Jere Luomajoki

RUST JA ECS-ARKKITEHTUURIMALLI PELIALAN PALVELINYMPÄRISTÖSSÄ

RUST JA ECS-ARKKITEHTUURIMALLI PELIALAN PALVELINYMPÄRISTÖSSÄ

Jere Luomajoki
Opinnäytetyö
Syksy 2018
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, ohjelmistokehitys

Tekijä: Jere Luomajoki
Opinnäytetyön nimi: Rust ja ECS-arkkitehtuurimalli pelialan palvelinympäristössä
Työn ohjaajat: Pertti Heikkilä, Mikael Saarenpää
Työn valmistumislukukausi ja -vuosi: Syksy 2018
Sivumäärä: 26

Rust on tuore, erittäin potentiaalinen, ohjelmointikieli. Sen käyttötarkoituksia on syytä tutkia laajasti, ja tämä opinnäytetyö punnitsee sen tuomia etuja palvelinympäristöön. Lisäksi työssä tutkitaan, voisiko peliohjelmoinnissa käytettyä ECS-arkkitehtuuria hyödyntää palvelinympäristössä.

Rustin ja ECS:n toimivuutta palvelinympäristössä tarkastellaan kehittämällä järjestelmä, joka mallintaa palvelinohjelmaa, joka keskustelee peliä mallintavan asiakasohjelman kanssa. Työtä varten tehtiin laajaa taustatutkimusta käytetyistä tekniikoista ja niiden opiskeluun käytettiin runsaasti aikaa.

Lopputulokseksi todetaan, että Rustilla on suuri potentiaali toimia palvelinympäristössä, mutta tällä hetkellä saatavissa olevat resurssit eivät ole täysin kypsiä. Rustin käyttäminen tuotannossa on toistaiseksi riski, joskin lupaava sellainen. ECS puolestaan todetaan jossain määrin toimivaksi, mutta sen hyödyt muihin laajalti käytössä oleviin arkkitehtuuriratkaisuihin ovat joko olemattomia tai vähäisiä.

Asiasanat: Rust, palvelinympäristö, palvelinohjelma, Backend, peliala, ohjelmistoarkkitehtuuri, ohjelmointikielet.

ABSTRACT

Oulu University of Applied Sciences
Information technology, Software engineering

Author: Jere Luomajoki
Title of thesis: Rust and ECS-architecture in a game industry server environment
Supervisors: Pertti Heikkilä, Mikael Saarenpää
Term and year when the thesis was submitted: Fall 2018
Pages: 26

Rust is a potential fresh programming language. The usages of the language should be examined extensively, and this thesis investigates its usability in a server-side environment. In addition to Rust, this thesis evaluates the usability of ECS, an architectural pattern used in game development, in a server environment. The motivation for the thesis came from a game development company, Fingersoft ltd, which functions as the assigner of the thesis.

The technologies are evaluated by developing a mockup of a backend service using Rust with an ECS, which in turn listens to a client software which mocks some actions which a video game would send to its backend.

Conclusion is that Rust has great potential in server environment, but as of this moment available resources, such as libraries, are not fully matured. Therefore, using rust in production might include some risks, but is still a viable option. ECS in turn seems to be usable, but its benefits are minor, if not non-existent in comparison to traditional architectural patterns.

Keywords: Rust, server-side environment, backend, gaming industry, software architecture, programming languages

ALKULAUSE

Erityisesti haluan kiittää Fingersoftin Anttia, jonka idea oli olla yritykseen yhteydestä opinnäytetyön tilaamisesta, sekä Mikaelia, jolta idea työn aiheestat tuli. Erityisesti Rust-kieli osoittautui erittäin mielenkiintoiseksi ja potentiaaliseksi tulevaisuuden työkaluksi, jonka tutkimista aion jatkaa myös opinnäytetyön jälkeen.

Kiitokset kuuluvat myös opinnäytetyön ohjaavalle opettajalle, Pertti Heikkilälle, sekä tekstiasun tarkastaneelle Tuula Hopeavuorelle.

2.12.2018

Jere Luomajoki

SISÄLLYS

1 JOHDANTO

Rust on tuore ohjelmointikieli, joka on saanut suurta huomiota ohjelmistoyhteisössä. Stack Overflow'n, suuren ohjelmoijille suunnatun yhteisön, vuosittaisessa kyselyssä se on äänestetty kolmena peräkkäisenä vuotena rakastetuimmaksi ohjelmointikieleksi (1; 2; 3)

Entity Component System, ECS, on erityisesti peleissä käytetty ohjelmistoarkkitehtuuri, jota käytettäessä poiketaan huomattavasti yleisesti käytetystä olio-ohjelmoinnista. Tämä arkkitehtuuri antaa huomattavia etuja pelinkehityksessä, erityisesti suorituskyvyssä. (4.)

Tämän opinnäytetyön tarkoituksena on tutkia Rustin sekä Entity Component Systemin, ECS:n, soveltuvuutta palvelinympäristössä pelialalla. ECS on ollut pitkään käytössä pelien moottoreissa (4) ja Rust on nostamassa päätään erinomaisena vaihtoehtona pelinkehitykseen (5). Kumpikaan tekniikoista ei kuitenkaan ole laajalti käytössä palvelinympäristössä. Työn pääasiallisena tavoitteena on selvittää, kuinka helppoa uuden järjestelmän toteuttaminen on kyseisillä tekniikoilla ja mitä etuja tai haittoja siinä on nykyisiin järjestelmiin verrattuna.

Työn tilaajana toimii Fingersoft Oy. Opinnäytetyön tavoitteena on lisäksi toteuttaa tilaajalle esimerkkiprojekti, jota se voi käyttää arvioidakseen, kannattaako Rust ottaa käyttöön osaksi yrityksen järjestelmiä ja voisiko ECS:ää hyödyntää palvelinohjelmistossa.

2 RUST-OHJELMOINTIKIELI

Rust on Mozilla Researchista lähtöisin oleva, vuodesta 2006 asti kehityksessä ollut järjestelmäohjelmointikieli (6, The Rust Project -> Is this project controlled by Mozilla?). Järjestelmäohjelmointikielenä Rust on erittäin monipuolinen ja suorituskykyinen. Perinteisesti monissa muissa järjestelmäohjelmointiin käytetyissä kielissä kehittäjältä vaaditaan syvää ymmärrystä laiteläheisestä ohjelmoinnista, muistinhallinnasta sekä muista alhaisen tason erikoisuuksista (7, Foreword).

2.1 Rustin lyhyt historia

Rustin historia alkaa vuodesta 2006 Mozillan työntekijän Graydon Hoaren henkilökohtaisesta projektista. Vuonna 2009 Mozilla alkoi sponsoroimaan projektia (6, The Rust Project -> Is this project controlled by Mozilla?) ja kieli julkistettiin vuonna 2010. Rustin versio 1.0 julkistettiin toukokuussa 2015.

Ensimmäisinä vuosina Hoare kehitti Rustia yksin, ja tällöin kielen tavoitteiksi määriteltiin muun muassa turvallinen muistinhallinta, tyhjien tai epävarmojen muistiosoittimien puute, sekä sivuvaikutusten hallinta (8, The Personal Years). Suurin osa näistä tavoitteista on pysynyt, vaikka menetelmät niiden saavuttamiseksi ovat muuttuneet.

Kun Mozilla ryhtyi osalliseksi Rustin kehitystä, tiimin koko kasvoi tasaisesti, Hoaren pysyessä tiukasti johdossa. Kieli kehittyi ja muuttui tasaisesti, sekä Rust-yhteisöön muodostui kolme pääryhmää: entiset C++-osaajat, skriptauskieliosaajat sekä funktionaalisten kielten osaajat. (8, The Graydon Years, The Typesystem Years.)

Julkaisuvuonna 2015 Rustin ekosysteemiin kuului Cargo-paketinhallintajärjestelmä Crates.io-alustan kera. Kielen suurimmat käyttökohteet olivat pelinkehitys, käyttöjärjestelmät ja web-kehitys. (8, The Release Year.)

Ennen versiota 1.0 Rust kehittyi ja muuttui huomattavasti, mutta sen jälkeen kielen luvataan olevan vakaa. Vaikka kieli jatkaa kehittymistä, tänä päivänä kirjoitettujen ohjelmien pitäisi olla yhteensopivia tulevien versioiden kanssa. (6, Why has Rust changed so much over time?.)

2.2 Rustin esittely

Yksi Rustin ensisijaisista käyttötarkoituksista on järjestelmäohjelmointi. Tämä ilmenee vahvana muistinhallintana, turvallisena rinnakkaisuutena ja mahdollisuutena tehdä erittäin optimoitua koodia. Tämä ei kuitenkaan tarkoita sitä, että se olisi rajoitettu vain alhaisen tason järjestelmäohjelmointiin. Rust on täysin kykenevä esimerkiksi komentojanaohjelmiin, web-palvelinohjelmointiin ja moneen muuhun korkeammankin tason ohjelmointiin. (7, Foreword.)

2.2.1 Syntaksi

Kuvassa 1 on nähtävissä lyhyen Rust-ohjelman päätiedosto nimeltä *Main.rs*. Ohjelmaan kuuluu pääohjelman lisäksi ohjelman tietoja sisältävä *Cargo.toml*-tiedosto.

```
1   fn main() {
2       let five = 5;
3       let result = sqr(five);
4       println!("The square of five is {}", result);
5   }
6
7   fn sqr(number: i32) -> i32 {
8       number * number
9   }
10
```

KUVA 1. Esimerkki Rustin syntaksista

Funktiot määritellään avainsanalla *fn*, ja funktio nimeltä *main()* ajetaan automaattisesti. Ohjelmassa alustetaan ensiksi muuttuja nimeltä *five*, jonka arvoksi määritellään nimensä mukaisesti 5. Muuttujan alustus tehdään käyttämällä avainsanaa *let*. Kääntäjän täytyy saada tietää kaikkien muuttujien tyypit jo ennen kuin ohjelma ajetaan, mutta se osaa kuitenkin päätellä useimpien muuttujien tyypit. Tässä tapauksessa *five* saa tyypikseen 32-bittisen kokonaisluvun, joka esitetään avainsanalla *i32*, jossa *i* tarkoittaa kokonaislukua (engl. Integer) ja numero 32 tarkoittaa 32-bittistä. Ilman avainsanaa *mut* muuttujan määrittelyssä muuttujan arvo ei voi muuttua. (7, Common Programming Concepts -> Data Types.)

Muuttuja *result* saa arvonsa funktiosta *sqr*, joka ottaa vastaan 32-bittisen kokonaisluvun, ja palauttaa myös 32-bittisen muuttujan.

Funktio *sqr* ottaa vastaan parametrin, joka saa funktion sisällä nimen *number*. Parametri määritellään funktion nimen jälkeisissä suluissa niin, että ensin tulee funktion kehossa käytettävä nimi parametrille. Tämän nimen voi valita itse, mutta suositeltavaa on, että on kuvaava. Funktion palautusarvo määritetään käyttäen nuolta ja palautusarvon tyyppiä. (7, Common Programming Concepts -> How Functions Work.)

Funktion kehossa, joka rajataan aaltosulkeilla, voidaan suorittaa tavanomaista laskentaa ja ohjelmalogiikkaa. Esimerkkiohjelmassa kerrotaan annettu parametri itsellään. Huomattavaa esimerkissä on, että laskutoimituksen jälkeen ei ole puolipistettä, mikä tekee tuloksesta palautusarvon. Saman lopputuloksen saisi myös käyttämällä *return*-avainsanaa (kuva 2). (7, Common Programming Concepts -> How Functions Work.)

```
7   fn sqr(number: i32) -> i32 {
8       let return_value = number * number;
9       return return_value;
10  }
```

KUVA 2. Vaihtoehtoinen tapa palauttaa arvo funktiosta.

2.2.2 Muistinhallinta

Kuten aikaisemmin mainittu, yksi Rustin huomattavammista ominaisuuksista on sen muistinhallinta. Useimmat ohjelmointikielet käyttävät muistinhallintaan joko roskankeruuta tai käsin muistipaikkojen varaamista ja vapauttamista. Molemmissa on omat hyötynsä ja haittansa. Rust ei kuitenkaan käytä suoraan kumpaakaan, vaan käyttää uniikkia ominaisuutta, omistusmallia.

Omistusmallissa jokaisella arvolla on omistaja, ja yhdellä arvolla voi olla vain yksi omistaja kerrallaan. Kun omistaja menee pois ohjelman rajoista, arvon muisti-

paikka vapautetaan. Omistajuutta voi myös muuttaa, esimerkkinä funktio voi ottaa omistajuuden sille annetusta parametrista. (7, Understanding Ownership -> What Is Ownership?.)

2.2.3 Kääntäjä

Rustin kääntäjä on erittäin tiukka ja estää suuren osan vaikeasti huomattavista ohjelmavirheistä. Mikäli kääntäjä huomaa tällaisen virheen, se ei suostu kääntämään ohjelmaa, ennen kuin virhe on korjattu tai se käsitellään kunnollisesti. Käytännössä tämä tarkoittaa sitä, että kehitysprosessissa kääntäjää varoittaa ohjelmoijaa erittäin usein erilaisista virheistä, jotka täytyy käsitellä jollain tavalla ennen jatkamista. (7, Common Programming Concepts -> Variables and Mutability.)

2.2.4 Cargo-paketinhallinta

Cargo on Rustin paketinhallintajärjestelmä. Cargolla voidaan ladata yhteisön luomia paketteja ja ohjelmia Crates.io-pakettirekisteristä. Sen lisäksi Cargoa voidaan käyttää projektin aloittamiseen, tarkistamiseen, kääntämiseen, ajamiseen sekä julkaisemiseen aiemmin mainittuun Crates.io:hon. (7, Getting Started -> Hello, Cargo!.)

Vaikka Cargo ei ole pakollinen Rust-ohjelmia kehittäessä, se helpottaa prosessia huomattavasti. Komentojanakäskyllä *cargo new* voidaan luoda uusi projekti. Tämä luo kansiot projektille, pienen koodinpätkän sisältävän tiedoston sekä manifestin, joka sisältää tietoa projektista. (7, Getting Started -> Hello, Cargo!.)

Kirjoitetun koodin voi tarkistaa nopeasti virheiltä käskyllä *cargo check*. Tämä komento tarkastaa nopeasti koodin käännösvirheiltä, mutta ei kuitenkaan käännä tai aja koodia. Tämä on nopea tapa tarkistaa, onko koodissa esimerkiksi kirjoitusvirheitä tai muita yksinkertaisia virheitä. (7, Getting Started -> Hello, Cargo!.)

Cargoa voi myös kääntää ohjelman kääntämiseen ja ajamiseen. Kääntämiseen löytyy erilaisia asetuksia, joilla voi tasapainottaa ohjelman kääntämiseen ja ajamiseen käytettyä aikaa. Hitaampi kääntäminen tarkoittaa tehokkaampia optimointeja, mitkä vähentävät ohjelman ajoaikaa. (7, Getting Started -> Hello, Cargo.)

3 ENTITY COMPONENT SYSTEM - ARKKITEHTUURI

Entity Component System on erityisesti peliohjelmoinnissä käytetty ohjelmistoarkkitehtuuri, joka toimii vaihtoehtona oliopohjaiselle luokkahierarkialle, joka voi paisua vaikeasti ymmärrettäväksi (9). Tämän luvun tarkoituksena on esitellä ECS pääpiirteittäin ja myöhemmässä kappaleessa tutkitaan, kuinka ECS toimii peliohjelmoinnin ulkopuolella palvelinympäristössä Rust-ohjelmointikielellä. ECS:stä ei ole tarkkaa määritelmää ja teoriaa (4), vaan se voidaan tulkita eräänlaisena ohjenuorana jota voi soveltaa tarpeiden mukaisesti.

3.1 Entity

Entity eli entiteetti kuvastaa ainoastaan abstraktia kokonaisuutta ottamatta kantaa sen ominaisuuksiin tai toimintaan. Ohjelman toteutuksessa entiteetti ilmentyy usein mahdollisimman yksinkertaisesti, kuten kokonaislukuna (10). Entiteeteillä itsellään ei ole mitään toiminnallisuutta tai muuta tarkoitusta kuin erottaa itsensä muista entiteeteistä ja sisältää komponentteja. (4.)

3.2 Component

Component eli suomeksi komponentti, on lyhyesti sanottuna ominaisuus, jonka Entiteetti voi omistaa. Se voi kuvastaa esimerkiksi pelihahmon ulkonäköä, myytävän tuotteen hintaa tai vaikkapa koiran kykyä haukkua. Entiteetti siis saa kaikki ominaisuutensa komponenteista. Komponenttiin kuuluu lisäksi usein myös tietoa ominaisuuden yksityiskohdista. Esimerkiksi koiran haukkuminen voisi olla vaikkapa ääntely-komponentti, jonka sisällä on tieto ääntelyn tyypistä, jonka arvo on haukkuminen. Kissalla sama komponentti voisi saada arvon maukuminen. (4.)

3.3 System

Systems eli järjestelmät ovat ECS:n tapa hallita komponentteja. Artikkelissaan *Entity Systems are the future of MMOG development – Part 2* Adam West kuvailee järjestelmät seuraavanlaisiksi: sen sijaan, että dataa hallittaisiin komponenteissa tai olioissa olevilla metodeilla, kuten olio-ohjelmoinnissa, sitä hallitaan jatkuvasti ajossa olevilla järjestelmillä, jotka vaikuttavat suoraan komponentteihin

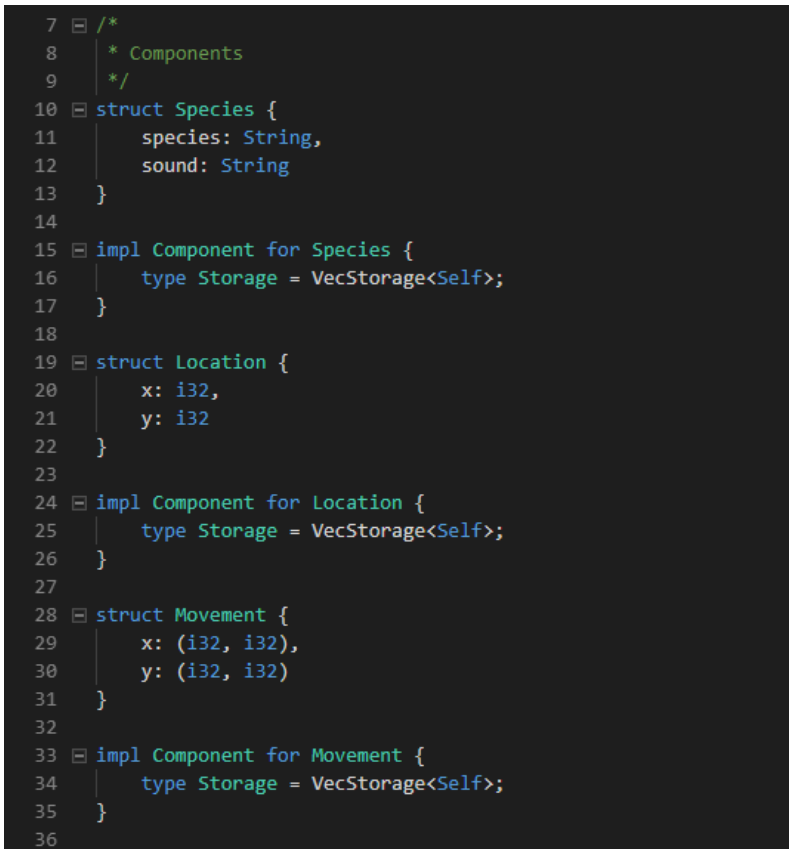
(10). Aikaisempaa esimerkkiä kissoista ja koirista käyttäen ääntelyjärjestelmää ajaessa kissa maukaisee ja koira haukahtaa.

3.4 Esimerkkiohjelma

Tässä kappaleessa esiteltävä ECS-esimerkkiohjelma on tehty käyttäen Rust-ohjelmointikieltä ja Specs-kirjastoa, joka tarjoaa työkalut ECS:n toteutukseen. Ohjelmassa määritellään ihminen, koira ja kissa, jotka osaavat esitellä itsensä ja liikkua. Ohjelman tarkoituksena on esitellä ECS:n perusperiaatteet.

Kuvassa 3 nähdään pienen esimerkkiohjelman komponenttien määrittely. Komponentit *Species*, *Location* ja *Movement* ovat yksinkertaisia tietorakenteita, jotka Specs-kirjastoa käyttäessä määritetään komponenteiksi käyttämällä Component-traitia. Traitit ovat Rustin ominaisuus, joka mahdollistaa yhtenäisten ominaisuuksien toteuttamisen eri tietorakenteissa (11, Traits). Component-traitin toteuttamiseksi täytyy määritellä tieto datan tallennuksesta. Tässä esimerkissä tallennukseen käytetään VecStoragea.

Speciesissä eli lajikomponentissa on tieto lajin nimestä ja ääntelystä. Locationissa eli sijaintikomponentissa on x- ja y-koordinaatti. Movementissa on tieto minimi- ja maksimirajoista, kuinka kauas entiteetti voi liikkua kerrallaan. Komponentit voisivat olla monimutkaisempia tai yksinkertaisempia ja niitä voisi olla huomattavasti enemmän.



```
/*
 * Components
 */
struct Species {
    species: String,
    sound: String
}

impl Component for Species {
    type Storage = VecStorage<Self>;
}

struct Location {
    x: i32,
    y: i32
}

impl Component for Location {
    type Storage = VecStorage<Self>;
}

struct Movement {
    x: (i32, i32),
    y: (i32, i32)
}

impl Component for Movement {
    type Storage = VecStorage<Self>;
}

```

KUVA 3. Komponenttien määrittely

Järjestelmien määrittelyt kuvassa 4 aloitetaan tyhjästä tietorakenteesta, jotka toteuttavat System-traitin. Specs-kirjaston System-traitin toteuttamista varten on määriteltävä SystemData, joka sisältää tiedon käytettävistä komponenteista, sekä run-funktio, joka ajetaan järjestelmää ajaessa. Esimerkissä SpeciesSystem, lajeista vastaava järjestelmä, ottaa vastaan Species-komponentin. LocationSystem, liikkumisesta vastaava järjestelmä, ottaa vastaan sekä Location-komponentin että Species-komponentin, jotta se osaa kertoa hieman enemmän tietoa liikkumisesta esimerkkiä varten. MovementSystem ottaa vastaan Movement-komponentin ja Location-komponentin, tällä kertaa kirjoitusoikeudella.

Esimerkin lajijärjestelmä kertoo yksinkertaisesti, mikä lajin nimi on ja kuinka se ääntelee. Aina kun se ajetaan, jokainen entity, jolla on lajikomponentti, kertoo tiedon itsestään.

Liikkumisjärjestelmä taas ottaa vastaan liikkumistiedon ja sijainnin, joka on määritetty muokattavaksi funktiolla WriteStorage ja avainsanalla *mut*. Järjestelmä arpoo uuden sijainnin liikkumistiedon mukaisesti jokaiselle entiteetille, jolla on komponentit Movement ja Location.

```
37  /*
38   * Systems
39   */
40  struct SpeciesSystem;
41  impl<'a> System<'a> for SpeciesSystem {
42      type SystemData = ReadStorage<'a, Species>;
43
44      fn run(&mut self, species: Self::SystemData) {
45          for species in species.join() {
46              println!("I'm a {}! {}", &species.species, &species.sound);
47          }
48      }
49  }
50
51  struct MovementSystem;
52  impl<'a> System<'a> for MovementSystem {
53      type SystemData = (
54              ReadStorage <'a, Movement>,
55              WriteStorage<'a, Location>
56          );
57
58      fn run(&mut self, (movement, mut location): Self::SystemData) {
59          for (movement, location) in (&movement, &mut location).join() {
60              let (x, y) = (movement.x, movement.y);
61              location.x += rand::thread_rng().gen_range(x.0, x.1);
62              location.y += rand::thread_rng().gen_range(y.0, y.1);
63          };
64      }
65  }
66
67  struct LocationSystem;
68  impl<'a> System<'a> for LocationSystem {
69      type SystemData = (
70          ReadStorage <'a, Location>,
71          ReadStorage <'a, Species>
72      );
73
74      fn run(&mut self, (location, species): Self::SystemData) {
75          for (location, species) in (&location, &species).join() {
76              let (species, x, y) = (&species.species, location.x, location.y);
77              println!("{} is now at x: {} y: {}", species, x, y);
78          }
79       }
80  }
81
```

KUVA 4. Järjestelmien määrittely

Kuvassa 5 esimerkkijärjestelmään luodaan kolme entiteettiä. Entiteettejä ei nimetä tai erotella muuten kuin komponenttien avulla. Kaikille entiteeteillä on Species- ja Location-komponentti, ja Movement-komponentti on annettu kaikille,

paitsi entiteetille, jonka lajiksi on määrätty kissa. Tämä tarkoittaa sitä, että liikkumisjärjestelmä käsittelee ainoastaan koira- ja ihmisentiteetin sijaintikomponentteja.

```
91  /*
92   * Entities
93   */
94  world.create_entity()
95      .with(Species{
96          species: String::from("Dog"),
97          sound: String::from("Woof!")
98      })
99      .with(Location{ x: 10, y: 15 })
100     .with(Movement{ x: (-5, 5), y: (-5, 5)})
101     .build();
102
103 world.create_entity()
104     .with(Species{
105         species: String::from("Cat"),
106         sound: String::from("Meow!")
107     })
108     .with(Location{ x: 20, y: 25 })
109     .build();
110
111 world.create_entity()
112     .with(Species{
113         species: String::from("Human"),
114         sound: String::from("Hello there!")
115     })
116     .with(Location{ x: 30, y: 35 })
117     .with(Movement{ x: (-10, 10), y: (-10, 10)})
118     .build();
```

KUVA 5. Entiteettien määrittelyt

Entiteettien, komponenttien ja systeemien lisäksi esimerkkiohjelmaan täytyy määrittää World eli maailma, jonne komponentit, entiteetit ja muut resurssit rekisteröidään. Loput ohjelmasta nähdään kuvassa 6.

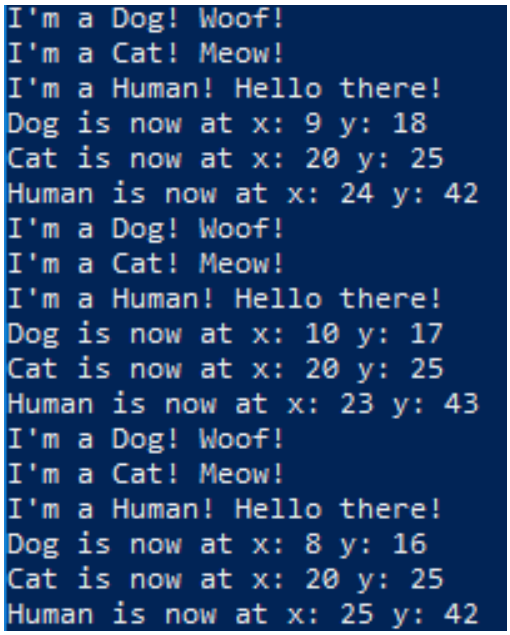
Dispatcher hoitaa ohjelman ajamisen asynkronisesti. Siihen määritellään ajettavat järjestelmät ja riippuvuudet. Esimerkkiohjelmassa SpeciesSystem- ja MovementSystem-järjestelmissä ei ole riippuvuuksia, joten ne ajetaan yhtäaikaisesti.

LocationSystem riippuu MovementSystemistä, ja se ajetaan vasta, kun MovementSystem on lopettanut. Ohjelman viimeisillä rivillä Dispatcheria ajetaan toistuvasti yhden sekunnin väliajoin, kunnes ohjelma keskeytetään väkisin.

```
82  /*
83   * Main program
84   */
85  fn main() {
86      /*
87       * World initialisation
88       */
89      let mut world = World::new();
90      world.register::<Species>();
91      world.register::<Location>();
92      world.register::<Movement>();
93
94      /*
95       * Entities
96       */
97      world.create_entity()···
106     world.create_entity()···
114     world.create_entity()···
123     /*
124      * Dispatcher
125      */
126     let mut dispatcher = DispatcherBuilder::new()
127         .with(SpeciesSystem, "species_system", &[])
128         .with(MovementSystem, "movement_system", &[])
129         .with(LocationSystem, "location_system", &["movement_system"])
130         .build();
131
132     loop {
133         dispatcher.dispatch(&world.res);
134         std::thread::sleep(time::Duration::from_millis(1000));
135     }
136 }
```

KUVA 6. Loput esimerkkiohjelmasta

Kuten kuvassa 7 nähdään, jokainen entiteetti kertoo ensin, mitä lajia ne ovat, ja päästää äänen. Sen jälkeen jokainen entiteetti kertoo, missä ne tällä hetkellä ovat. Koira ja ihminen liikkuvat, sillä niille annettiin Movement-komponentti, ja kissa pysyy paikallaan.



KUVA 7. Esimerkki ohjelman ajosta

4 ESIMERKKIJÄRJESTELMÄ

Esimerkkijärjestelmä rakennettiin Rustilla käyttäen Specs-kirjastoa, jolla toteutettiin ECS. Järjestelmä mallintaa asiakasohjelman kanssa keskustelevaa palvelinohjelmaa, joka ottaa vastaan komentoja, jotka se tulkitsee ja toteuttaa. Sen pääasiallinen tarkoitus on todistaa Rustin ja ECS:n toimivuutta palvelinympäristössä. Järjestelmän tueksi tilaaja tarjosi asiakasohjelman, joka keskustelee palvelinohjelman kanssa käyttäen TCP-yhteyden yli lähetettäviä viestejä.

4.1 Asiakasohjelma

Tilaajan tarjoama asiakasohjelma on yksinkertainen ohjelma, joka muokattiin simuloimaan peliä. Ohjelma ottaa vastaan yksinkertaisia komentoja, kuten liikkuminen ja viestiminen, pakkaa ne BSON-formaattiin ja lähettää ne palvelinohjelmaan. Myös palvelimelta tulevia viestejä pystytään tulostamaan.

Asiakasohjelman lähettämä data sisältää toimenpiteen tunnisteen, viestin järjestysnumeron sekä varsinaisen sisällön. Eri toimenpiteitä ovat *”Hello World”*, johon palvelin vastaa, liikkumisen ja puhuminen simulointi sekä tunnistautuminen.

4.2 Palvelinohjelma

Palvelinohjelma, eli varsinainen esimerkkijärjestelmä, on komentojanassa ajettava ohjelma, joka vastaa ainoastaan asiakasohjelmalta tuleviin käskyihin.

Palvelinohjelma on toteutettu ECS-arkkitehtuuria noudattaen ja sen toiminta on parhaiten selitettävissä purkamalla ohjelma komponentteihin ja järjestelmiin.

4.2.1 Komponentit

ECS:n mukaisesti ohjelmassa tarvittava tieto on jaettu eri komponentteihin, jotka yhdistetään toisiinsa entiteeteissä. Järjestelmässä luodaan yhtä asiakasohjelmaa varten yksi entiteetti.

TCP-kuuntelija

Jo ennen asiakasohjelman käynnistymistä luodaan valmiiksi kuuntelija, joka avaa portin asiakasohjelmaa varten. Komponentissa on Rustin *std::net*-kirjastosta löytyvä TcpListener, joka kuuntelee sille ennalta määrättyä porttia sisääntulevaa yhteyttä varten. Lisäksi komponentti sisältää tiedon siitä, onko yhteys luotu tai katkennut.

TCP-yhteys

TCP-yhteys luodaan erilliseen komponenttiin, jotta yhteyden kuunteleminen voidaan suorittaa eri järjestelmässä kuin yhteyden luominen. Komponentti sisältää ainoastaan varsinaiden yhteyden asiakasohjelmaan.

BSON-viesti

BSON-viestikomponenttiin luetaan asiakasohjelmalta tuleva binääridata. Komponenttiin myös merkataan tieto siitä, onko se purettu.

Dekoodattu data

Asiakasohjelmalta luettu data puretaan erilliseen komponenttiin, joka sisältää puretun viestin sekä tiedon siitä, onko viesti tallennettu tai käsitelty.

Pelaaja

Pelaaja-komponenttiin sisältyy ainoastaan pelaajan, eli asiakasohjelman, kutsumanimi. Mikäli ohjelmaa kehitettäisiin eteenpäin, komponenttiin lisättäisiin muita arvoja.

Sijainti

Asiakasohjelman ohjaaman pelaajan sijainti tallennetaan myös omaan komponenttiinsa. Sijainti sisältää x- ja y-koordinaatit.

Tallennettava

Viimeinen komponentti ei sisällä mitään tietoa, vaan toimii merkkinä, että entiteetti tulee tallentaa tietokantaan.

4.2.2 Järjestelmät

Kaikki ohjelman logiikka ja toiminnallisuus on jaettu järjestelmiin, jotka vastaavat tietyistä komponenteista. Toisin kuin ideaalisessa ECS:ssä, lähes kaikki järjestelmät käyttävät useita komponentteja tavalla tai toisella, ja useissa järjestelmissä on päällekkäisyyksiä.

TCP-yhteyden luoja

TCP-yhteyden luomiseen vaaditaan TCP-kuuntelukomponentti. Jokaisella ajolla järjestelmä katsoo, löytyykö kyseiselle kuuntelijalle yhteyttä, ja jos löytyy, se luo samaan entiteettiin uuden TCP-yhteyskomponentin. Lisäksi järjestelmä luo uuden entiteetin ja TCP -kuuntelijan.

TCP-yhteyden kuuntelija

Kyseinen järjestelmä lukee yhteyttä, ja mikäli siellä on jotain, luo entiteettiin uuden bson-komponentin yhteydestä saadusta datasta. Mikäli yhteys katkeaa, se poistaa TCP-yhteyskomponentin entiteetistä ja merkitsee TCP-kuuntelijaan tiedon yhteyden katkeamisesta, jotta se voidaan aukaista uudelleen.

BSON-dekooderi

BSON-datan dekoodaus suoritetaan omasa järjestelmässään. Dekoodatusta datasta luodaan uusi komponentti ja merkataan raaka data puretuksi.

Työjärjestelmä

Työjärjestelmä lukee dekoodatut viestit ja päättää, mitä tehdä. Esimerkiksi kun luetaan viestiä, jossa on käsky liikkua, muokataan entiteetin sijaintikomponenttia.

Tallennusjärjestelmä

Kun entiteetti on merkitty tallennettavaksi lisäämällä siihen tallennuskomponentti, tarvittavat tiedot tallennetaan MongoDB-tietokantaan.

4.3 Esimerkkijärjestelmän rakentaminen

Prosessi esimerkkijärjestelmän rakentamiseen oli suoraviivainen. Kun yleiskuva järjestelmästä oli suunniteltu, komponenttien suunnittelu oli helppoa. Kun komponentit olivat selvänä, järjestelmien suunnittelu ei myöskään vaatinut paljon aikaa. Koska Rust-ohjelmien kirjoittamisesta ei ollut syvää kokemusta, sen kääntäjää vastaan joutui ikään kuin taistelemaan ja ohjelman osia joutui suunnittelemaan uudestaan alusta asti useaan otteeseen. Tämä on kuitenkin yleistä aloittelijoiden keskuudessa (12). Kääntäjän luonteen takia kuitenkin virheet huomasi nopeasti ja ne olivat korjattavissa aikaisessa vaiheessa.

ECS tuki projektin iteratiivista kehittämistä. Järjestelmät eivät olleet suoraan riippuvaisia toisistaan, joten niiden kehittäminen yksi kerrallaan oli erittäin sujuvaa.

4.4 Esimerkkijärjestelmän ongelmat

Suurin haaste projektin kehitysvaiheessa oli ehdottomasti kokemuksen sekä lähdemateriaalin puute. Vaikka Rustille löytyy monipuolisesti erinomaisia oppaita, ne ovat usein suppeita ja suuntaa antavia. Lisäksi kielen nopean kehittymisen takia suuressa osassa oppaista on vanhentunutta tietoa.

Myös valmiin ECS-kirjaston käyttäminen aiheutti haasteita. Vaikka Specs-kirjasto tukee järjestelmien asynkroonista ja rinnakkaista ajamista, jokainen järjestelmä täytyy ajaa loppuun, ennen kuin uuden kierroksen voi aloittaa. Esimerkiksi ajoajaltaan pitempien järjestelmien toteutus osoittautui mahdottomaksi, sillä muiden järjestelmien täytyi odottaa, että kaikki järjestelmät olivat suoritettu, ennen kuin järjestelmät voitiin käynnistää uudestaan.

4.5 Esimerkkijärjestelmän hyödyt

Esimerkkijärjestelmää katselmoidessa Rustin hyödyt tulevat ilmi hyvin nopeasti. Kehitysvaiheessa Rustin muistinhallinta pakottaa ohjelmoijan kirjoittamaan ohjelman niin, että jokainen mahdollinen virhetilanne täytyy käsitellä tavalla tai toisella. Tämä lisää hieman kehitykseen menevää aikaa, ennen kuin mitään toiminnallisuutta voidaan testata, esimerkkijärjestelmän tapauksessa ohjelman ensimmäinen iteraatio kuitenkin toimi heti, kun se kääntyi.

Kehitysvaiheessa myös Cargo oli suureksi avuksi. Komentojanakäskyllä *cargo check* oli kätevä tarkistaa mahdolliset ongelmat nopeasti ennen kääntämistä. Sen lisäksi kääntäjä antoi huomautuksia käyttämättömistä muuttujista sekä muista suorituskykyyn negatiivisesti vaikuttavista asioista. Käytettyyn kehitysympäristöön, Microsoft Visual Studio Codeen, oli myös mahdollista saada Rust Language Server ja rustfmt-tuki, jotka antoivat huomautukset sekä tyylitys- että käännösvirheistä suoraan kehitysympäristöön.

ECS pakotti suunnittelemaan erittäin paljon etukäteen. Komponentit muuttuivat hyvin vähän ohjelman iteroinnin aikana, eikä olemassa olevien funktioiden tai rakenteiden muokkaamiseen tai korjaamiseen mennyt juurikaan aikaa.

Palvelinympäristöön ajatellen ECS tuo erittäin hyviä mahdollisuuksia erilaisten asynkroonisuudesta johtuvien ongelmatilanteiden välttämiseen. Perinteisessä palvelinohjelmistossa erityisesti tietokantakyselyt johtavat tilanteisiin, joissa kaksi tai useampi eri kyselyä lukitsevat resursseja, joita muut kyselyt tarvitsisivat. Tästä syntyy tilanne, jossa yksikään kysely ei kykene jatkamaan tai vapauttamaan käytössä olevia resursseja. ECS:ää käyttäessä kyselyt voidaan kuitenkin joko suorittaa synkronisesti yhdessä järjestelmässä tai suunnitella järjestelmät niin, ettei eri järjestelmät aiheuta toisilleen mainittuja lukkotilanteita. Myös Rust itsessään pyrkii välttämään tämän kaltaiset tilanteet muistinhallinnallaan (13, Concurrency -> Races).

Valitettavasti suorituskykyvertailu ei mahtunut käytettävissä olleisiin aikamääreisiin, mutta pintapuolisella tarkastelulla ohjelman suorituskyky oli erittäin tyydyttävä. Jatkokehitystä varten olisi syytä tehdä prototyyppijärjestelmät Rustilla ilman ECS:ää, kilpailevalla kielellä käyttäen ECS:ää sekä kilpailevalla kielellä ilman ECS:ää. Vertailukohtina olisi syytä käyttää suorituskykyä, kehittämiseen käytettyä aikaa sekä vakautta. Myös saman tekniikan ja arkkitehtuurin käyttö yhdessä kehitettävien järjestelmien välillä antaa lisäarvoa.

5 YHTEENVETO

Työn tavoitteena oli tutkia Rustin ja Entity Component System-arkkitehtuurimallin soveltumista palvelinympäristöön ja tavoitteena oli sekä perehtyä tekniikoihin että kehittää esimerkkijärjestelmä käyttäen kyseisiä tekniikoita.

Kaiken kaikkiaan Rust ja Entity Component System ovat erittäin yhteensopivia. alhaisen tason takia Rust toimii hyvänä tyhjänä tauluna, jonka päälle on hyvä rakentaa monipuolisia arkkitehtuurillisia ratkaisuja.

Palvelinympäristössä Rust ja ECS toimivat kohtuullisesti yhteen, mutta ECS:n hyötyjä verrattuna vakiintuneisiimpaan arkkitehtuurimalleihin on syytä punnita lisää, ennen kuin sillä alkaa rakentamaan suuria tuotantoratkaisuja.

Rust itsessään on käyttövalmis pieniin ja keskikokoisiin järjestelmiin, mutta suurta järjestelmää rakennettaessa perinteisemmät teknologiat ovat todennäköisesti turvallisempia. Mikäli on epävarma siitä, onko Rust sopiva kieli tiettyyn projektiin, Steve Klabnik antaa esityksessään Rust in Production (14, 8:07) vinkiksi katsoa, löytyykö tarvittavia kirjastoja Crates.io:sta. Suurin hyöty Rustista on suorituskyvyssä, mikäli kielen ominaisuuksia pääsee hyödyntämään, sekä sen turvallisuudessa. Haittapuolina on jyrkähkö oppimiskäyrä, joka hidastaa kielen omaksumisessa, sekä se, että suurin osa kirjastoista ovat toistaiseksi vasta kehitysvaiheessa.

LÄHTEET

1. Most Loved, Dreaded, and Wanted. 2018. Developer Survey Results 2018. StackOverflow. Saatavissa: https://insights.stackoverflow.com/survey/2018/#most-loved-dreaded-and-wanted Hakupäivä 10.9.2018.

2. Most Loved, Dreaded, and Wanted. 2017. Developer Survey Results 2017. StackOverflow. Saatavissa: https://insights.stackoverflow.com/survey/2017#most-loved-dreaded-and-wanted Hakupäivä 10.9.2018.

3. Most Loved, Dreaded, and Wanted. 2016. Developer Survey Results 2016. StackOverflow. Saatavissa: https://insights.stackoverflow.com/survey/2016#technology-most-loved-dreaded-and-wanted Hakupäivä 10.9.2018.

4. Martin, Adam 2007. Entity Systems are the future of MMOG development – Part 2. Saatavissa http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/. Hakupäivä 5.10.2018.

5. Are we game yet? Saatavissa http://arewegameyet.com/. Hakupäivä 5.11.2018.

6. Frequently Asked Questions. Rust. Saatavissa: https://www.rust-lang.org/en-US/faq.html#is-this-project-controlled-by-mozilla Hakupäivä 10.9.2018.

7. Matsakis, Nicholas, Turon, Aaron. The Rust Programming Language 2018 Edition. Saatavissa: https://doc.rust-lang.org/book/2018-edition/. Hakupäivä 10.9.2018.

8. Klabnik, Steve 2016. History of Rust. Diaesitys. Ohjaus nuolinäppäimillä. Saatavissa http://steveklabnik.github.io/history-of-rust/. Hakupäivä 20.9.2018.

9. West, Mick 2007. Evolve Your Hierarchy. Saatavissa: http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/. Hakupäivä 5.10.2018.

10. ES Terminology. Entity Systems Wiki. Saatavissa http://entity-systems-wiki.t-machine.org/es-terminology. Hakupäivä 5.10.2018.

11. Rust By Example. Saatavissa https://doc.rust-lang.org/rust-by-example. Hakupäivä 9.10.2018

12. De Coster, Mathieu 2017. Fighting the Borrow Checker. Saatavissa https://m-decoster.github.io/2017/01/16/fighting-borrowchk/. Hakupäivä 6.11.2018.

13. The Rustonomicon. Saatavissa https://doc.rust-lang.org/nomicon/ Hakupäivä 8.11.2018.

14. Klabnik, Steve 2017. Rust In Production, Øredev Conference 2017. Saatavissa https://vimeo.com/242056778. Hakupäivä 28.11.2018