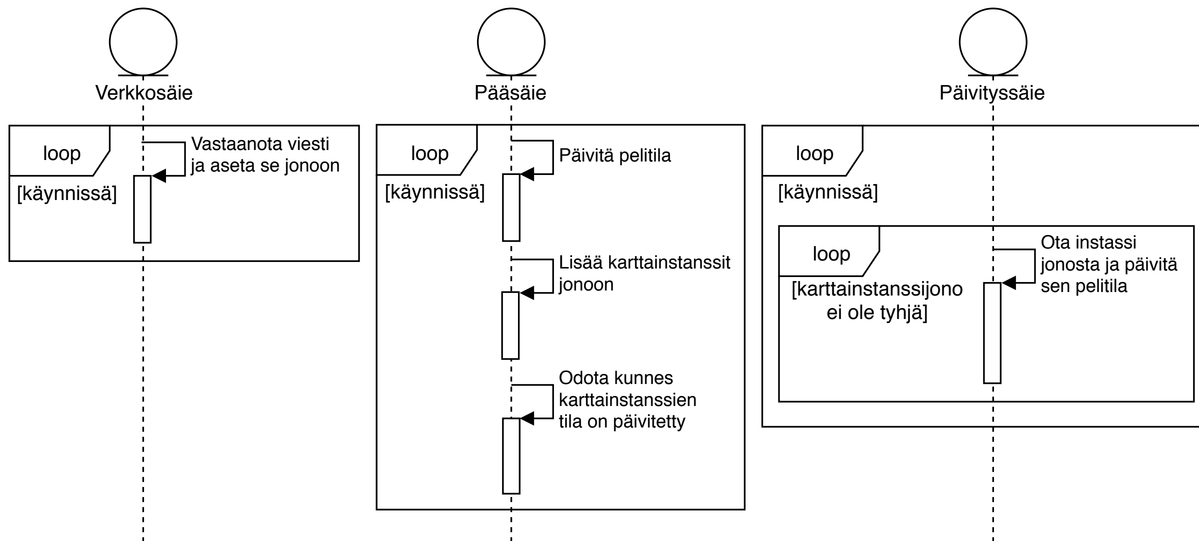
Tarkasteltu palvelinohjelma on suunnattu fantasiapeleille, joissa pelaajat suorittavat tehtäviä, tutkivat pelimaailmaa ja taistelevat vihollisia vastaan yhdessä tai yksin. Jotta pelaajilla riittää tekemistä, on tärkeää luoda jatkuvasti lisää uutta ja mielenkiintoista sisältöä peliin esimerkiksi uusien pelinsisäisten tapahtumien avulla, jotka vievät pelin tarinaa eteenpäin ja esittelevät uusia vihollisia peliin. Tehtävien, tapahtumien ja vihollisten monimuotoisuus ja vuorovaikutus ovat tärkeitä mielenkiinnon ylläpitämiseksi. Pelaajia kiinnostavat myös haasteet, jolloin monimutkaiset käyttäytymiset vihollisilla ovat tärkeitä, jotta pelaajan tulee tarkkailla vihollisen käyttäytymistä ja varautua siihen hankkimalla etukäteen esimerkiksi erityisiä taitoja tai esineitä ja kehittää koordinointia muiden pelaajien kanssa vihollisten voittamiseksi.

Peleissä on erilaisia hahmoja ja objekteja, joita yhdessä kutsutaan entiteeteiksi (entity). Kuvassa 5.1 voidaan nähdä numerolla 1 merkitty pelaajahahmo, jota pelaaja ohjaa. Kuvassa nähdään myös numerolla 2 merkitty hahmo, jota pelaajat eivät liikuta. Kyseisiä hah-

Kuva 5.1: Pelaajalle näytetty MMORPG-pelin näkymä, jossa on merkitty pelin eri elementtejä numeroilla. Numerolla 1 on merkitty pelaajahahmo, numerolla 2 NPC-hahmo, numerolla 3 lyhtynä toimiva objekti ja numerolla 4 NPC-hahmon pelaajalle näyttämä valikko.

moja kutsutaan ei-pelaaja-hahmoiksi (non-player character, NPC). NPC-hahmot antavat tehtäviä tai toimivat esimerkiksi vihollisina. Jotta NPC-hahmot voivat toimia, tarvitsevat ne erilaisia käyttäytymismalleja. Käyttäytymismallit määräävät hahmon käyttäytymisen eri tilanteissa ja ne voivat olla hyvinkin monimutkaisia toteuttaakseen esimerkiksi tunteja kestävän tapahtuman, jossa useat hahmot toimivat yhdessä ja reagoivat pelin tapahtumiin dynaamisesti. Pelaaja voi olla vuorovaikutuksessa myös erilaisten objektien kanssa. Esimerkiksi vipuobjektia kääntämällä voi avata oven tai käynnistää jonkin tapahtuman. Kuvassa voidaan nähdä numerolla 3 merkitty objekti, joka toimii lyhtynä. Pelaajan päätteen näytöllä näkyy pelimaailman lisäksi käyttöliittymä. Käyttöliittymässä voidaan näyttää pelaajalle erilaisia valikoita, joiden kautta pelaaja voi vuorovaikuttaa entiteettien ja muiden pelaajien kanssa. Kuvassa voidaankin nähdä numerolla 4 merkitty valikko, jonka avulla NPC-hahmo tarjoaa tehtävää pelaajalle.

Pelimaailma on yleensä jaettu karttoihin, jotka määrittävät alueen maaston. Kartoista voi olla olemassa kerrallaan useita instansseja. Instanssi voi sisältää useita entiteettejä, ja yksi entiteetti voi olla kerrallaan vain yhdessä instanssissa. Karttojen instanssit eivät ole toisistaan täysin eristettyjä, joten eri instansseissa olevat entiteetit voivat vaikuttaa toisiinsa esimerkiksi lähettämällä toisilleen viestejä. Pelaajat eivät kuitenkaan esimerkiksi näe entiteettejä toisista karttainstansseista, vaikka instanssi olisi samasta kartasta. Pelaa-



Kuva 5.2: Korkean tason kaavio palvelinohjelmien yleisestä toimintatavasta. Vasemmalla on asiakasohjelman viestejä vastaanottava säie, keskellä on ohjelman pääsäie ja oikealla päivityssäie. Verkkosäikeitä ja päivityssäikeitä voi olla useita. Viestejä käsitellään pelitilojen päivityksen yhteydessä.

jat voivat liikkua karttainstanssien välillä teleportaation avulla tai ennalta määrättyjen kulkupisteiden kautta, jotka käytännössä toimivat teleportaation tavoin.

5.2 Palvelinohjelman arkkitehtuuri

Tapaustutkimuksen kohdepalvelinohjelma, TrinityCore [62], käyttää asiakas-palvelinarkkitehtuurimallia (client-server model) [56], ja palvelinohjelma on toteutettu C++-kielellä. Kuvassa 5.2 nähdään kaavio palvelinohjelman keskeisistä säikeistä ja niiden suorituksesta. Kuvassa nähdään, miten ohjelma päivittää jatkuvasti pelitiloja, kunnes ohjelma pysäytetään muuttamalla muuttuja `käynnissä` epätodeksi. Kuvassa näkyviä pääsäikeitä on vain yksi, mutta päivityssäikeitä ja verkkosäikeitä voi olla useita samanaikaisesti suorittamassa, jotta palvelin voi skaalautua käyttäjämäärän kasvaessa. Karttainstansseja kuvaa `Kartta`-luokka ja pääsäie suorittaa toimintoja kutsumalla `Maailma`-luokan instanssin funktioita.

Kuvan 5.2 verkkosäikeet ottavat vastaan viestejä jatkuvasti asiakasohjelmilta ja asettavat niitä `Maailma`-luokan instanssin omistamaan jonoon. `Maailma`-luokka käyttää singletonmallia [54], joten siitä on aina vain yksi kaikkialta käytettävä instanssi. Viestejä käsitellään pelitilojen päivityksen yhteydessä. Pääsäie lisää välillä karttainstanssit jonoon ja odottaa, että päivityssäikeet ovat päivittäneet ne kaikki. Yksittäinen päivityssäie ottaa jatkuvasti

karttainstansseja jonosta, jota pääsäie täyttää, ja päivittää niiden pelitiloja yksi kerrallaan. Pääsäikeessä voidaan karttainstanssien pelitilojen päivittämisen ulkopuolella turvallisesti lukea ja kirjoittaa kaikkiin pelitiloihin ilman kriittisen osion suojausta, sillä sen suorituksen aikana pelitiloja muokkaa vain yksi säie. Tästä syystä pääsäikeen suorituksen aikana käsitellään karttainstanssien väliset vuorovaikutukset ja niiden jakaman pelitilan päivitykset, kuten pelaajien puhuminen, siirtyminen karttainstanssista toiseen ja pelaajien muodostamien ryhmien tilan päivitys.

Verkkosäikeiden vastaanottamilla viesteillä on operaatiokoodi (opcode), joka määrää viestin aiheuttaman toiminnan. Asiakasohjelman (client) lähettämät viestit käsitellään eri vaiheissa. Jotkin viestit käsitellään verkkosäikeessä, toiset ohjelman pääsilmukassa ja jotkin viestit käsitellään karttainstanssin pelitilan päivittämisen yhteydessä. Viestin käsittelyn sijainti riippuu viestin operaatiokoodista, eli sen aiheuttamasta toiminnasta. Toiminta saattaa esimerkiksi vaikuttaa muuttujiin, joita ei voida muokata karttainstanssien pelitiloja päivitettäessä rinnakkain, jolloin viesti tulee käsitellä ohjelman pääsilmukassa. Pääsäikeen ja karttainstanssin pelitilan päivittämisen aikana käsitellään ajastetut tapahtumat, viestijonon määräämät operaatiot ja näistä aiheutuneet tapahtumaketjut. Lisäksi karttainstanssin pelitilan päivityksessä käsitellään instanssiin liittyvien entiteettien jatkuvat toiminnat kuten entiteettien sijainnin päivitys.

Palvelinohjelma tarjoaa ohjelmointirajapinnan tapahtumille. Se koostuu luokista, joissa on funktioita eri tapahtumia varten. Luokista luodut oliot laitetaan ohjelmaa alustettaessa listoihin, joista ne löydetään myöhemmin. Tapahtuman sattuessa haetaan listasta tapahtumaan liittyvät oliot ja kutsutaan niiden tapahtumaan liittyviä tapahtumafunktioita. Tapahtumafunktiot ovat virtuaalisia, jolloin luokan perivä luokka voi ohittaa (override) funktioiden toiminnan uudella määritelmällä. Ohjelmointirajapinnan luokat määrittävät aina uniikin nimen, joka toimii luokan olioiden tunnisteena esimerkiksi niitä talletettaessa. Ohjelmointirajapinnan oliot vastaavat tapahtumista, jotka eivät ole sidottu hahmojen käyttäytymismalleihin.

Entiteettejä kuvaa `Entiteetti`-luokan olio, joka pitää sisällään sen käyttäytymismallin määräävän olion eli käytösolion. Entiteetillä voi olla kerrallaan vain yksi käytösolio. Entiteetti-luokka peritään `Objekti`-, `Pelaaja`- ja `NPC`-luokissa, joita käytetään eri entiteettityyppien yhteydessä. Yhdessä ohjelmointirajapinnan luokassa on tapahtumafunktio, jota kutsumalla luodaan uudelle hahmolle sen käytösolio. Ohjelmointirajapinnan olioiden tavoin käytösolio koostuu luokasta, jossa on ennalta määriteltyjä funktioita eri tapahtumia varten. Eri puolilla ohjelmaa, esimerkiksi hyökkäämistä käsittelevissä funktioissa,

kutsutaan hahmon käytösolion tapahtumille tarkoitettuja tapahtumafunktioita. Tapahtumafunktiot ovat virtuaalisia, joten niitä voidaan ohittaa käytösolion luokan perivissä luokissa. Perinnän avulla voidaan siis luoda erilaisia käytösolioita eri hahmoille. Perintä mahdollistaa myös erilaisten käytöspohjien luomisen, jotka voi sitten periä ja kehittää eteenpäin. Käytösolioiden perimässä luokassa on tapahtumafunktio, jota kutsutaan jokaisen pelitilapäivityksen yhteydessä ja jossa päästään käsiksi funktion parametriin, joka ilmaisee funktion edellisestä kutsusta kuluneen ajan. Käytösolion voi myös määritellä tallettavan tilatietoa entiteettikohtaisesti. Tilatiedon ja kuluneen ajan avulla käytösolio voi ajastaa tapahtumia laskemalla kulunutta aikaa.

Ohjelmointirajapinnan olioiden tapahtumafunktiot toimivat kaikki samalla tavoin. Tapahtumafunktio saa parametriensa kautta tapahtumaan liittyvää tietoa, kuten esimerkiksi hyökkääjän, hyökkäyksen kohteen ja kohteen menettämien vointia kuvaavien pisteiden (hit points, HP) määrän. Tapahtumafunktion parametrit voivat olla tyypiltään osoittimia tai viitteitä, jolloin niiden kautta voidaan muokata esimerkiksi kohteen menettämien pisteiden määrää. Tapahtumafunktiot voivat myös palauttaa arvoja, jotka esimerkiksi ilmaisevat tulisiko tapahtumaketju keskeyttää, jolloin kohde ei menettäisikään pisteitä.

Palvelinohjelma käyttää tietokantaa tiedon pysyvään tallentamiseen. Jokainen NPC-hahmo ja objekti on kuvattu tietokannassa ilmaisten niiden koon, ulkonäön ja muut attribuutit. Ohjelmointirajapinnan olioiden tunnisteiden avulla olio voidaan liittää NPC-hahmoihin ja objekteihin tietokannassa. Tietokannan avulla voidaan myös ohjelmoida hahmoille reaktioita eri tapahtumiin. Reaktiot talletetaan tietokannassa olevan taulun avulla, jossa on erilaisia kenttiä. Kentät ilmaisevat mitä tehdään ja milloin. Tietokantarivi sisältää kentät tapahtuman tunnisteelle, tehdyn toiminnan tunnisteelle ja toiminnan erityyppisille parametreille. Tietokannassa määritellyt reaktiot suoritetaan käyttäytymismallijärjestelmän päälle rakennetun järjestelmän kautta. Yksinkertaistettuna järjestelmän käyttämän käytösolion tapahtumafunktio etsii tapahtumaan liittyvät tietokannassa kuvatut reaktiot ja suorittaa ne.

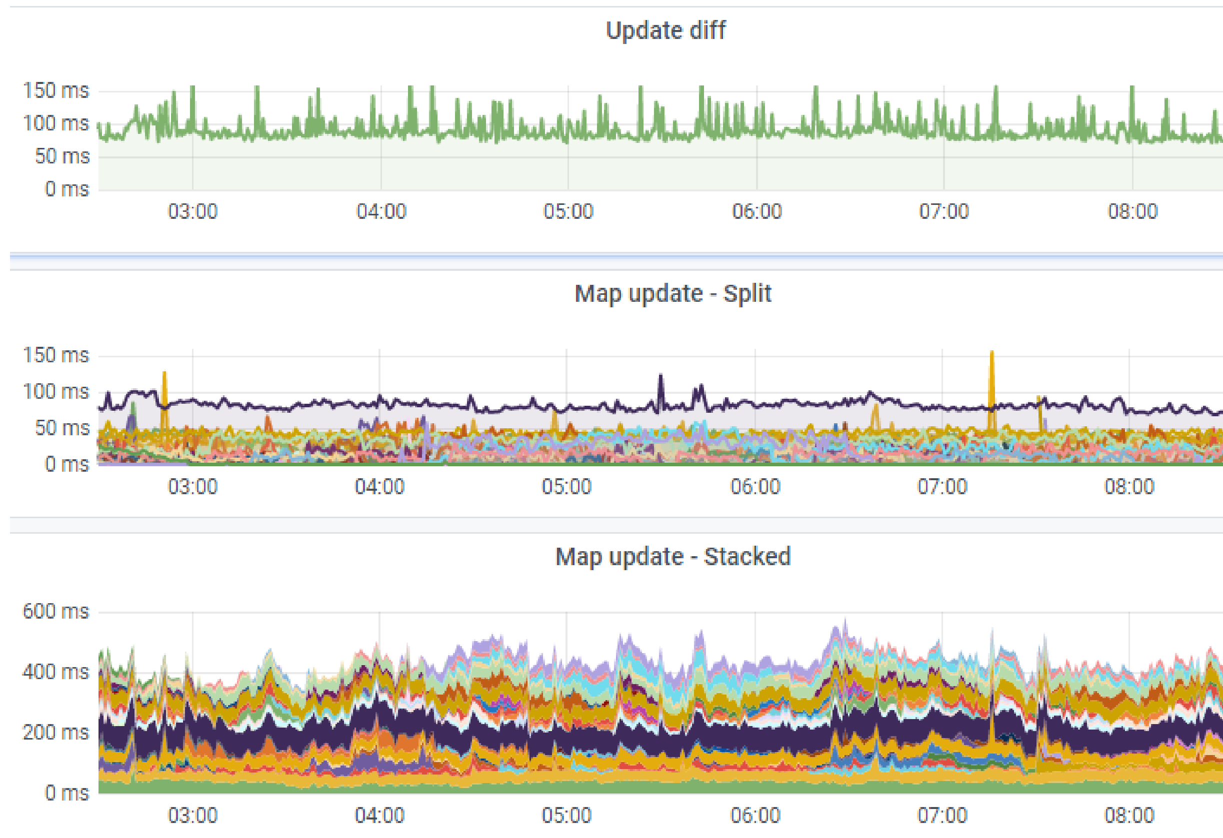
5.3 Palvelinohjelman päivityssykli

Osana peliä pelaajat voivat taistella toisiaan vastaan tai toistensa kanssa. Hahmoilla on erilaisia taitoja, joita he voivat käyttää. Taitoihin liittyy erilaisia ajastimia, jotka määräävät esimerkiksi, kuinka pitkään taito kestää ja kuinka usein sitä voi käyttää. Taidot usein vaativat, että taidon kohde on pelaajahahmon edessä. Palvelinohjelman tehtävänä

Kuva 5.3: Tilanne, jossa palvelinohjelma ei ole vielä ehtinyt lähettää yhden pelaajan tekemiä toimintoja toiselle pelaajalle, jolloin yhden pelaajan hyppy ei vielä näy toisen pelaajan ruudulla.

on päivittää pelin tila ja lähettää se kaikille pelaajille. Jotta pelaajat näkevät toistensa liikkeet, voivat kohdistaa taitonsa oikein, ja voivat ajoittaa taitojensa käytön pelin määräämien rajoitusten sisällä tehokkaasti, on pelin tila pystyttävä päivittämään tarpeeksi usein. Palvelinohjelman päivitysajan ylittäessä asetetut aikamääreet kestävät taidot pelaajilla tarkoitettua pitempään ja päivitysajan kasvaessa pelaajien havaitsema pelin responsiivisuus heikkenee. Kuvassa 5.3 nähdään, miten oikealla puolella hahmo on omalla ruudullaan jo hypännyt, mutta vasemmalla puolella nähtävässä kuvassa toisen pelaajan ruudussa hahmo ei ole vielä liikkunut ollenkaan. Kuva on otettu palvelinohjelman päivityssyklien välisen ajan ollessa 200 millisekuntia. Empiirisesti voidaan havaita yli sadan millisekunnin päivitysten välisen ajan vaikuttavan kahden pelaajan väliseen kamppailuun negatiivisesti, sillä pelaajat eivät näe toistensa liikkeitä tarpeeksi nopeasti. Tässä työssä pidämme 100 millisekuntia rajana tarpeeksi hyvälle päivitysten väliselle ajalle.

Päivitysten välinen aika määrittyy päivityssyklien keston perusteella. Kuvassa 5.4 nähdään tuotantoympäristössä suoritettavasta palvelinohjelmasta kerättyä julkisesti jaettua metriikkaa [61]. Metriikan tuottanut palvelinohjelma on konfiguroitu käyttämään kahdeksaa prosessoriydintä. Kuvassa on kolme eri kuvaajaa, joiden x-akselilla on mittauksen ajan-



Kuva 5.4: Kolme kuvaajaa palvelinohjelman tuotantokäytöstä kerätystä metriikasta, joista ylimmässä näkyy pelin päivityssykliin kuluva kokonaisaika, keskimmäisessä näkyy eri karttojen päivitykseen kuluva aika ja alimmassa näkyy karttojen päivitysaikojen summa.

hetki ja y-akselilla on päivittämiseen kulunut mitattu aika millisekunteina. Ensimmäinen kuvaaja näyttää koko pelitilan päivityssyklin kokonaiskeston, toinen kuvaaja näyttää yksittäisten pelikarttojen päivityksen keston samassa kuvaajassa erillisinä viivoina ja kolmas kuvaaja näyttää pelikarttojen päivityksen keston kuvaajana, jossa päivitysajat on summattu toisiinsa. Karttoja on kuvassa 80 ja ne on merkitty eri väreillä kahdessa alemmassa kuvaajassa. Yksi päivityssykli koostuu yksisäikeisestä suorituksesta ja monisäikeisestä karttojen päivityksestä. Jotta peli tuntuu sulavalta, halutaan maksimoida päivitysnopeus, eli minimoida päivitykseen kuluva aika.

Kolmannen kuvaajan perusteella nähdään, että yksisäikeisessä suorituksessa palvelimen kokonaispäivitysaika olisi keskimäärin yli 400 millisekuntia. Monisäikeisyyden avulla on päivitysaika saatu siis pidettyä alle neljänneksenä sarjalliseen suoritukseen kuluvasta ajasta. Kuvasta nähdään ensimmäisen ja toisen kuvaajan avulla, että yksi kartta varaa yhden säikeen lähes koko päivityssyklin keston ajaksi. Päivityssyklin kesto on keskimäärin noin 70 millisekuntia, eli pelitila päivitetään noin 14.29 kertaa sekunnissa. Jos jokaisen kartan suoritukseen lisätään 0.5 millisekunnin sarjallinen suoritus, voidaan laskea kokonaispäivitysajan nousevan 40 millisekuntia. Tämä nostaisi päivityssyklin kokonaiskeston noin 120 millisekuntiin, eli päivitys tapahtuisi noin 9.09 kertaa sekunnissa. Jos lisätty työmäärä jaetaan kahdeksalle säikeelle, kasvaa kokonaiskesto arviolta vain 5 millisekuntia, eli päivitys tapahtuisi noin 13.33 kertaa sekunnissa.

Vaikka laskelmat eivät ole tarkkoja, voidaan niiden avulla arvioida päivitystiheyden huomattava aleneminen pienenkin sarjallisen suorituksen takia verrattuna rinnakkaiseen suoritukseen. Tästä voidaan johtaa, että sarjallisen suorituksen minimointi on erityisen tärkeää ja hyödyllistä päivityssyklien tiheyden kannalta. Voidaan myös todeta, ettei kuvan tilanteessa sulautetun kielen toteutus saa johtaa 30 millisekuntia pidempään kartan päivitykseen, jotta päivityssyklit voivat kestää keskimäärin alle 100 millisekuntia. Julkisen metriikan mukaan entiteettejä on useimmilla kartoilla lähes 400 ja muutamilla suuremmilla kartoilla entiteettejä voi olla jopa 20000.

5.4 Vieraskielen sulauttamisen vaatimukset

Tutkimuksen kohteena on Lua-ohjelmointikielen lisääminen palvelinohjelmaan. Lua-kielellä on tarkoitus voida ohjelmoida reaktioita pelin kartoilla sattuviin tapahtumiin, jotka käsitellään palvelinohjelmassa. Lua-kielen minimalistisuus auttaa uusia sisällöntuottajia ja ohjelmoijia vähentämällä heidän tarvitsemansa tiedon ja taidon määrää

verrattuna C++-kieleen. Syitä Lua-ohjelmointikielen valinnalle on yhteisöjen ja kehittäjien aiempi osaaminen kyseisen kielen kanssa ja kielen tarkoitusta tukevat piirteet, kuten minimalistisuus ja nopeus. Lisäksi palvelimen tukeman pelin käyttöliittymä on ohjelmoitu osin Lua-ohjelmointikielellä, jolloin käyttöliittymän ja palvelinohjelman osia voisi toteuttaa samalla kielellä helpottaen kehittämistä. Lua-ohjelmointikieli on tulkattu, jolloin se eliminoi kääntämiseen kuluvan ajan ja siten mahdollistaa nopean muutoksien kokeilemisen. Kielen tulkkaus mahdollistaa myös suorituksenaikaisen ohjelmoinnin toteutuksen, jota käytettäessä ohjelmaa ei tarvitse käynnistää uudelleen ohjelmakoodia lisätessä, poistaessa tai muokatessa. Lua-kieli on tulevaisuudessa tarkoitus lisätä useampaan samanlaiseen palvelinohjelmaan.

Lua-kielen lisäämisen toteutuksella on 5 vaatimusta.

1. Lua-koodia tulee voida suorittaa yhtäaikaisesti, jotta palvelimen päivitystiheys ei kärsi.
2. Sulautuksen toteutus ei saa riippua palvelinohjelman toteutuksesta.
3. Lua-ohjelmia tulee voida lisätä, poistaa ja muokata ilman palvelinohjelman uudelleenkäynnistystä.
4. Lua-ohjelmassa tapahtuva virhe ei saa voida keskeyttää palvelinohjelman suorittamista.
5. Päivityssyklin suoritusaika ei saa nousta yli 30 millisekuntia toteutuksen takia.

Vaatimukset tukevat lisätyn ohjelmointikielen käyttötarkoituksia. Palvelinohjelma on monisäikeinen ja vaatimuksen 1 avulla varmistetaan, ettei Lua-kielen suorittaminen johda yksisäikeiseen suoritukseen palvelimen arkkitehtuurin vastaisesti. Vaatimuksen 2 tuloksena kieli voidaan lisätä saman toteutuksen avulla useaan erilaiseen ohjelmaan, sitä voidaan tarkastella itsenäisesti, ja muutokset ohjelmissa eivät aiheuta muutoksia sulautuksen toteutukseen. Vaatimus 3 varmistaa, että tulkatun kielen hyötyihin päästään käsiksi myös suorituksen aikana. Sen avulla voidaan kehityksen aikana välttää palvelinohjelman uudelleenkäynnistys, joka voi viedä jopa kymmenen minuuttia. Vaatimus 4 varmistaa, että ohjelmoijat saavat virheilmoituksia vikatilojen sijaan, ja ettei heidän tarvitse huolehtia kielelle epätavallisista piirteistä, kuten vapautetun muistin käsittelystä. Vaatimus 5 asettaa aikavaatimuksen, joka määritettiin edellä oikean palvelimen metriikan perusteella. Vaatimuksen avulla varmistetaan, että toteutusta voidaan käyttää todellisessa tilanteessa.

6 Toteutus

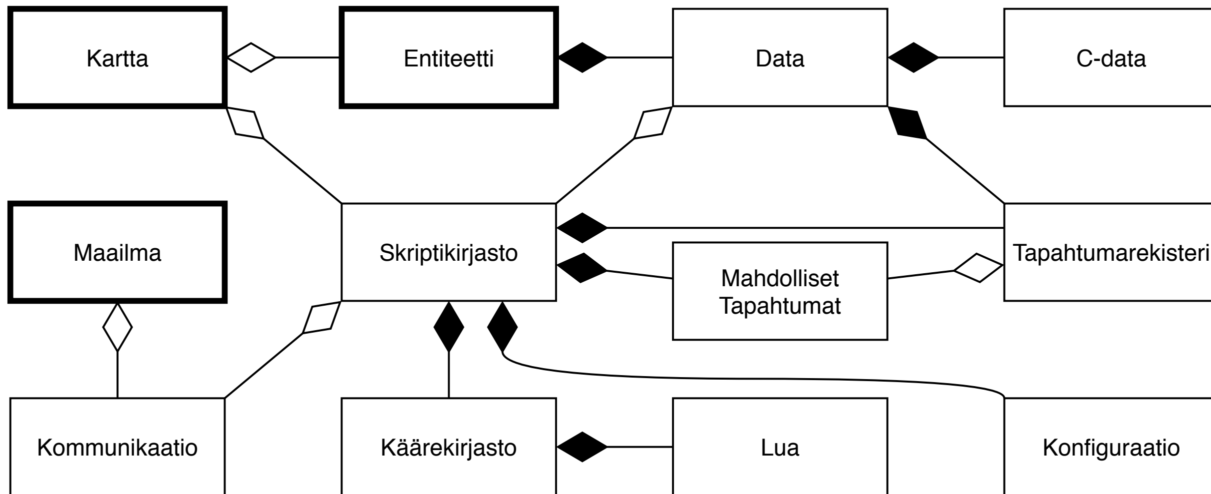
Tässä luvussa kuvataan Lua-kielen tulkin sulautuksen toteutuksen yksityiskohdat. Esittelemme toteutuksen korkean tason arkkitehtuurin, tapahtumien käsittelyn toteutuksen, C++-ohjelman ja Lua-kielen tulkin rajapinnan toteutuksen kuvauksen, Lua-kielen tulkkien välisen kommunikoinnin, ja virheidenkäsittelyn.

6.1 Arkkitehtuuri

Karttojen pelitiloja päivittäessä sattuu tapahtumia, joista halutaan suorittaa tapahtumiin liitettyä Lua-koodia. Karttojen pelitilojen päivittäminen tapahtuu rinnakkaisesti. Lua-tulkin toteutuksen takia vain yksi säie voi suorittaa kerrallaan Lua-ohjelmaa yhdessä Lua-tulkin instanssissa eli Lua-instanssissa. Jotta usean säikeen ja prosessorin mahdollistaman monisäikeisyyden tuomat edut säilytetään karttainstanssien päivityksessä, niin käytetään useaa Lua-instanssia, jolloin usea säie pystyy suorittamaan Lua-ohjelmaa rinnakkain. Mittauksia varten käytämme samaa toteutusta, mutta muokkaamme sen käyttämään yhtä lukolla suojattua Lua-instanssia.

Kuvassa 6.1 nähdään toteutuksen keskeiset luokat ja niiden yhteydet. Kuvasta näkyy, että Lua-instansseja käytetään itse asiassa käärekirjaston (wrapper library) avulla, joka helpottaa Lua-kielen tarjoaman C-rajapinnan käyttöä. Käärekirjastoksi on tässä toteutuksessa valittu Sol2-käärekirjasto, joka tukee Lua-kieltä ja käyttää C++-kielen uusimpien versioiden toimintoja hyödyksi [38]. Kuvassa näkyy myös, miten toteuttamamme `Skriptikirjasto`-luokka, eli skriptikirjasto, omistaa käärekirjaston instanssin. Kyseinen luokka omistaa lisäksi konfiguraation, mahdollisten tapahtumien rekisterin, ja tapahtumarekisterin. Se myös viittaa `Kommunikaatio`-luokan instanssiin. Kaikki viitteet skriptikirjastoon ovat aggregaatteja, eli siihen vain viitataan eikä sitä omisteta. Tämä johtuu siitä, että skriptikirjastoon viitataan ainoastaan jaetuilla älykkäillä osoittimilla, jotka varmistavat, ettei oliota tuhota, kun siihen vielä viitataan.

Kuvassa 6.1 näkyvä `Konfiguraatio`-luokka toteuttaa tavan ladata konfiguraation levyltä ja sisältää esimerkiksi lokitusasetukset ja Lua-ohjelmia sisältävän kansion sijainnin. Kuvassa näkyvä `Mahdolliset tapahtumat`-luokka toteuttaa listan tapahtumista, joita kyseisessä skriptikirjaston instanssissa voi tapahtua. Sen avulla voidaan validoida, onko kysei-



Kuva 6.1: Toteutuksen eri luokkien yhteydet toisiinsa. Palvelinohjelman omat luokat on lihavoitu.

nen tapahtuma mahdollinen sen sattuessa tai liittäessä tapahtumaan jokin Lua-funktio. `Tapahtumarekisteri`-luokka tallettaa tapahtumiin liitetyt Lua-funktiot ja käyttää validointiin apunaan skriptikirjaston instanssin omistamaa mahdollisten tapahtumien listaa. `Kommunikaatio`-luokan instanssi on jaettu kaikkien skriptikirjastoinstanssien välillä. Sen avulla Lua-instanssien välillä voidaan kommunikoida. Se varastoi kommunikointiin käytetyt viestilistat.

Palvelinohjelman päivityssäikeet päivittävät satunnaisten karttainstanssien pelitiloja päivityskerrasta toiseen. Jos jokaiselle säikeelle annetaan oma Lua-instanssi, niin tällöin samalla kartalla sattuneen tapahtumaan liitetyn Lua-ohjelman asettama muuttuja ei Lua-instanssin vaihdon takia olisi seuraavan tapahtuman sattuessa enää välttämättä asetettu. Päivityssäikeet päivittävät kerrallaan yhden karttainstanssin pelitilaa. Annettaessa jokaiselle karttainstanssille oma Lua-instanssi, joka käsittelee kyseisen kartan tapahtumiin liitetyt Lua-ohjelmat, säilyvät Lua-ohjelmissa asetetut muuttujat päivityssyklien välillä kartalla sattuvien tapahtumien välillä. Kuvassa 6.1 nähdään, että kartta viittaa skriptikirjaston instanssiin, jonka kautta se voi käyttää skriptikirjaston funktioita tapahtumien käsittelyyn.

Kuvassa 6.1 näkyvät karttaan liitetyt entiteetit omistavat `Data`-luokan instanssin, joka sisältää viitteen kartan skriptikirjastoinstanssiin. `Data`-luokka omistaa oman tapahtumarekisterinsä, jonne entiteetin tapahtumiin liitetyt Lua-funktiot on talletettu. Tapahtumarekisterin ja skriptikirjastoinstanssin avulla entiteetin avulla voidaan käsitellä entiteettiin liittyviä tapahtumia. `Data`-luokka sisältää myös tietorakenteen, johon Lua-ohjelma voi tallettaa entiteettiin liittyvää tietoa. Kyseinen tieto on sarjallistettu ja talletettu erilleen

Lua-instansseista, mutta siihen päästään käsiksi Lua-instansseista sen omistavan entiteetin kautta. Sen avulla voidaan jakaa entiteettiin liittyvää tietoa eri Lua-instanssien välillä, sillä se ei ole sidottu mihinkään Lua-instanssiin. Lisäksi talletettu tieto ei katoa, vaikka Lua-instanssit tuhottaisiin ja luotaisiin uudelleen.

Ohjelman käynnistyessä pääsäie alustaa ohjelman tilan. Samalla se luo `Kommunikaatio`-luokan instanssin ja tallettaa sen `Maailma`-luokan instanssiin, josta siihen voidaan viitata mistä vain. Karttainstanssin käyttämä skriptikirjaston instanssi luodaan karttainstanssin luonnin yhteydessä. Skriptikirjaston instanssia luodessa talletetaan siihen viite `Kommunikaatio`-luokan instanssiin. Skriptikirjaston instanssin luonnin jälkeen skriptikirjasto lataa konfiguraation levyltä. Sitten suoritetaan kaikki konfiguraation määräämästä kansiosta löytyvät tiedostot Lua-ohjelmina. Entiteettien omistaman `Data`-luokan instanssi luodaan entiteettiä luodessa ja sen omistama skriptikirjaston viite asetetaan aina entiteetti asetettaessa jollekin kartalle käyttämällä kyseisen kartan omistaman skriptikirjaston viitettä. Skriptikirjaston viitettä asetettaessa myös tapahtumarekisteri luodaan uudelleen.

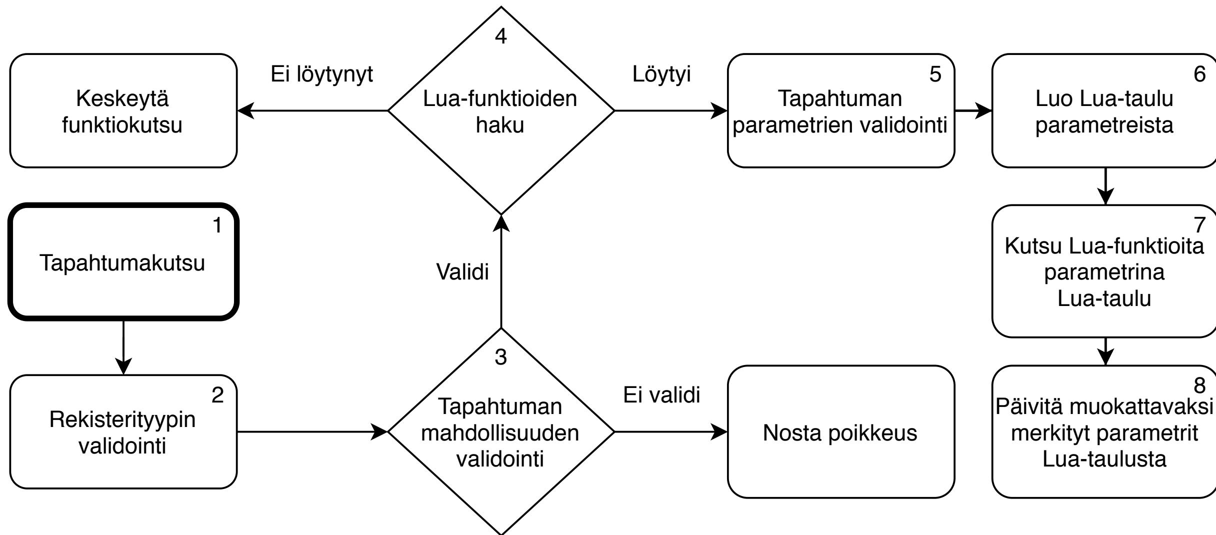
6.2 Tapahtumien käsittely

Jotta Lua-koodia voidaan suorittaa tapahtuman sattuessa, sijoitetaan palvelimen ohjelmakoodin sekaan funktiokutsuja, jotka suorittavat tapahtumaan liitetyt Lua-funktiot. Tapahtumat suoritetaan yhteisen tapahtumankäsittelysysteemin kautta, joka kuvataan seuraavaksi.

6.2.1 Tapahtumakutsun kulku

Tapahtumakutsuun liittyvät tapahtumafunktiot, tapahtumaoliot, tapahtumarekisterit, ja mahdollisten tapahtumien joukko. Kuvassa 6.2 nähdään tapahtumakutsun eri suorituspolut ja vaiheet. Kuvassa merkityssä ensimmäisessä vaiheessa kutsutaan tapahtuman sattuessa tapahtumafunktiota. Tapahtumafunktio ottaa vastaan ensimmäisenä parametrinaan tapahtumarekisterin, joka sisältää tapahtumiin liitetyt suoritettavat Lua-funktiot. Toinen parametri on tapahtumaolio, joka sisältää tapahtumarekisterin tyypin, tapahtuman nimen, ja tapahtuman parametrien nimet ja tyypit. Lopuksi funktio saa vaihtelevan määrän parametreja, jotka välitetään tapahtumaan liitetyille Lua-funktioille.

Kuvassa 6.2 merkityssä toisessa vaiheessa validoidaan rekisterin tyyppi. Tapahtumarekisterille on määrätty tyyppi, jonka tapahtumia se käsittelee. Tapahtumakutsun yhteydessä



Kuva 6.2: Tapahtumakutsun kulku tilakaaviona. Alkutila on lihavoitu ja osa tiloista on merkitty numeroin.

tapahtumaolion sisältämän tapahtumarekisterin tyypin avulla voidaan tarkistaa tapahtuman ja rekisterin yhteensopivuus. Tapahtumafunktio on skriptikirjaston funktio, jonka sisällä päästään käsiksi skriptikirjaston eri osiin. Skriptikirjaston instanssia luodessa kaikki instanssin mahdolliset tapahtumat talletetaan instanssiin, jolloin niiden avulla voidaan validoida voiko tapahtuma sattua kyseisessä instanssissa. Kuvassa näkyvässä kolmannessa vaiheessa tapahtuman mahdollisuus validoidaan. Jos tapahtuma ei ole mahdollinen, niin nostetaan poikkeus. Jos tapahtuma on mahdollinen, niin jatketaan Lua-funktioiden haulla rekisteristä. Neljännessä vaiheessa tapahtumaolion sisältämän nimen avulla voidaan tapahtumarekisteristä hakea tapahtumaan liitetyt Lua-funktiot. Jos funktioita ei ole, niin keskeytetään funktiokutsu. Muutoin suoritus jatkuu parametrien validoinnilla.

Tapahtumaoliot tietävät tapahtumafunktion parametrien tyypit ja määrän, joten niiden avulla kuvan 6.2 kohdassa viisi voidaan tapahtumakutsun parametrit validoida. Tapahtumaoliot on tarkoitettu luotavaksi ohjelmassa vain kerran, jolloin niihin voidaan viitata suoraan muuttujanimellä, eikä parametrien nimiä tai tyyppejä tarvitse määritellä useita kertoja ohjelmassa. Parametrien validoinnin jälkeen kohdassa kuusi luodaan Lua-taulu, johon talletetaan tapahtumaoliosta saatavien parametrien nimet avaimiksi ja tapahtumafunktiolle annetut tapahtuman parametrit arvoiksi. Kohdassa seitsemän kutsutaan kohdassa neljä haettuja Lua-funktioita yksi kerrallaan parametrinaan kohdassa kuusi luotu Lua-taulu. Kaikkien tapahtumiin liitettyjen Lua-funktioiden parametrina on siis yksittäinen Lua-taulu, joka sisältää tapahtumaolion määräämät nimet avaimina ja tapahtuman parametrit arvoina. Lua-funktio voi muutella Lua-taulua haluamallaan tavalla. Lopuksi kuvan

kohdassa kahdeksan parametrina käytetystä Lua-taulusta haetaan tapahtumaoliossa muokattaviksi merkityt parametrit ja asetetaan niiden arvot tapahtumafunktion parametrien arvoiksi. Muokattavat parametrit ovat käytännössä viitteitä funktiokutsun ulkopuolella oleviin muuttujiin, jolloin niiden muokkaaminen muokkaa suoraan kyseisiä muuttujia.

Eri tapahtumilla on erilaisia parametreja ja tapahtumarekistereiden tyypit ovat skriptikirjaston käyttäjän, eli sulauttajan, valittavissa. Jotta jokaiselle tapahtumalle ei tarvitsisi määritellä omaa tapahtumafunktiota erityyppisten luokkien käsittelyyn, niin käytetään mallipohjaista ohjelmointia. Sen avulla kääntäjä luo tarvitsemansa ohjelmakoodin funktion kutsujen ja abstraktin määritelmän perusteella. Siten tapahtumafunktiota voidaan kutsua eri määrillä erityyppisiä parametreja. Myös tapahtumaoliot käyttävät mallipohjaista ohjelmointia, jonka avulla määritellään tapahtumaolion tapahtumarekisterin tyyppi ja parametrien tyypit. Mallipohjainen ohjelmointi mahdollistaa myös erilaiset tyyppitarkistukset käännöksen aikana. Kuvassa 6.2 kohtien 2 ja 5 validoinnit tehdään mallipohjaisen ohjelmoinnin avulla jo ohjelman käännöksen aikana, jolloin ohjelman suorituksen aikana ei tarvitse tehdä mitään. Lisäksi kuvan kohdan 3 validointi voidaan tarvittaessa jättää kokonaan pois ohjelman julkaisuversiosta. Tällöin erilaisiin validointeihin ei käytetä ollenkaan arvokasta suoritusaikaa.

```
1 const LuaEvent<
2   Global, int, Mutable<std::string>
3 > OnSendText(
4   "OnSendText", { "integer", "text" }
5 );
6 void SendText(Registry<Global>& reg, std::string text) {
7   sWorld->lua->DoEventFrom(reg, OnSendText, 100, std::ref(text));
8   Send(text);
9 }
```

Listaus 6.1: Riveillä 1-5 nähdään esimerkki tapahtumaoliosta ja riveillä 6-9 nähdään esimerkki sen käytöstä tapahtumakutsun yhteydessä.

Listauksessa 6.1 näkyy esimerkki `OnSendText`-tapahtumaolion käytöstä tapahtumakutsussa. Listauksessa riveillä 2 ja 6 nähdään, että tapahtumaolio ja tapahtumarekisteri käyttävät `Global`-luokkaa tapahtumarekisterin tyyppinä. Listauksen rivillä 2 nähdään tapahtumarekisterin tyypin jälkeen tapahtuman parametrien tyypit. Rivillä 4 nähdään tapahtuman nimi ja tapahtuman parametrien nimet, joita käytetään Lua-kielen kontekstissa rekisteröitäessä funktio tapahtumaan ja käsitellessä tapahtuman parametreja Lua-taulun avulla, jossa parametrien nimet ovat parametrien arvojen avaimina. Tapahtuman toinen parametri `Mutable<std::string>` on merkkijono, joka on merkitty muo-

kattavaksi. Tapahtumaan sidottujen Lua-funktioiden parametrina käytetyn Lua-taulun arvoja voi muokata Lua-funktiokutsun aikana ja muuttuneen muokattavaksi merkityt arvot päivitetään C++-ohjelman ympäristöön tapahtumakutsun jälkeen. Listauksen rivillä 6 nähdään muutettavan merkkijonon muuttuja `text` ja rivillä 7 nähdään tapahtumafunktion kutsu `OnSendText`-tapahtumaolion avulla. Muuttuja `text` syötetään tapahtumafunktiolle apufunktion kautta, joka käärii sen luokkaan. Kääreen avulla se tunnistetaan muokattavaksi parametriksi tapahtumakutsussa. Tapahtumakutsun jälkeen muuttuja `text` voi olla Lua-ohjelmassa määritetty merkkijono, jolloin Lua-koodista pystyy vaikuttamaan rivillä 8 käytettyyn tekstiin.

6.2.2 Eri tapahtumatyyppien käsittely

Tapahtumia on kolmenlaisia: (i) olioihin liittyviä tapahtumia, (ii) tapahtumia, jotka eivät ole sidottu mihinkään olioon, ja (iii) tunnisteeseen sidottuja tapahtumia. Tapahtumatyypit käyttävät edellä esiteltyä tapahtumankäsittelysysteemiä. Systeemiä on kuitenkin kankea käyttää sen tarvitseman tapahtumarekisterin takia, joten tapahtumatyypeille on erilaisia apufunktioita ja luokkia.

```
1  const LuaEvent<Object, Object> OnEvent("OnEvent", "object");
2  void Object::MemberFunction() {
3    data->DoEvent(OnEvent, this);
4  }
5  const GlobalLuaEvent OnGlobalEvent("OnGlobalEvent");
6  void GlobalFunction() {
7    sWorld->lua->DoEvent(OnGlobalEvent);
8  }
9  const IDLuaEvent<Object, Object> OnAIInit("OnAIInit", "object");
10 void Object::SomeEvent() {
11   std::string id = this->GetScriptId();
12   sWorld->lua->DoEventId(id, OnAIInit, this);
13 }
```

Listaus 6.2: Esimerkkejä erilaisten tapahtumatyyppien tapahtumakutsuista. Riveillä 1-4 on i-tyypin, riveillä 5-8 on ii-tyypin ja riveillä 9-13 on iii-tyypin tapahtumakutsun esimerkki.

Tyypin i tapahtumien, eli olioon sidottujen tapahtumien, tapahtumarekisteri sijaitsee olioiden omistaman aliluvussa 6.1 esitellyn `Data`-luokan instanssissa. Oliot ovat siis esimerkiksi entiteettejä. `Data`-luokalla on `DoEvent`-funktio, joka tekee tapahtumafunktiokutsun `Data`-luokan instanssin omistaman tapahtumarekisterin avulla. Listauksessa 6.2 nähdään esimerkki olioon sidotun tapahtuman tapahtumakutsusta riveillä 1-4. Konventiona tapahtu-

marekisterin tyyppinä käytetään rekisterin omistavan olion tyyppiä. Rivillä 1 on tapahtumaolion määritys, joka käyttää `Object`-luokkaa tapahtumarekisterin tyyppinä. Myös `Object`-luokan instanssin sisältämän `Data`-luokan instanssin tapahtumarekisterin tyyppi on `Object`. Kyseinen tapahtumatyyppi on tarkoitettu tapahtumille, jotka tapahtuvat jollekin oliolle. Käytännössä jokaisessa tapahtumakutsussa on mukana kyseinen olio parametrina. Olio omistaa tapahtumarekisterin, joten tapahtumat ovat oliokohtaisia ja rekisteriin voi lisätä tapahtumia vain pääsemällä käsiksi olioon.

Tyypin ii tapahtumien, eli sitomattomien tapahtumien, tapahtumarekisteri sijaitsee skriptikirjaston instanssissa. Skriptikirjastolla on siksi oma `DoEvent`-funktionsa, joka käyttää skriptikirjaston tapahtumarekisteriä. Listauksessa 6.2 nähdään esimerkki tapahtumakutsusta riveillä 5-8. Rivillä 5 on tapahtumaolion määritys, joka käyttää `GlobalLuaEvent`-apumääritystä tapahtumaolion tyyppinä. Se määrittelee tapahtumarekisterin tyypiksi `Global`, jota käytetään kaikille sitomattomille tapahtumille. Esimerkkitapauksessa tapahtumalle ei ole määritelty yhtään parametria. Kyseinen tapahtumatyyppi on tarkoitettu yleisille tapahtumille. Tapahtumarekisterin omistaa skriptikirjaston instanssi, joten rekisteriin voi lisätä tapahtumia ilman pääsyä mihinkään tiettyyn olioon.

Tyypin iii tapahtumien, eli tunnisteeseen sidottujen tapahtumien, tapahtumarekisterit sijaitsevat skriptikirjaston instanssissa. Skriptikirjastolla assosiaatiotaulu, jonka avaimena toimii tekstimuotoisen tunnisteen ja olion tyypin yhdistelmä. Arvona assosiaatiotaulussa on tapahtumarekistereitä. Listauksessa 6.2 nähdään esimerkki tapahtumakutsusta riveillä 9-13. Rivillä 11 ja 12 nähdään, miten skriptikirjastolla on `DoEventId`-funktio, joka hakee assosiaatiotaulusta funktiolle annetun tunnisteen ja tapahtumaolion tallettaman tyypin avulla tapahtumakutsuun käytettävän tapahtumarekisterin. Listauksen rivillä 9 on tapahtumaolion määritys, joka käyttää `IDLuaEvent`-luokkaa tapahtumaolion tyyppinä. Kyseinen luokka määrittelee tapahtumarekisterin tyypiksi `GlobalID`. Luokalle annettu tapahtumarekisterin tyyppi `Object` talletetaan tapahtumarekisterin hakua varten. Kyseinen tapahtumatyyppi on tarkoitettu yhdistämään i- ja ii-tyypin tapahtumat. Tapahtumatyypin iii avulla voidaan määrittää ii-tyypin tapahtumia, mutta ne voidaan jakaa ne eri kategorioihin luokkien avulla ja ne voidaan rajata lisäksi tiettyyn tunnisteeseen, joka voisi olla esimerkiksi eri olioilla uniikki.

6.2.3 Lua-funktion liittäminen tapahtumaan

Tapahtumiin tulee voida liittää Lua-funktioita, joita sitten suoritetaan tapahtuman sattuessa. Lua-funktiot liitetään tapahtumiin tallettamalla ne tapahtumarekistereihin, jotka sijaitsevat eri paikoissa riippuen tapahtumatyypistä. Tässä toteutuksessa pääsy rekistereihin on toteutettu C++-funktioiden kautta, jotka ovat paljastettu Lua-kielen ympäristöön. Lua-ohjelmia suoritetaan Lua-instanssin luonnin yhteydessä, jolloin ohjelmakoodissa voidaan kutsua paljastettuja funktioita. Paljastettujen funktioiden avulla voidaan Lua-funktioita liittää tapahtumiin niihin liitettyjen nimien kautta. Tapahtumiin liitettyjä funktioita voidaan myös poistaa tapahtumarekistereistä kutsumalla tapahtumien rekisteröintiin käytettävien funktioiden *Unregister*-loppuisia vastineita.

Listauksessa 6.3 nähdään Lua-koodia, jossa on esimerkkejä Lua-funktioiden sitomisesta tapahtumiin. Listauksen rivillä 6 kutsutaan kaikkialta saatavissa olevaa funktiota `Register`, jolla voidaan liittää funktioita ii-tyypin tapahtumiin. Funktiolle tulee antaa tapahtuman nimi ja Lua-funktio, joka lisätään tapahtuman sattuessa suoritettavien tapahtumien listaan. Tässä tapauksessa esimerkkinä käytetään kokemusta kuvaavien pisteiden muutoksen aiheuttamaa tapahtumaa. Rivillä 1 nähdään, miten funktiolla on yksi parametri, joka on Lua-taulu. Taulusta haetaan kokemuspisteitä saava pelaaja ja tarkistetaan hänen lepotilansa rivillä 2. Rivillä 3 Lua-taulussa oleva kokemuspisteiden määrä kerrotaan kahdella ja talletetaan takaisin tauluun, jos hän on levännyt. Pelaajalle annetaan tällöin enemmän kokemusta hänen ollessa levännyt. Kokemuspisteiden muokkaaminen kutsumatta kokemuspisteitä muokkaavaa funktiota on tärkeä toiminto, sillä muutoin funktiokutsu voisi aiheuttaa tapahtuman uudelleen. Tästä aiheutuisi rekursiivinen ikuinen toisto.

```lua
local function Experience(args)
  if args.player:IsRested() then
    args.points = args.points*2
  end
end
Register("OnExperience", Experience)

local function Init(args)
  local timer = 1000
  local function Update(args)
    if timer < args.diff then
      print("One second passed!")
      timer = 1000
    else
      timer = timer - args.diff
    end
  end
  args.creature:Register("OnUpdate", Update)
end
CreatureRegister(23478, "OnAIInit", Init)
```

Listaus 6.3: Esimerkkejä Lua-ohjelmista, joissa tapahtumiin sidotaan Lua-funktioita.

Listauksen 6.3 rivillä 20 kutsutaan kaikkialta saatavissa olevaa funktiota `CreatureRegister`, jolla voidaan liittää funktioita iii-tyypin tapahtumiin. Tämä funktio on tarkoitettu NPC-hahmoille ja funktioita on myös muille entiteettityypeille. Funktiolle tulee antaa parametreina tunniste, tapahtuman nimi ja tapahtuman sattuessa suoritettava Lua-funktio. Aliluvussa 6.2.2 esitettiin, miten tunniste on tekstimuotoinen, mutta funktiolle annettu tunniste on numero. Toteutuksessamme tunnisteena käytetään yleisesti entiteeteille niiden tunnistenumeroa. Tekstimuotoinen kenttä sisäisessä toteutuksessa antaa kuitenkin laajemmat mahdollisuudet tarvittaessa. `CreatureRegister`-funktio muuntaa numeron tekstiksi tapahtumankäsittelysysteemiä varten. Se myös esimerkiksi tarkistaa löytyykö tunnisteella NPC-hahmoa tietokannasta ja antaa virheen, jos hahmoa ei löydy. Rekisteriin lisätty Lua-funktio suoritetaan esimerkin tapauksessa NPC-hahmolle käytösolion initialisointitapahtuman sattuessa, jos hahmolla on kyseinen tunniste.

Listauksen 6.3 riveillä 10-17 määritellään sulkeuma, joka liitetään Lua-taulusta haetun NPC-hahmon päivitystapahtumaan `Register`-funktion avulla rivillä 18. Kyseisellä funktiolla liitetään siis i-tyypin tapahtumiin Lua-funktioita. Päivitystapahtumassa päästään käsiksi `diff`-muuttujaan, joka ilmaisee edellisestä päivitystapahtumasta kuluneen ajan millisekunteina. Rivillä 9 alustetaan ajastinmuuttuja tuhanteen millisekuntiin. Rivillä 11

kokeillaan päivitysfunktiossa, onko kulunut aika suurempi kuin jäljellä oleva aika ajastinmuuttujassa. Jos kulunut aika on suurempi, tulostetaan viesti ja ajastinmuuttuja asetetaan uudelleen tuhanteen millisekuntiin. Jos kulunut aika ei ole suurempi, niin ajastimesta vähennetään kulunut aika. Näin voidaan tehdä ajastettuja tapahtumia.

6.3 Tietotyyppien muunnos kielten välillä

Edellä todettiin, että C++-kielestä voi viitata Lua-kielen muuttujiin C-rajapinnan avulla könttäsulautuksen tavoin. Tämän lisäksi on kuitenkin tarve vaihtaa tietoa kielten välillä luonnollisen sulautuksen tavoin, joten kielten erityyppisiä muuttujia tulee voida muuntaa keskenään. Lua-kielessä ja C++-kielessä on eri määrä mahdollisia tietotyyppejä. Lua-kielessä on vain yksi lukutyyppi, mutta C++-kielessä niitä on useita. Lisähaasteena C++-kielen kaikkia lukutyyppejä ei voida esittää Lua-kielen lukutyypin avulla, sillä sen lukuavaruus ei riitä esittämään kaikkia 64-bittisiä kokonaislukuja tarkasti. Lua-kieli ei myöskään tunne käsitettä luokka, jolloin C++-kielen oliot tulee voida esittää jollakin toisella tavalla. Erojen takia on tärkeä määrittää, miten erilaiset C++-kielen tietotyypit ja oliot esitetään Lua-kielessä ja päinvastoin.

Järjestelmä joutuu määrittämään yhtenäisen tavan C++-kielen arvojen viemiseen Lua-kieleen ja takaisin, jotta erilaiset muuttujat voidaan aina tunnistaa ja muuntaa turvallisesti kielten välillä. Toteutuksessamme käytämme apuna Sol2-käärekirjastoa, joka määrittää valmiiksi erilaisia muunnosoperaatioita erityyppisille arvoille C++-kielestä Lua-kieleen ja takaisin [38]. Merkkijonot, monet numeroarvot ja Boolen arvot ovat suoraan muunnettavissa kielten välillä vastaaviksi arvoiksi Lua-kielen C-rajapinnan avulla. Muut arvot, kuten C++-kielen oliot, esitetään kirjaston avulla `userdata`-tyypin kautta käyttäen sen tyyppisiä arvoja edustusolioina. Funktiot kirjasto osaa itse kääriä tarvitsemallaan tavalla, jolloin kirjastolle voidaan suoraan antaa pelkkä osoitin paljastettavaan funktioon. Kirjasto toteuttaa luokat erilaisille Lua-kielen tyypeille, joiden avulla Lua-kielen arvojen viitteet voidaan abstraktoida pois. Luokat määrittävät erilaisia funktioita, joiden avulla esimerkiksi Lua-taulun viitettä voidaan indeksoida C++-kielen assosiaatiotaulun tavoin ilman Lua-kielen C-rajapinnan käyttöä.

```cpp
void Bind(lua_State* L) {
  auto state = sol::state_view(L);
  {
    auto ut = state.new_usertype<MyType>("MyType");
    ut["Method"] = &MyType::Method;
    ut["memberVariable"] = MyType::memberVariable;
  }
  {
    auto ut = state.new_usertype<SubType>(
      "SubType", sol::base_classes, sol::bases<MyType>()
    );
    ut["OverloadedMethod"] = sol::overload(
      sol::resolve<void(float)>(&SubType::OverloadedMethod),
      sol::resolve<void(int)>(&SubType::OverloadedMethod)
    );
  }
}
```

Listaus 6.4: Esimerkki C++-kielen olioiden taulujen määrittämisestä Sol2-kirjaston avulla.

Tyypin `userdata` avulla voidaan esittää C++-kielen luokkia ja olioita. Kyseisen tyyppiseen arvoon liitetyn metataulun avulla voidaan toteuttaa C++-kielen olioiden metodien tyylisiä funktiokutsuja käyttämällä indeksointioperaattoria hyödyksi. Metataulun avulla voidaan arvolle määrittää myös yhtäläisyysfunktio, jolloin identiteetti kahden arvon välillä voidaan tarkistaa. Listauksessa 6.4 nähdään, miten Sol2-kirjaston avulla luodaan metatauluja C++-kielen luokille. Metataulua luodessa sille annetaan nimi, jonka avulla siihen voi viitata Lua-kielen sisältä. Tauluun talletetaan sitten funktioita ja muuttujia, jotka kirjasto käärii siten, että niihin päästään käsiksi Lua-kielestä. Listauksen rivillä 10 voidaan nähdä, miten kirjaston avulla `SubType` perii `MyType`-luokan. Kirjasto tukee myös funktioiden ylimäärittelyä, kuten riveillä 12-15 nähdään. Kun viite `MyType`-tyypin olioon viedään Lua-kieleen, luodaan `userdata`-tyypin muuttuja. Muuttujaan talletetaan osoitin olioon ja olion tyypille luotu metataulu liitetään juuri luotuun muuttujaan. Muuttujalle tehdyt metodikutsut hakevat sitten metataulusta talletetut metodit ja kutsuvat niitä muuttuja ensimmäisenä parametrinaan. Kun muuttujaa halutaan käyttää taas C++-kielessä, tarkistetaan Lua-kielen muuttujaan liitetyn metataulun vastaavuus halutun C++-tyypin metataulun kanssa. Jos taulut ovat samat, luetaan muuttujasta osoittimen arvo, muutoin nostetaan virhe. Sol2-kirjasto pystyy tunnistamaan ja löytämään tyyppitiedosta, esimerkiksi `MyType`-tyypistä, tyypille talletetun taulun. Tämä tapahtuu C++-kielen ominaisuuksien avulla, joilla voidaan muuntaa C++-kielen tyyppi merkkijonoksi. Merkkijonon avulla

tyypin taulu voidaan tallettaa ja hakea.

C++-kielen säiliöt (containers), kuten assosiaatiotaulut ja erilaiset listat, voidaan tarvittaessa muuttaa Lua-kielen `table`-tyypiksi. Muunnos Lua-taulusta takaisin C++-kielen säiliöksi ei kuitenkaan ole suoraviivaista, sillä Lua-taulu on samanaikaisesti sekä assosiaatiotaulu että lista, ja taulun avaimena ja arvona voidaan käyttää `nil`-tyyppiä lukuun ottamatta mitä tahansa Lua-kielen tyyppiä. Muunnosoperaatiot vaativat myös kaikkien elementtien läpikäynnin. Näistä syistä Lua-taulun ja C++-säiliöiden välinen muunnos jätetään järjestelmän käyttäjän huoleksi. Säiliöt voidaan kuitenkin esittää muiden C++-kielen luokkien tavoin. Lua-kielen `thread`-tyyppi on varattu korutiinien toteutukselle, ja sille ei ole C++-kielen vastinetta. Siihen voidaan kuitenkin viitata viitteen avulla C++-kielestä. Lua-kielen `string`-tyyppi käyttää C-tyylisiä merkkijonoja, joita voidaan välittää kielien välillä. Siten C++-kielen C-tyyliseksi merkkijonoksi muunnettavat tyypit, kuten standardikirjaston `std::string`-tyyppi, voidaan esittää Lua-kielessä. Lua-kielen ja C++-kielen Boolen arvot vastaavat toisiaan.

Lua-kielen `number`-tyyppi on 64-bittinen liukuluku, jolloin sitä voidaan käyttää useimpien C++-kielen liukulukujen ja kokonaislukujen esittämiseen Lua-kielessä. Myös `Enum`-tyyppiset C++-kielen arvot esitetään tavallisina lukuina `number`-tyypin avulla Lua-kielessä. Lua-ielen `number`-tyyppi voi esittää peräkkäisiä kokonaislukuja vain väliltä $[-2^{53}-1, 2^{53}]$, joten suurempia lukuja esittävät kokonaislukutyypit, kuten 64-bittiset kokonaislukutyypit, tulee esittää muussa muodossa tarkkuuden säilyttämiseksi. Toteutuksessamme 64-bittiset kokonaislukutyypit esitetään `userdata`-tyypin avulla. Kielen `nil`-tyyppi käsitellään tilannekohtaisesti, esimerkiksi lukuarvoksi muunnettaessa nostetaan virhe, mutta osoittimeksi muunnettaessa se muunnetaan nolla-arvoiseksi osoittimeksi, joka tarkoittaa, ettei osoitin osoita minnekään. Kielen `lightuserdata`-tyypin avulla voi viitata eri olioihin ja muistialueisiin ja minkä tahansa osoittimien voi muuntaa kyseisen tyyppiseksi Lua-muuttujaksi. Muunnoksessa kuitenkin menetetään osoittimen tarkempi tyyppi ja osoitinta myöhemmin käytettäessä C++-kielessä ei välttämättä voida selvittää osoittimen alkuperää. Tuntemattomia osoittimia käytettäessä tulee olla huolellinen, sillä ohjelmointivirhe voi aiheuttaa muistin korruptoitumista ja siksi niitä ei käytetä tässä toteutuksessa.

6.4 Lua-instanssien välinen kommunikointi

Lua-kielen ohjelmissa voidaan haluta tallettaa tietoa esimerkiksi pelaajan tilasta. Lua-instanssit eivät jaa muuttujia, joten tilatietoon ei päästä käsiksi, jos se on talletettu toi-

seen Lua-instanssiin. Pelaaja voi liikkua kartalta toiselle, jolloin yhden kartan tallettamat tiedot Lua-instanssiin ovat toisen kartan Lua-instanssin suorittamien ohjelmien ulottumattomissa. Lua-instanssien välistä tiedonsiirtoa varten on toteutettu tapa sarjallistaa Lua-kielen tietotyyppejä C++-kielen tietotyypeiksi, jotta ne voidaan kuljettaa Lua-instanssista toiseen.

Sarjallistettu tieto talletetaan palvelinohjelman entiteetteihin, kuten pelaajahahmoon, jolloin pelaajaan liitettyyn tilatietoon päästään käsiksi pelaajan liikkuessa kartalta toiselle. Pelaajahahmo paljastaa Lua-ympäristölle sarjallistetun tietonsa, jotta tietoa voidaan lukea ja kirjoittaa. Entiteettiin talletettu tilatieto tuhoutuu entiteetin tuhoutuessa, ellei sitä erikseen talleteta esimerkiksi pysyvään muistiin. Jotta Lua-instanssien välillä voitaisi kommunikoida myös ilman entiteettejä, on toteutettu viestintäjärjestelmä, jolla tietoa voidaan lähettää Lua-instanssista toiseen. Kyseinen järjestelmä käyttää hyväksi samaa sarjallistamista, kuin tilatiedon talletus entiteetteihin. Seuraavissa aliluvuissa esittelemme sarjallistamiseen ja Lua-instanssien väliseen viestintään käytetyt järjestelmät.

6.4.1 Sarjallistaminen

Tiedonsiirrolle hyödylliset Lua-kielen tyypit `nil`, `string`, `number`, `boolean` ja `table` valittiin sarjallistamista varten. Osoittimet on päätetty jättää pois, jotta niiden mahdollisen väärän käytön aiheuttamat riskit vältetään. Lisäksi monia arvoja ei voida muuntaa C++- ja Lua-kielten välillä, joten ne on myös jätetty pois. Lua-muuttujaa C++-ympäristössä esittää `LuaValue`-luokka, joka voi sisältää valitun tyyppisten muuttujien sarjallistetun esityksen ja muuttujan tyypin. Koska Lua-kielessä muuttujat voivat vaihtaa tyyppiä kesken suorituksen, Lua-muuttujan sarjallistaminen alkaa Lua-muuttujan tyypin määrittämisellä. C++-kielen primitiivisiin tyyppeihin suoraan assosioituvat Lua-kielen tyypit voidaan tyyppimäärityksen jälkeen suoraan lukea Lua-kielen rajapinnan tarjoamien funktioiden avulla ja tallettaa `LuaValue`-luokan instansseihin tyyppitiedon kanssa. Kutsutut rajapinnan funktiot valitaan muuttujan tyypin avulla. Muut tyypit, kuten `nil` ja `table`, vaativat muuntamista C++-kielen tietotyyppeihin sopivaksi.

Lua-kielen `nil`-tyyppiä kuvaamaan riittää tallettaa vain tieto muuttujan tyypistä, sillä kyseisen tyyppinen muuttuja ei sisällä muuta tietoa. Tyypin `table` muuttuja voi sisältää avain-arvo-pareja, jotka voivat olla eri Lua-kielen tyyppejä. Lua-taulu sarjallistetaan iteroimalla Lua-taulun avain-arvo-parit, sarjallistamalla yksittäisien parien avain ja arvo `LuaValue`-luokan instansseiksi, ja tallettamalla instanssit avain-arvo-pareiksi C++-kielen

assosiaatiotauluun. Assosiaatiotaulu talletetaan `LuaValue`-luokan instanssiin määritetyn tyypin kanssa. Sykliset Lua-taulut aiheuttavat toteutuksessa ikuisen suoritussilmukan. Syklit tauluissa voidaan havaita pitämällä kirjaa talletetuista Lua-tauluista sarjallistamisprosessin aikana ja käsitellä esimerkiksi nostamalla virhe, mutta tämän toteutus on jätetty myöhemmälle.

`LuaValue`-luokan instanssin muuntaminen takaisin Lua-kielen muuttujaksi tapahtuu vastaavasti talletetun tyyppitiedon avulla. Lua-kielen tyyppeihin assosioituvat C++-kielen tyyppiset muuttujat voidaan suoraan Lua-kielen rajapinnan funktioiden avulla muuntaa Lua-kielen muuttujiksi. Tyypin `nil` sisältävä instanssi käsitellään luomalla Lua-kielen rajapinnan avulla `nil` arvo. Lua-taulua vastaavan C++-kielen assosiaatiotaulun muuntaminen Lua-tauluksi tapahtuu luomalla tyhjä Lua-taulu, iteroimalla C++-kielen assosiaatiotaulun avain-arvo-parit, ja luomalla Lua-tauluun avain-arvo-parit muuntamalla `LuaValue`-tyyppiset avaimet ja arvot Lua-kielen rajapinnan funktioiden avulla Lua-kielen muuttujiksi. Kutsutut rajapinnan funktiot voidaan valita `LuaValue`-instanssin sisältämän tyyppitiedon avulla.

Entiteetteihin voidaan haluta liittää paljon tietoa Lua-taulujen muodossa. Suurten Lua-taulujen sarjallistaminen ja sarjallistetun tiedon muuntaminen takaisin Lua-kielen muuttujiksi vie aikaa. Usein talletetusta tiedosta kuitenkin halutaan päästä käsiksi vain pieneen osaan. Nopeaa osittaista tiedon hakua ja talletusta varten Lua-taulujen sarjallistaminen on toteutettu assosiaatiotaulujen avulla. Assosiaatiotaulujen avainten avulla voidaan hakea ja tallettaa vain tarvittua tietoa, jolloin aikaa kuluu vain tarvitun tiedon muuntamiseen ympäristöstä toiseen.

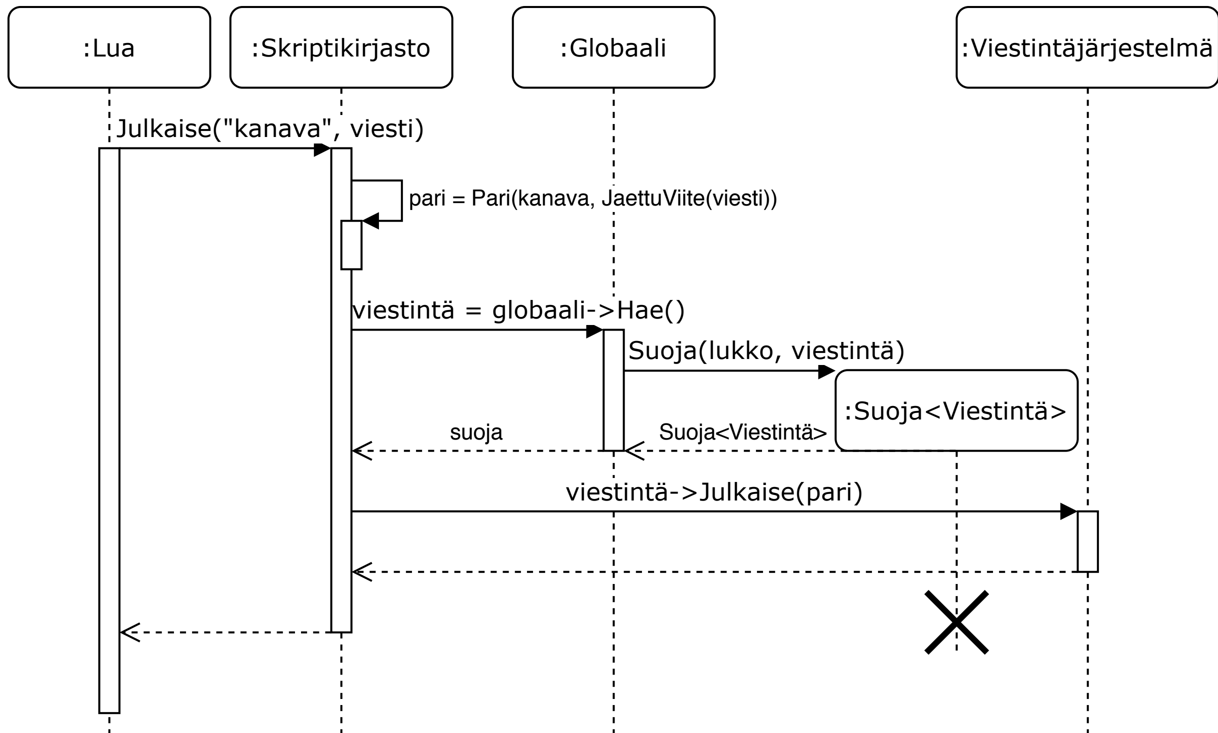
```lua
serialized_number = LuaVal.new(123)
serialized_string = LuaVal.new("example")
serialized_table = LuaVal.new({})

serialized_table.test1 = 5
serialized_table.test2 = serialized_string
serialized_table.foo = {1,2,serialized_number,4,bar={"baz"}}

baz = serialized_table.foo.bar[1]
serialized_table.foo.bar[1] = "qux"

lua_table = serialized_table:asLua()
mixed_table = serialized_table:asLua(1)
for k,v in serialized_table:iterate() do
  print(k, v)
end
```

Listaus 6.5: Käytännön esimerkki Lua-koodista, jossa luodaan ja käsitellään sarjallistettuja muuttujia.

Jotta sarjallistettua muuttujaa voidaan käsitellä Lua-kielestä käsin, paljastetaan `LuaValue`-luokka Lua-ympäristölle `userdata`-tietotyyppinä liittäen tyypin arvoon funktioita käsittelyä varten metataulun avulla. Apuna käytetään Lua-kielen metametodeja, jolloin rajapinta saadaan toimimaan näennäisesti Lua-taulun tavoin. Listauksessa 6.5 havainnollistetaan paljastetun rajapinnan käyttöä. Riveillä 1-3 luodaan Lua-kielen tietotyypeistä sarjallistettuja muuttujia. Riveillä 5-7 sarjallistettuun Lua-tauluun asetetaan sarjallistamatonta ja sarjallistettua tietoa, jolloin avaimena ja arvona käytetty tieto sarjallistetaan rekursiivisesti ja talletetaan olemassaolevan sarjallistetun Lua-taulun sisään. Rivillä 9 sarjallistetusta tiedosta haetaan vain osa avainten `foo`, `bar` ja `1` avulla ja talletetaan se muuttujaan. Rivillä 10 haettu tieto korvataan uudella tiedolla sarjallistetussa muuttujassa, mutta edellä talletettu muuttuja pitää edelleen arvonsa samana. Rivillä 12 sarjallistettu Lua-taulu muutetaan takaisin Lua-tauluksi kokonaisuudessaan ja seuraavalla rivillä näytetään, miten muunnos voidaan tehdä vain annetulle tasolle asti. Rakenteen läpikäymistä varten tarjotaan funktio iteraattorin hakemiseen, jota voi käyttää Lua-kielen silmukassa, kuten riveillä 14-16 voidaan nähdä.

6.4.2 Viestintäjärjestelmä

Ohjelman päivityssyklin halutaan suorittavan mahdollisimman nopeasti ilman keskeytyksiä. Siksi Lua-instanssien välinen viestintä toteutetaan publish–subscribe-mallin [9] avulla



Kuva 6.3: Sekvenssikaavio Lua-koodista tehdyn viestijulkaisun kulusta.

asynkronisesti. Järjestelmän avulla Lua-ohjelmat voivat lähettää viestejä eri kanaville ja vastaanottaa viestejä vain halutuilta kanavilta. Kanavien tunnisteena käytetään merkkijonoja ja kanaville voi lähettää viestinä edellä esitellyn järjestelmän avulla sarjallistettuja arvoja. Järjestelmä tarjoaa Lua-instanssille skriptikirjaston kautta funktion kanavalle liittymiseen, kanavalta poistumiseen, kanavatilauksen tarkistamiseen ja kanavalle julkaisemiseen. Järjestelmä antaa skriptikirjastoinstanssille funktion myös sille varatun viestijonon hakemiseen ja viestijonon sekä kanavatilausten poistamiseen.

Skriptikirjaston-instanssit eivät tiedä toistensa olemassaolosta. Jotta ne voivat välittää Lua-instanssien viestejä keskenään sisältävät ne viitteen yhteiseen olioon. Viestintäjärjestelmä on toteutettu `Kommunikaatio`-luokkaan, jonka instanssi on talletettu skriptikirjastoinstanssien jakamaan `Globaali`-luokan instanssin. Viite jaettuun instanssiin annetaan skriptikirjastoinstanssille sen konstruktorin kautta. Lua-instanssit, ja siten myös skriptikirjastoinstanssit, suorittavat mahdollisesti eri säikeillä. Jotta eheys säilytetään jaettua tietoa käsitellessä, käytetään jaetun instanssin suojaamiseen lukkoa, jonka `Globaali`-luokka tarjoaa. Skriptikirjastoinstanssit poistavat kaiken niihin liitetyn tiedon viestintäjärjestelmästä ennen tuhoutumistaan käyttämällä järjestelmän funktioita.

Kuvassa 6.3 on sekvenssikaavio skriptikirjaston paljastaman viestinjulkaisufunktion kut-

susta Lua-koodissa. Funktio ottaa sisään kanavan tunnisteen ja viestinä sarjallistetun arvon, joista se tekee parin. Viesti talletetaan pariin jaetun viitteen avulla, jolloin yhtä viestiä on olemassa kerrallaan vain yksi ja siihen viitataan useasta paikasta muistin säästämiseksi. Funktio hakee viestijärjestelmän instanssin skriptikirjaston omistaman viitteen avulla. Haettu instanssi on suojattu lukolla hyödyntäen RAII-idiomia, siksi instanssia haettaessa palautuu takaisin olio, joka pitää lukkoa lukittuna, kunnes olio tuhoutuu. Olion kautta päästään käsiksi viestintäjärjestelmään, jonka funktiota kutsumalla pari lisätään jokaiseen annetulle kanavalle liittyneen Lua-instanssin viestijonoon. Funktiokutsuketjun purkautuessa RAII-idiomin mukaisesti lukko avataan.

Järjestelmän muiden funktioiden kutsuminen tapahtuu viestijärjestelmää käyttäen julkaisufunktion tavoin lukkoa apuna käyttäen. Kanavalle liittyessä viestijärjestelmään talletetaan kanavatunnisteeseen assosioitu skriptikirjastoinstanssin uniikki tunniste. Kanavatilauksen tarkistaminen tapahtuu tarkistamalla kanavatunnisteeseen assosioidun skriptikirjastoinstanssin tunnisteen olemassaolo ja kanavatilauksen poistaminen tapahtuu poistamalla kanavatunnisteeseen assosioitu skriptikirjastoinstanssin tunniste. Talletettujen skriptikirjastoinstanssien tunnisteiden avulla kaikille kanavalle liittyneille skriptikirjastoinstansseille voidaan tallettaa viestejä tunnisteisiin assosioituihin listoihin julkaisufunktion kautta. Skriptikirjastoinstanssin tunnisteen avulla instanssin kanavatilaukset voidaan poistaa ja viestilista voidaan hakea ja poistaa.

Viestijärjestelmään kertyvät viestit käsitellään palvelinohjelmaan kovakoodatuissa funktiokutsuissa, esimerkiksi kartan skriptikirjastoinstanssin viestit käsitellään osana karttainstanssin päivitysfunktiokutsua. Viestinkäsittely alkaa hakemalla viestit viestijärjestelmästä skriptikirjastoinstanssin tunnisteen avulla. Viestit käydään läpi kutsuen edellä esitettyä tunnisteeseen sidottua iii-tyypin tapahtumafunktiota käyttäen tunnisteena kanavan tunnistetta. Tapahtumalle annetaan parametrina kanavan tunniste ja sarjallistettu viesti.

6.5 Virheiden käsittely

Tapahtumajärjestelmän funktiot suorittavat Lua-kielen funktioita, joissa voi tapahtua virheitä. Lisäksi aiemmin todettiin, että Lua-kielen tulkin toteutuksen tarjoaman C-kielisen rajapinnan avulla C++-kieli voi nostaa virheen. Tapahtumajärjestelmä voi todeta virheen sattuneen ja voi hakea Lua-kielen ympäristöstä virheviestin. Virheviestissä on yleensä esimerkiksi virheen aiheuttaneen rivin numero, virheen syy ja virheeseen johtaneiden funk-

tiokutsujen pino. Toteutettu tapahtumajärjestelmä antaa virheen sattuessa virheviestin suoraan lokitusjärjestelmälle ja jatkaa sitten suoritusta. Lokitusjärjestelmä tulostaa viestin palvelinohjelman konsoliin ja tiedostoon myöhempää tarkastelua varten.

Edellä todettiin myös, että Lua-ohjelmassa tapahtuvien virheiden lisäksi ohjelman kutsumat C++-kielen funktiot voivat aiheuttaa poikkeuksia. Myös huomattiin, että poikkeus voi aiheuttaa määrittämätöntä toimintaa, jos se kulkee Lua-tulkin toteutuksen kautta takaisin C++-kielen ohjelmalle. Toteutuksessa kaikki Lua-kieleen paljastetut C++-kielen funktiot kääritään funktioon, joka kaappaa kaikki poikkeukset. Poikkeukset muunnetaan sitten tekstimuotoon hakemalla poikkeuksen sisältämä virheviesti. Virheviesti muunnetaan sitten Lua-kielen virheeksi kutsumalla Lua-kielen tulkin toteutuksen tarjoamaa virhefunktiota. Samalla tavoin erilaisten muunnosoperaatioiden virheet on toteutettu aiheuttamaan Lua-kielen virheilmoituksia.

Edellä todettiin, että jo vapautettuun muistiin viittaavan muuttujan käyttäminen johtaa virhetilaan. Tätä tilannetta ei käsitellä toteutuksessa mitenkään. Suunniteltu toteutus käärisi Lua-kieleen paljastetut C++-kielen osoittimet ja tallettaisi kääreeseen arvon, jota muutetaan osoittimen viittaaman olion hajotessa. Arvo tarkistettaisiin aina osoitinta käytettäessä Lua-kielestä, jolloin voidaan havaita olion hajoaminen ja nostaa virhe vapautetun muistin käyttämisen sijaan. Toteutuksessa käytetty Sol2-kirjasto ei kuitenkaan mahdollista vielä tarkistusten tekemistä, vaikka Lua-kielen rajapinta ei aseta tarkistuksen tekemiselle esteitä.

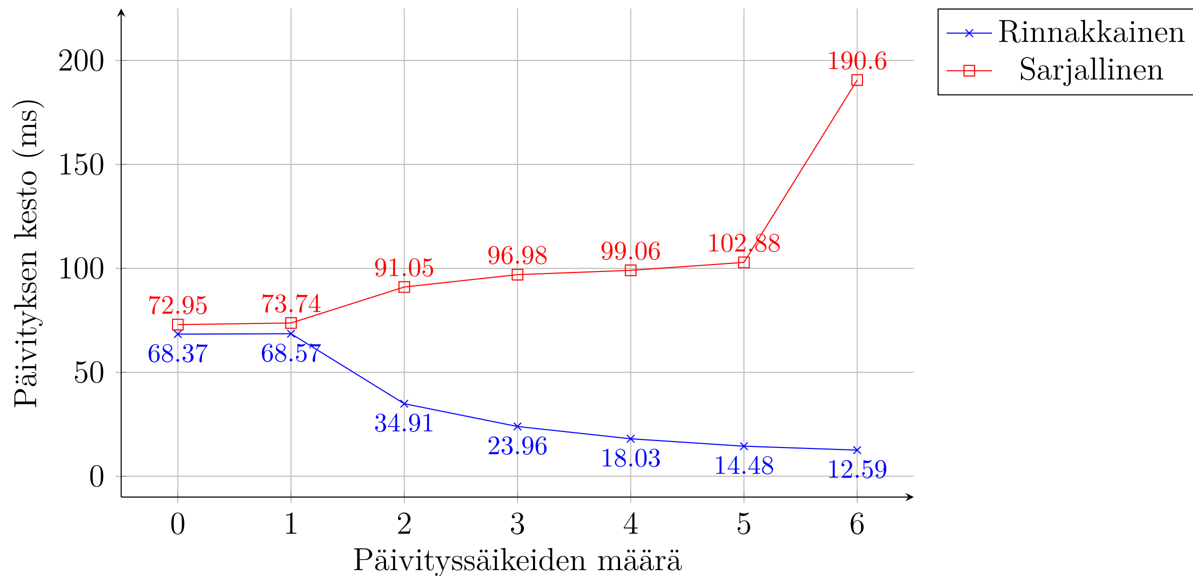
7 Arviointi

Tässä luvussa vertailemme simulaatiotestin avulla toteutuksen suoritusta kahdessa eri tapauksessa: lukolla suojatun yhden tulkki-instanssin tapauksessa ja usean tulkki-instanssin tapauksessa, joita kutsumme sarjalliseksi ja rinnakkaiseksi tapaukseksi. Käymme sitten läpi, täytetäänkö toteutukselle alussa asetetut vaatimukset. Lopuksi listaamme toteutuksen, arvioinnin ja tutkimuksen puutteet.

7.1 Suorituksen vertailu

Rinnakkaisuuden käytön pääasiallinen tarkoitus tässä työssä on vähentää palvelinohjelman päivityksen kokonaiskestoon kuluvan ajan määrää käytettäessä Lua-ohjelmia. Kuten aliluvussa 3.1 todettiin, on saatuja hyötyjä tärkeä mitata käytännössä. Vertaamme sarjallista ja rinnakkaista tapausta toisiinsa suoritusnopeuden kannalta. Nopeuden lisäksi on tärkeää mitata muistin käyttö, sillä rinnakkaisessa lähestymistavassa jokaisella kartalla on oma Lua-instanssi, kun taas sarjallisessa lähestymistavassa on vain yksi Lua-instanssi, jonka kaikki kartat jakavat. Mittausalustana kaikille tuloksille käytettiin 64-bittistä Windows 10 Home 1909 käyttöjärjestelmää kahdeksan gigatavun keskusmuistin ja Intel Core i5-9400F@2.90GHz prosessorin kanssa, jossa on kuusi fyysistä prosessoriydintä ja säiettä. Suoritettu ohjelma on myös 64-bittinen.

Kuvassa 7.1 nähdään palvelinohjelman päivityksen kokonaiskesto käytettäessä tiettyä määrää säikeitä. Kuvassa päivityssäikeiden määrän ollessa nolla päivitetään kartat ohjelman pääsäiettä käyttäen, eikä sitä varten luoda lisäsäikeitä. Päivityssäikeitä käytettäessä pääsäie ei osallistu päivitykseen. Mittaukset on tehty simuloimalla todellista tilannetta käyttämällä oikean pelipalvelimen päivityssyklijärjestelmää, jossa säikeet ottavat karttoja listasta ja suorittavat niiden tarvitsemat päivitysoperaatiot. Simulaatiossa on jätetty pois kaikki palvelinohjelman pelilogiikka. Todellisella palvelimella eri kartoilla on eri määrä entiteettejä. Simulaatiota varten on arvioitu todellisen palvelimen tietojen avulla kartoilla olevan noin 500 entiteettiä, joista jokainen suorittaa viisi tapahtumakutsua joka päivityksen aikana. Mittauksessa karttainstansseja on 80, kuten kuvassa 5.4, joista jokainen suorittaa 2000 tapahtumakutsua toteutetun skriptikirjaston kautta yhden päivityksen aikana. Tapahtumakutsun aikana lasketaan fibonaccin sarjan viides luku listauksessa 7.1



Kuva 7.1: Päivityksen kesto säikeiden määrän funktiona. Kuvan tilanteessa karttoja on 80 ja kartat suorittavat 2000 tapahtumakutsua jokaisen päivityksen aikana.

riveillä 1-6 toteutetulla funktiolla simuloiden Lua-ohjelman tekemää työtä.

```
1 local function Fibonacci(m)
2     if m < 2 then
3         return m
4     end
5     return Fibonacci(m-1) + Fibonacci(m-2)
6 end
7
8 local message = LuaVal.new({1,2,123,4,bar={"baz"}, "qux"})
```

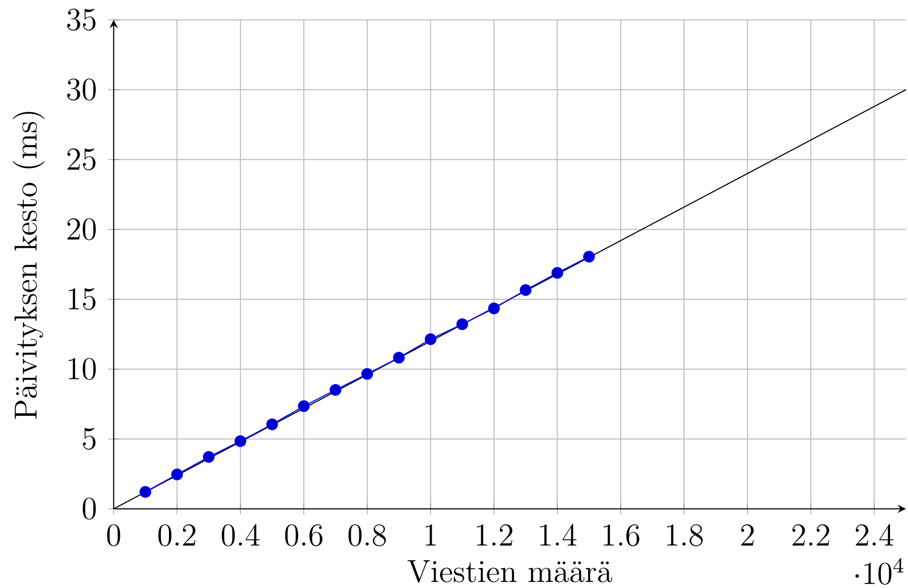
Listaus 7.1: Fibonaccin sarjan lukuja laskevan Lua-funktion toteutus ja viesti-instanssin luonti.

Kuvan tulosten perusteella voidaan huomata rinnakkaisen suorituksen olevan tehokkaampaa käytettyjen säikeiden määrästä riippumatta, joka todennäköisesti johtuu sarjallisen tapauksen tulkkia suojaavan lukon aiheuttamasta lisäprosessoinnista jokaisen tapahtumakutsun yhteydessä. Sarjallisessa tapauksessa käytettyjen säikeiden määrän kasvaessa säikeet joutuvat odottamaan toisiaan yhä enemmän päästäkseen suorittamaan haluamansa tapahtumat Lua-tulkin avulla, joten päivitysten kesto kasvaa. Oikealla palvelimella päivityksen aikana suoritetaan muutakin kuin skriptikirjaston tapahtumia, ja suoritetun Lua-koodin määrä voi vaihdella, jolloin säikeiden odotusaika voi vaihdella. Kuva antaa kuitenkin suuntaa sarjallisen ja rinnakkaisen suorituksen eroihin. Rinnakkaisessa suorituksessa voidaan nähdä suorituksen jakautuminen säikeiden kesken, esimerkiksi kuusi säiettä an-

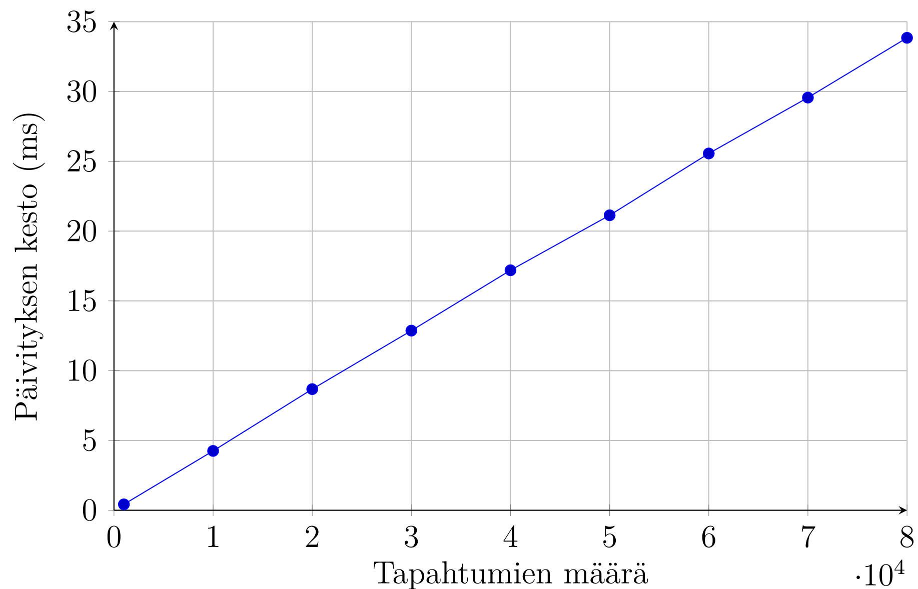
taa noin 5.4-kertaisen nopeutuksen alaspäin pyöristettäessä, joka on melko lähellä ideaalia kuusinkertaista nopeutusta. Kuuden säikeen päivityksen kesto kasvaa muihin säiemääriin verrattuna huomattavasti sarjallisessa tapauksessa toistetuista mittauksista huolimatta. Syy kasvulle ei ole selkeä, mutta indikoi järjestelmän kaikkien säikeiden käytön olevan kyseisessä tapauksessa haitallinen. Kasvu voi johtua esimerkiksi käyttöjärjestelmän keskeytyksistä, jotka näkyvät ohjelman suorituksessa sen käyttäessä kaikkia järjestelmän prosessointiresursseja. Kuvasta nähdään, miten rinnakkainen toteutus on simuloidussa tilanteessa pystynyt pitämään päivityksen keston alle 30 millisekunnin, eli sen avulla on päästy tavoiteajan alapuolelle. Sarjallinen suoritus sen sijaan ei pääse tavoiteaikaan simuloidussa tilanteessa ollenkaan.

Muistinkulutusta testattiin mittaamalla skriptikirjastoinstanssien luontien välillä tapahtuvaa muistinkulutuksen muutosta. Skriptikirjaston instansseja luotiin kymmenen, yksi kerrallaan, ja jokaisen jälkeen ohjelman kokonaismuistinkulutus mitattiin. Pienimmillään muistinkulutus kasvoi 60.77 ja suurimmillaan 61.65 kilotavua uuden skriptikirjaston instanssin luonnin takia. Julkisen metriikan [61] mukaan pelaajia oli kerrallaan paikalla alle 500, joista jokainen voisi olla omassa karttainstanssissaan. Kyseisessä tilanteessa voitaisiin siis rinnakkaisella lähestymistavalla maksimissaan kuluttaa noin 500-kertainen määrä muistia sarjalliseen tapaan verrattuna. Yhden instanssin kuluttaman muistin perusteella se olisi vain noin 30 megatavua. Muistinkulutusmittaus on kuitenkin otettava vain suuntaa antavana, sillä todellisessa tilanteessa Lua-instansseissa on tallennettuna erilaisia määriä tietoa Lua-ohjelmien tarvitsemiin muuttujiin ja tietorakenteisiin, joita mittauksissa ei ole huomioitu. Kuitenkin vaikka muistia käytettäisiin todellisuudessa 3 gigatavua, eli 100-kertainen määrä mitattuun verrattuna, ei se olisi välttämättä kovinkaan suuri osuus esimerkiksi 32 gigatavun muistista.

Kuvassa 7.2 nähdään yhden karttainstanssin päivityksen kesto viestien määrän funktiona. Karttainstanssi tekee yhden päivityksen aikana yhden tapahtuman, jossa lähetetään ja vastaanotetaan kuvan osoittama määrä viestejä. Kuvassa on ekstrapoloitu viiva 30 millisekunnin rajaan asti päivityksen kestolle. Lähetetty viesti on luotu etukäteen ohjelman alussa listauksen 7.1 rivillä 8 osoitetulla tavalla, missä viesti sisältää erilaisia lukuja, merkkijonoja ja yhden sisäkkäisen rakenteen. Kuvassa 7.3 taas nähdään yhden karttainstanssin päivityksen kesto tapahtumien määrän funktiona. Kuvista nähdään, että yksittäinen kartta voi yhdellä säikeellä lähettää ja vastaanottaa 30 millisekunnin aikana maksimissaan 25000 edellä mainittua viestiä tai tehdä maksimissaan noin 70000 tapahtumafunktiokutsua, joissa suoritetaan muutamia funktiokutsuja.



Kuva 7.2: Yhden karttainstanssin päivityksen kesto sen tehdessä päivityksen aikana tapahtumakutsu, jossa lähetetään ja vastaanotetaan kuvan osoittama määrä viestejä. Kuvassa on ekstrapoloitu viiva mittaustuloksien avulla 30 millisekunnin rajaan asti.



Kuva 7.3: Yhden karttainstanssin päivityksen kesto tapahtumien määrän funktiona. Yksittäisessä tapahtumassa työtä simuloidaan laskemalla fibonaccin sarjan viides luku.

7.2 Vieraskielen sulauttamisen vaatimukset

Toteutuksen alussa asetettiin 5 vaatimusta, jotka toteutuksen tulisi täyttää. Tässä aliluvussa käymme läpi, täytettiinkö vaatimukset ja millä tavalla.

Vaatimuksen 1 mukaan Lua-koodia tulee voida suorittaa yhtäaikaisesti. Edellä tehty mittaus osoittaa toteutuksen pystyvän käyttämään useita suoritussäikeitä hyväksi ja suorittamaan karttojen Lua-ohjelmia rinnakkain vähentäen Lua-ohjelmien vaikutusta päivitystiheyteen ja täyttää siten vaatimuksen. Lisäksi arviointi osittaa, että rinnakkainen suoritus ilman säikeiden välistä viestintää on keskimäärin ytimien määrästä riippumatta nopeampaa, kuin käytettäessä yksittäistä Lua-instanssia globaalin tulkkilukon kautta. Tulosten mukaan muistinkulutus kasvaa moninkertaiseksi rinnakkaisen toteutuksen takia. Muistinkulutus on kuitenkin käytännössä vähäistä.

Vaatimuksen 2 mukaan sulautuksen toteutus ei saa riippua palvelinohjelman toteutuksesta. Toteutus on tehty täysin itsenäiseksi osaksi, joten se ei käytä palvelinohjelman koodia. Toteutuksen rajapinta koostuu tapahtumafunktioista, joiden kutsupaikan ja parametrit sulautuksen toteuttaja voi päättää tapauskohtaisesti, ja Sol2-kirjaston avulla toteutetusta Lua-kielen laajennosrajapinnasta, jonka avulla Lua-kielelle voidaan päättää tapauskohtaisesti paljastaa erilaisia C++-kielen funktioita ja rakenteita. Skriptikirjaston instansseja voidaan vapaasti luoda useita ja niille voidaan paljastaa erilaisia tapahtumia ja funktioita tarpeen mukaan. Toteutus ei siis riipu palvelinohjelman toteutuksesta koodinsa eikä rajapintansa kautta ja täyttää siten vaatimuksen. Seurauksena toteutus voidaan lisätä helposti useaan erilaiseen ohjelmaan. Toteutus voi kuitenkin rajoittaa erilaisia sulautusmahdollisuuksia, esimerkiksi Sol2-käärekirjaston käyttö helpottaa ohjelmointia, mutta samalla pakottaa skriptikirjaston käyttämiin konventioihin.

Vaatimuksen 3 mukaan Lua-ohjelmia tulee voida lisätä, poistaa ja muokata ilman palvelinohjelman uudelleenkäynnistystä. Edellä esitetään, miten sulautuksen toteutuksessa karttainstanssit omistavat skriptikirjaston instanssin, joka voidaan tuhota ja luoda uudelleen esimerkiksi kartan päivityksen suorituksen aikana. Tuloksena vapautetun instanssin muisti, ja siten funktiot ja muuttujat, on vapautettu kokonaan ja uusi skriptikirjaston instanssi lataa levyltä uudet Lua-ohjelmat. Eri entiteetteihin talletettu sarjallistettu tieto pysyy muuttumattomana, ellei sitä erikseen aseteta uudelleen Lua-ohjelmasta tai palvelinohjelmasta. Lua mahdollistaa myös tiedostojen lataamisen ajon aikana. Yhdessä Lua-funktioiden poiston tapahtumalistasta kanssa sen avulla Lua-ohjelma voi siis minkä tahansa tapahtuman yhteydessä poistaa entiteettiin liitetyt tapahtumat ja suorittaa uuden

Lua-ohjelman levyltä, joka rekisteröi uudet tapahtumat. Sulautuksen toteutus siis määrää paljastamansa rajapinnan kautta toteutetaanko vaatimus vai ei, mutta skriptikirjaston toteutus ei estä vaatimuksen toteuttamista.

Vaatimuksen 4 mukaan Lua-ohjelmassa tapahtuva virhe ei saa voida keskeyttää palvelinohjelman suorittamista. Edellä kuvattiin, miten tapahtumaan liitettyjen funktioiden virheet käsitellään. Käsittelyssä Lua-ohjelman tuottamat virheet voidaan kaapata ja sitten sivuuttaa. Niihin liittyvät virheviestit lokitetaan. C++-koodin tuottamat poikkeukset käsitellään kaappaamalla ne ja muuntamalla ne Lua-kielen virheiksi, jolloin kaikki virheet käyttävät samaa käsittelymekanismia. Lua-kieleen voidaan siirtää esimerkiksi viitteitä pelaajiin ja pelaajan viite on validi vain tietyn kartan kontekstissa. Sarjallistaminen ei tue C++-kielen osoittimia, joka estää viitteiden siirtämisen skriptikirjaston instanssista toiseen ja siten estää myös viitteiden siirtämiseen väärään kontekstiin. Viitteitä voidaan kuitenkin käyttää väärin saman skriptikirjastoinstanssin sisällä. Muuttujia, jotka viittaavat jo poistettuihin olioihin voidaan tallettaa ja käyttää ja niiden käytöstä aiheutuvia virheitä ei voida havaita tai estää. Vaatimus siis täytetään Lua-kielen tuottamien virheiden ja C++-kielen virheiden osalta, muttei osoittimien virheellisen käytön osalta.

Vaatimuksen 5 mukaan päivityssyklin suoritusaika ei saa nousta yli 30 millisekuntia toteutuksen takia. Edellä tehty suorituksen vertailu näyttää, että kolmea tai useampaa säiettä käyttämällä voidaan päästä tavoitteeseen simulaation tilanteessa rinnakkaista toteutusta käyttämällä. Vaatimus siis täyttyy simulaation tilanteessa, mutta on riippuvainen esimerkiksi käytettyjen säikeiden ja suoritettavan koodin määrästä, eikä voida suoraan sanoa sen täyttyvän kaikissa tilanteissa. Lisäksi tilanteessa, missä viestintää säikeiden välillä ei tapahdu tai se on vähäistä, voidaan todeta rinnakkaisen toteutuksen täyttävän vaatimuksen säikeitä lisäämällä tilanteissa, missä sarjallinen toteutus ei sitä täytä.

7.3 Puutteet

Tutkimus, toteutus ja mittaukset on kaikki tehty yhden henkilön toimesta, jolloin tutkimuksen kaikissa osa-alueissa voi esiintyä puolueellisuutta. Tutkimus käsittää vain pienen osan kokonaisuuden laajuudesta ja siksi siinä on luonnollisesti puutteita. Kieltenvälinen rajapinta on monimutkainen aihe, jota on tutkittu matemaattisin ja käytännön keinoin. Ratkaisuja sulautuksen ja monisäikeisyyden ongelmiin on ehdotettu laitteistotason muutoksista aina sulautettujen kielten suunnittelun tasolle asti. Tässä tutkimuksessa käsiteltiin vallitsevia ja käytännönläheisiä ongelmia ja ratkaisuja.

Toteutus käsittelee Lua-kielen tulkin sulautusta C++-kieliseen ohjelmaan tilanteessa, jossa on aikavaatimuksia. Toteutusta ei päästy kokeilemaan todellisessa tilanteessa eikä todellisessa palvelinympäristössä. Vertailu yksisäikeisen ja monisäikeisen toteutuksen kanssa on tehty samalla toteutuksella rajoittaen suoritus yksisäikeiseksi. Kuitenkin todellisessa tilanteessa toteutusten erilaiset ratkaisut voisivat olla eriävät ja tuottaa hieman erilaisen tuloksen. Toteutusta ei ole toistaiseksi pyritty optimoimaan. Lisäksi toteutuksessa Lua-kielelle paljastettu C++-kielen toiminnallisuus on hyvin vähäistä. Lua-kielen suoritusta simuloitiin yksinkertaisella ohjelmalla, eikä tuotannossa käytettävällä ohjelmalla. Näiden syiden takia mittaukset ovat vain suuntaa antavia ja absoluuttisia lukuarvoja tärkeämpi on eri mittausten suhde toisiinsa.

Monisäikeisen toteutuksen myötä tulkki-instanssien tulee kommunikoida. Kommunikoinnin toteutus asettaa erilaisia rajoituksia ja lisäprosessointia verrattuna Lua-ohjelmien kommunikointiin yhden tulkki-instanssin sisällä. Työssä oletetaan, että tarvittu kommunikointi on vähäistä ja siksi sen arviointi on vähäistä. Työssä tehty toteutus olettaa, että yhdessä loogisessa kokonaisuudessa voidaan ja halutaan käyttää aina samaa tulkki-instanssia. Tämä ei kuitenkaan välttämättä ole yleisesti haluttu rajoitus. Toteutuksessa ei myöskään ole huomioitu ylläpitoaspekteja eikä dokumentointia.

8 Yhteenveto

Lähdimme tarkastelemaan kielten yhteentoimivuuden ongelmia ja ratkaisuja sulauttaessa yhden kielen tulkki toiseen kieleen monisäikeisessä ympäristössä kirjallisuuden ja tapaustutkimuksen avulla. Tutkimuksen tavoitteet jaettiin ja esitettiin tutkimuskysymysten avulla. Tarkastelimme ensin kirjallisuuden avulla ongelmia ja ratkaisuja valitussa ongelma-alueessa. Toteutimme kirjallisuuden pohjalta tapaustutkimuksen kohteeseen skriptikirjaston, jonka avulla ratkaisujen käyttöä arvioitiin käytännössä.

Kävimme läpi kielten välisen rajapinnan eri osa-alueiden ongelmia ja ongelmien ratkaisuja sulautettaessa yhden kielen tulkki toiseen kieleen. Löysimme ongelmia liittyen kielten arvojen käyttämiseen kielten välisen rajapinnan läpi ja virheiden käsittelyn eroavaisuuksiin. Löysimme kirjallisuudesta joitakin ratkaisuja arvojen käyttöön, kuten muunnosoperaatioiden, viitteiden ja kääreiden käyttö ja ratkaisun virheidenkäsittelyyn kaappaamalla virheet ja muuntamalla ne toisen kielen virheidenkäsittelylle sopiviksi.

Käsittelimme monisäikeisyyttä, sen tuomia ongelmia ja ongelmien ratkaisuja liittyen sulautetun tulkin käyttöön. Löysimme ongelmia monisäikeisyyden käyttämisestä tulkkien kanssa, ongelmiin käytettyjen ratkaisujen tuomasta monimutkaisuudesta, ja ongelmien ratkaisujen mahdollisista tehokkuudesta. Pääasialliset ratkaisut tulkkien käytössä monisäikeisessä ohjelmassa olivat tehdä useita instansseja tulkista tai käyttää yhtä tulkkia, joka suojataan lukolla monisäikeiseltä suoritukselta. Näimme suorituksen vertailusta, että yhden lukolla suojatun tulkin käyttö aiheuttaisi rinnakkaisuuden tuomien etujen menetystä. Monen tulkki-instanssin käyttö taas vaati arvojen sarjallistamista tietoa välitettäessä tulkista toiseen.

Kävimme läpi toteutuksen arkkitehtuuria ja ongelmiin käytettyjä ratkaisuja. Tapaustutkimuksen kohde ja sen vaatimukset asettivat joitakin erityisiä rajoitteita, jotka ohjasivat tiettyjen ratkaisujen käyttöön. Esitimme toteutukselle tehdyn suorituksen vertailun. Siinä verrattiin lukolla suojatun yksittäisen tulkki-instanssin suoritusnopeutta ja muistinkulutusta simuloidussa tilanteessa monen tulkki-instanssin ratkaisuun. Analyysin pohjalta huomattiin, että rinnakkaisen toteutuksen avulla voitiin pysyä ennalta asetetuissa aikamääreissä suorituksen osalta. Saavutettu nopeutus oli suhteellisen lähellä ideaalia, noin 5.4-kertainen kuudella ytimellä. Muistinkäytön arvioitiin nousevan rinnakkaisen toteutuksen tuloksena simuloidussa tilanteessa noin sata tai jopa viisisataakertaiseksi riippuen kart-

tainstanssien määrästä. Muistinkulutus pysyi kuitenkin suhteellisen pienenä, esimerkiksi 500 karttainstanssin tapauksessa arvioitiin muistinkulutuksen olevan noin 30 megatavua.

Hyödyistä huolimatta karttainstanssien välisen viestinnän hitaus sarjallistamisen takia, viestinnän asynkronisuus rinnakkaisuuden säilyttämiseksi, ja rinnakkaisuuden tuoma monimutkaisuus toteutukseen ovat rinnakkaisen lähestymistavan ratkaisemattomat ongelmat. Instanssien välisen kommunikoinnin odotetaan kuitenkin olevan vähäistä ja rinnakkaisuuden tuoman nopeutuksen odotetaan olevan ratkaiseva hyöty.

Tulevaan työhön kuuluu toteutuksen optimointi, testaus todellisessa käytössä ja osoittimien virheellisen käytön havaitsemismekanismien toteutus. Lisäksi tärkeäksi osaksi todettiin rajapinnan kattava ja ajantasainen dokumentointi. Dokumentaation muutosten dokumentointi eri rajapinnan versioiden välillä on erityisen tärkeää tapaustutkimuksen tilanteessa, jossa kielen dynaaminen tyypitys estää ohjelmien staattisen analyysin tekemisen rajapintayhteensopivuusongelmien havaitsemiseksi. Tuleva työ voisi käsitellä dokumentaation automaattista generointia ja dokumentaation muutosten kirjaamista eri versioiden välillä.

Kirjallisuus

[1] D. Abrahams ja A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[2] G. M. Amdahl. ”Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. Teoksessa: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, s. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. URL: https://dl.acm.org/doi/10.1145/1465482.1465560.

[3] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde ja K. Chard. ”Parsl: Pervasive Parallel Programming in Python”. Teoksessa: *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC ’19. Phoenix, AZ, USA: Association for Computing Machinery, 2019, s. 25–36. ISBN: 9781450366700. DOI: 10.1145/3307681.3325400. URL: https://dl.acm.org/doi/10.1145/3307681.3325400.

[4] D. M. Beazley et al. ”SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++.” Teoksessa: *Tcl/Tk Workshop*. Vol. 43. 1996.

[5] D. M. Beazley. ”An Extensible Compiler for Creating Scriptable Scientific Software”. Teoksessa: *Computational Science — ICCS 2002*. Toim. P. M. A. Sloot, A. G. Hoekstra, C. J. K. Tan ja J. J. Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, s. 824–833. ISBN: 978-3-540-46080-0.

[6] J. Bentley. ”Programming pearls: associative arrays”. *Communications of the ACM* 28.6 (1985), s. 570–576.

[7] N. Benton. ”Embedded interpreters”. *J. Funct. Program.* 15 (heinäkuu 2005), s. 503–542. DOI: 10.1017/S0956796804005398.

[8] E. D. Berger ja B. G. Zorn. ”DieHard: Probabilistic Memory Safety for Unsafe Languages”. Teoksessa: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’06. Ottawa, Ontario, Cana-

da: Association for Computing Machinery, 2006, s. 158–168. ISBN: 1595933204. DOI: 10.1145/1133981.1134000. URL: https://dl.acm.org/doi/10.1145/1133981.1134000.

[9] K. Birman ja T. Joseph. ”Exploiting Virtual Synchrony in Distributed Systems”. Teoksessa: *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*. SOSP ’87. Austin, Texas, USA: Association for Computing Machinery, 1987, s. 123–138. ISBN: 089791242X. DOI: 10.1145/41457.37515. URL: https://dl.acm.org/doi/10.1145/41457.37515.

[10] R. D. Cameron ja M. R. Ito. ”Grammar-Based Definition of Metaprogramming Systems”. *ACM Trans. Program. Lang. Syst.* 6.1 (tammikuu 1984), s. 20–54. ISSN: 0164-0925. DOI: 10.1145/357233.357235. URL: https://dl.acm.org/doi/10.1145/357233.357235.

[11] E. Chailloux, P. Manoury ja B. Pagano. *Développement d’applications avec Objective Caml*. Viitattu linkitettyyn käännökseen, joka haettu 05.06.2020. O’Reilly Sebastopol, CA, 2000. URL: https://caml.inria.fr/pub/docs/oreilly-book/.

[12] L. Dagum ja R. Menon. ”OpenMP: an industry standard API for shared-memory programming”. *IEEE Computational Science and Engineering* 5.1 (1998), s. 46–55.

[13] B. Daloze, A. Tal, S. Marr, H. Mössenböck ja E. Petrank. ”Parallelization of Dynamic Languages: Synchronizing Built-in Collections”. *Proc. ACM Program. Lang.* 2.OOPSLA (lokakuu 2018). DOI: 10.1145/3276478. URL: https://doi.org/10.1145/3276478.

[14] E. W. Dijkstra. ”Solution of a Problem in Concurrent Programming Control”. *Commun. ACM* 8.9 (syyskuu 1965), s. 569. ISSN: 0001-0782. DOI: 10.1145/365559.365617. URL: https://dl.acm.org/doi/10.1145/365559.365617.

[15] *Doxygen*. Toukokuu 2020. URL: https://www.doxygen.nl/index.html.

[16] S. Feferman. ”Transfinite Recursive Progressions of Axiomatic Theories”. *J. Symbolic Logic* 27.3 (syyskuu 1962), s. 259–316. URL: https://projecteuclid.org:443/euclid.jsl/1183734529.

[17] E. Fortuna, O. Anderson, L. Ceze ja S. Eggers. ”A limit study of JavaScript parallelism”. Teoksessa: *IEEE International Symposium on Workload Characterization (IISWC’10)*. 2010, s. 1–10.

[18] E. Gamma, R. Helm, R. Johnson ja J. Vlissides. ”Design Patterns: Abstraction and Reuse of Object-Oriented Design”. Teoksessa: *Pioneers and Their Contributions to Software Engineering: sd&m Conference on Software Pioneers, Bonn, June 28/29, 2001, Original Historic Contributions*. Toim. M. Broy ja E. Denert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, s. 361–388. ISBN: 978-3-642-48354-7. DOI: 10.1007/978-3-642-48354-7_15. URL: https://doi.org/10.1007/978-3-642-48354-7_15.

[19] V. Gramoli. ”More than You Ever Wanted to Know about Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms”. *SIGPLAN Not.* 50.8 (tammikuu 2015), s. 1–10. ISSN: 0362-1340. DOI: 10.1145/2858788.2688501. URL: https://dl.acm.org/doi/10.1145/2858788.2688501.

[20] V. Harko ja J. Magnus. ”Demystifying guilds: MMORPG-playing and norms”. Teoksessa: *DiGRA ཅ - Proceedings of the 2009 DiGRA International Conference: Breaking New Ground: Innovation in Games, Play, Practice and Theory*. Brunel University, syyskuu 2009. URL: http://www.digra.org/wp-content/uploads/digital-library/09291.50401.pdf.

[21] R. Hickey. ”The Clojure Programming Language”. Teoksessa: *Proceedings of the 2008 Symposium on Dynamic Languages*. DLS ’08. Paphos, Cyprus: Association for Computing Machinery, 2008. ISBN: 9781605582702. DOI: 10.1145/1408681.1408682. URL: https://dl.acm.org/doi/10.1145/1408681.1408682.

[22] M. D. Hill ja M. R. Marty. ”Amdahl’s Law in the Multicore Era”. *Computer* 41.7 (2008), s. 33–38.

[23] K. Hinsen. ”The Promises of Functional Programming”. *Computing in Science Engineering* 11.4 (2009), s. 86–90.

[24] P. Hudak. ”Modular domain specific languages and tools”. *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)* (1998), s. 134–142.

[25] D. Huffman. ”The synthesis of sequential switching circuits”. *Journal of the Franklin Institute* 257.3 (1954), s. 161–190. ISSN: 0016-0032. DOI: https://doi.org/10.1016/0016-0032(54)90574-8. URL: http://www.sciencedirect.com/science/article/pii/0016003254905748.

[26] R. Ierusalimschy. *Programming in Lua*. 2nd. Roberto Ierusalimschy, 2006.

[27] R. Ierusalimschy, L. H. De Figueiredo ja W. Celes. ”The evolution of an extension language: A history of Lua”. Teoksessa: *Proceedings of V Brazilian Symposium on Programming Languages, pages B–14–B–28*. 2001.

[28] R. Ierusalimschy, L. H. de Figueiredo ja W. Celes. ”Passing a Language through the Eye of a Needle: How the Embeddability of Lua Impacted Its Design”. *Queue* 9.5 (toukokuu 2011), s. 20–29. ISSN: 1542-7730. DOI: 10.1145/1978862.1983083. URL: https://dl.acm.org/doi/10.1145/1978862.1983083.

[29] R. Ierusalimschy, L. H. de Figueiredo ja W. Celes. ”The Evolution of Lua”. Teoksessa: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: Association for Computing Machinery, 2007, 2–1–2–26. ISBN: 9781595937667. DOI: 10.1145/1238844.1238846. URL: https://doi.org/10.1145/1238844.1238846.

[30] *Javadoc*. Toukokuu 2020. URL: https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html.

[31] B. Juurlink ja C. Meenderinck. ”Amdahl’s law for predicting the future of multicores considered harmful”. *ACM Sigarch Computer Architecture News* 40 (toukokuu 2012), s. 1–9. DOI: 10.1145/2234336.2234338.

[32] V. Kahlon, N. Sinha, E. Kruus ja Y. Zhang. ”Static Data Race Detection for Concurrent Programs with Asynchronous Calls”. Teoksessa: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC/FSE ’09. Amsterdam, The Netherlands: Association for Computing Machinery, 2009, s. 13–22. ISBN: 9781605580012. DOI: 10.1145/1595696.1595701. URL: https://dl.acm.org/doi/10.1145/1595696.1595701.

[33] T. Khot. ”Parallelization in Python”. *XRDS* 23.3 (huhtikuu 2017), s. 56–58. ISSN: 1528-4972. DOI: 10.1145/3063591. URL: https://dl.acm.org/doi/10.1145/3063591.

[34] J. Kurs, J. Vraný ja A. Bergel. ”Supporting Language Interoperability by Dynamically Switched Behaviors”. Teoksessa: *Proceedings of the Dateso 2011 Workshop*. Vol. 706. Tammikuu 2011, s. 73–84. URL: http://www.ceur-ws.org/Vol-706/.

[35] Z. D. Luo, L. Hillis, R. Das ja Y. Qi. ”Effective Static Analysis to Find Concurrency Bugs in Java”. Teoksessa: *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. 2010, s. 135–144.

[36] T. M. Malone. ”Interoperability in Programming Languages”. Teoksessa: 2014.

[37] J. Matthews ja R. B. Findler. ”Operational Semantics for Multi-Language Programs”. Teoksessa: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’07. Nice, France: Association for Computing Machinery, 2007, s. 3–10. ISBN: 1595935754. DOI: 10.1145/1190216.1190220. URL: https://doi.org/10.1145/1190216.1190220.

[38] J. Meneide. *Sol2: a C++ <-> Lua API wrapper (Version 3.0.3)*. [Software]. 1. tammikuuta 2020. URL: https://github.com/ThePhD/sol2.

[39] G. E. Moore. ”Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.” *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), s. 33–35.

[40] A. L. D. Moura ja R. Ierusalimschy. ”Revisiting Coroutines”. *ACM Trans. Program. Lang. Syst.* 31.2 (helmikuu 2009). ISSN: 0164-0925. DOI: 10.1145/1462166.1462167. URL: https://doi.org/10.1145/1462166.1462167.

[41] M. Pall. *LuaJIT*. [Software]. 11. syyskuuta 2020. URL: https://luajit.org/.

[42] G. Pinto, W. Torres ja F. Castor. ”A Study on the Most Popular Questions about Concurrent Programming”. Teoksessa: *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*. PLATEAU 2015. Pittsburgh, PA, USA: Association for Computing Machinery, 2015, s. 39–46. ISBN: 9781450339070. DOI: 10.1145/2846680.2846687. URL: https://dl.acm.org/doi/10.1145/2846680.2846687.

[43] *Programming languages — C*. N1256 draft for C99 standard. ISO/IEC. 7. syyskuuta 2007. URL: http://www.open-std.org/jtc1/sc22/wg14/www/projects.

[44] S. Qadeer ja D. Wu. ”KISS: Keep It Simple and Sequential”. Teoksessa: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, kesäkuu 2004. URL: https://www.microsoft.com/en-us/research/publication/kiss-keep-simple-sequential/.

[45] W. C. R. Ierusalimschy L. H. de Figueiredo. *Lua 5.1 Reference Manual*. Haettu 18.09.2020. Lua.org, PUC-Rio. Elokuu 2006. URL: https://www.lua.org/manual/5.1/.

[46] N. Ramsey. ”Embedding an Interpreted Language Using Higher-Order Functions and Types”. Teoksessa: *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*. IVME ’03. San Diego, California: Association for Computing Machinery, 2003, s. 6–14. ISBN: 1581136552. DOI: 10.1145/858570.858571. URL: https://dl.acm.org/doi/10.1145/858570.858571.

[47] A. Savidis. ”Supporting Cross-Language Exception Handling When Extending Applications with Embedded Languages”. Teoksessa: *Software Engineering for Resilient Systems*. Toim. E. A. Troubitsyna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, s. 93–99. ISBN: 978-3-642-24124-6.

[48] N. Shavit ja D. Touitou. ”Software Transactional Memory”. Teoksessa: *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC ’95. Ottowa, Ontario, Canada: Association for Computing Machinery, 1995, s. 204–213. ISBN: 0897917103. DOI: 10.1145/224964.224987. URL: https://doi.org/10.1145/224964.224987.

[49] P. H. Shiu, Y. Tan ja V. J. Mooney. ”A Novel Parallel Deadlock Detection Algorithm and Architecture”. Teoksessa: *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*. CODES ’01. Copenhagen, Denmark: Association for Computing Machinery, 2001, s. 73–78. ISBN: 1581133642. DOI: 10.1145/371636.371684. URL: https://dl.acm.org/doi/10.1145/371636.371684.

[50] A. Skyrme, N. Rodriguez ja R. Ierusalimschy. ”Exploring Lua for Concurrent Programming”. *Journal of Universal Computer Science* 14.21 (1. joulukuuta 2008), s. 3556–3572. URL: http://www.jucs.org/jucs_14_21/exploring_lua_for_concurrent.

[51] B. Smith, M. I. of Technology. Laboratory for Computer Science ja P. M. (I. of Technology). *Reflection and Semantics in a Procedural Language*. MIT/LCS/TR. Massachusetts Institute of Technology, Department of Electrical Engineering ja Computer Science, 1982.

[52] R. Smith. *Working Draft, Standard for Programming Language C++*. N4659 draft for C++17 standard. ISO/IEC. 17. lokakuuta 2017. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/standards.

[53] M. Soldevila, B. Ziliani ja D. Fridlender. *Understanding Lua’s Garbage Collection – Towards a Formalized Static Analyzer*. 2020. arXiv: 2005.13057 [cs.PL].

[54] K. Stencel ja P. Wegrzynowicz. ”Implementation Variants of the Singleton Design Pattern”. Teoksessa: *On the Move to Meaningful Internet Systems: OTM 2008 Workshops*. Toim. R. Meersman, Z. Tari ja P. Herrero. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, s. 396–406. ISBN: 978-3-540-88875-8.

[55] B. Stroustrup. *The Design and Evolution of C++*. Pearson Education, 1994. ISBN: 9780135229477.

[56] S. Sulyman. ”Client-Server Model”. *IOSR Journal of Computer Engineering* 16 (tammikuu 2014), s. 57–71. DOI: 10.9790/0661-16195771.

[57] G. Sussman ja G. Steele. ”Scheme: A Interpreter for Extended Lambda Calculus”. *Higher-Order and Symbolic Computation* 11 (joulukuu 1998), s. 405–439. DOI: 10.1023/A:1010035624696.

[58] T. Suzumura, T. Takase ja M. Tatsubori. ”Optimizing Web services performance by differential deserialization”. Teoksessa: *IEEE International Conference on Web Services (ICWS'05)*. 2005, 185–192 vol.1.

[59] J. Swaine, K. Tew, P. Dinda, R. B. Findler ja M. Flatt. ”Back to the Futures: Incremental Parallelization of Existing Sequential Runtime Systems”. *SIGPLAN Not.* 45.10 (lokakuu 2010), s. 583–597. ISSN: 0362-1340. DOI: 10.1145/1932682.1869507. URL: https://dl.acm.org/doi/10.1145/1932682.1869507.

[60] A. Tanimura ja H. Iwasaki. ”Integrating Lua into C for Embedding Lua Interpreters in a C Application”. Teoksessa: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. SAC '16. Pisa, Italy: Association for Computing Machinery, 2016, s. 1936–1943. ISBN: 9781450337397. DOI: 10.1145/2851613.2851747. URL: https://dl.acm.org/doi/10.1145/2851613.2851747.

[61] *TrinityCore issue #24356: Update Time Diff Unstable*. 31. heinäkuuta 2020. URL: https://github.com/TrinityCore/TrinityCore/issues/24356.

[62] *TrinityCore MMORPG Framework*. commit: fa8c0dd534723a573268. 13. maaliskuuta 2019. URL: https://github.com/TrinityCore/TrinityCore.

[63] T. Watanabe ja A. Yonezawa. ”Reflection in an Object-Oriented Concurrent Language”. Teoksessa: *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA '88. San Diego, California, USA: Association for Computing Machinery, 1988, s. 306–315. ISBN: 0897912845. DOI: 10.1145/62083.62111. URL: https://dl.acm.org/doi/10.1145/62083.62111.

[64] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon ja C. Wimmer. ”Self-Optimizing AST Interpreters”. Teoksessa: *Proceedings of the 8th Symposium on Dynamic Languages*. DLS ’12. Tucson, Arizona, USA: Association for Computing Machinery, 2012, s. 73–82. ISBN: 9781450315647. DOI: 10.1145/2384577.2384587. URL: https://dl.acm.org/doi/10.1145/2384577.2384587.

[65] Z. Yu, L. Song ja Y. Zhang. ”Fearless Concurrency? Understanding Concurrent Programming Safety in Real-World Rust Software”. *CoRR* abs/1902.01906 (2019). arXiv: 1902.01906. URL: http://arxiv.org/abs/1902.01906.