Tuukka Pasanen

ETHEREUM-VIRTUAALIKONEELLA SOLMITTU ÄLYKÄS SOPIMUS TUOTANNON SEURANNASSA

Ethereum-virtuaalikoneella solmittu älykäs sopimus tuotannon seurannassa

Tuukka Pasanen
Opinnäytetyö
Syksy 2018
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikka, Tietotekniikan tutkinto-ohjelma

Tekijä(t): Tuukka Pasanen
Opinnäytetyön nimi: Ethereum-virtuaalikoneella solmittu älykäs sopimus tuotannon seurannassa
Työn ohjaaja(t): Jukka Jauhiainen ja Veijo Väisänen
Työn valmistumislukukausi ja -vuosi: 2018
Sivumäärä: 29 + 1 liitettä

Tässä opinnäytetyössä Ethereum-virtuaalikoneella solmittu älykäs sopimus on esimerkki tavasta tehdä älykkäitä sopimuksia rajatussa yksityisessä verkossa tuotannon ohjauksen kehittämiseksi pohjamaanrakennusalalla.

Opinnäytetyössä Solidity-kielen 0.4-versiolla tehtiin esimerkkejä älykkäistä sopimuksista (Smart Contract). Toteutus lähtee perusteista dokumentoiden miten Go-kielisellä Geth-komentoriviohjelmalla saa Solidity-koodin louhittua eli liitettyä lohkoketjuun sekä louhitun sopimuksen suoritettua.

Lopputuloksena on dokumentoitu tapa tehdä yksityinen lohkoketju Ethereum-alustalle ja liittää tähän lohkoketjuun älykkäitä sopimuksia. Lohkoketju ei tämän opinnäytetyön testausten perusteelta vaikuta kovin käyttökelpoiselta tekniikalta tuotannon ohjaukseen ilman integraatiota olemassa oleviin tekniikoihin. Lohkoketjutekniikan muuttuessa arkipäiväisemmäksi ja sen suosion kasvaessa lohkoketjujen käyttö voi yleistyä aukottoman tunnistuksen vaatimissa järjestelmissä.

Asiasanat: lohkoketju, toiminnanohjaus, ethereum

ABSTRACT

Oulu University of Applied Sciences
Computer Science Degree programme

Author(s): Tuukka Pasanen
Title of thesis: Ethereum Smart Contracts in Production Management
Supervisor(s): Jukka Jauhiainen ja Veijo Väisänen
Term and year when the thesis was submitted: 2018
Pages: 29 + 1 appendices

This thesis presents an example of using Ethereum blockchain technology to create smart contracts for developing production management in the deep foundation construction business.

Solidity version 0.4 is used to create examples of smart contracts. The objective is to document the creation of a private blockchain with a Geth-console application that runs the Go programming language, and to show how one can mine a smart contract to be part of a private blockchain.

The end result is a documented method for creating a private blockchain on the Ethereum platform and attaching smart contracts to this blockchain. In its current form Ethereum blockchain technology does not appear to be ready for use in production management without integration into existing technologies. However, as blockchain technology develops and grows in popularity, it may become a viable option for uses requiring seamless user authentication and a distributed database.

Keywords: blockchain, production management, ethereum

SISÄLLYS

1 JOHDANTO

Tämä opinnäytetyö käsittelee lohkoketjujen käyttöä tuotannon seurannassa. Opinnäytetyössä selvitettiin olemassa olevalle mutta anonyymiksi jäävälle rakennusalan yritykselle, voidaanko Ethereum-alustan lohkoketjutekniikkaa soveltaa suuren komponenttimäärän kirjanpidon toteutuksena. Lohkoketjun ajatus hajautetusta hallinnosta yhdistettynä saumattomaan osapuolien tunnistukseen mahdollistaa aivan uusia tapoja toimia tuotannon seurantaan.

Työssä pyrittiin yksinkertaisen esimerkin avulla toteamaan, onko Ethereum-alusta tarpeeksi vakaa, että sen päälle voitaisiin rakentaa tekniikkaa, jonka, ottaen huomioon rakennusalan hitaan muutoksen tulisi olla käytettävissä vuosikymmeniä. Lohkoketjun ja etenkin Ethereum-alustan ollessa alkumetreillä ei ole mitään takeita tekniikan pysyvyydestä. On myös mahdollista, että se korvautuu jollain sitä seuraavalla tekniikalla.

Työ jakaantuu kolmeen osaan. Ensimmäiseksi tutustutaan teoreettisesti lohkoketjuihin. Toinen osa on oman täysin yksityisen Ethereum-verkon luominen. Tämä siksi, että yksityisen verkon sisäisiä asioita on helpompi hallinnoida kuin vaikkapa julkisesta Ethereum-lohkoketjusta lohkotun oman lohkon asioita. Tähän ketjuun vaikuttavat kaikki julkisen Ethereum-lohkoketjun lohkon säännöt ja rajoitukset.

Tästä aiheesta ei ole paljoakaan julkaisuja kirjallisuudessa ja verkko luotiin Go-Ethereumin (Geth) dokumentaation *Setting up private network or local cluster* (Lange 2017) sekä blogikirjoituksen pohjalta *The Geth's saga: setting up Ethereum private network on Windows* (Gueiros 2017). Molemmat ovat vajavaisia tietolähteitä ja vaativat kokeiluja että olivat hyödyllisiä.

Kolmas osa on Solidity-kielellä tehdyn älykkään sopimuksen yhdistäminen omaan yksityiseen verkkoon ja sen kutsuminen. Onneksi julkaisuja löytyi normaalista älykkäiden sopimusten käytöstä, joten yhdistämällä tietoa tekstistä *Testing Smart Contracts Locally using Geth* (Arvanaghi 2018) sekä hyödyntämällä vielä julkaisematonta kirjaa *Mastering Ethereum* (Antonopoulos & Wood 2018a), tämä onnistui.

2 LOHKOKETJUT

Lohkoketjun ensimmäinen tuleminen, bitcoin, on selkeästi yhteen tarpeeseen synnytetty lohkoketju – säätelemättömän rahatalouden ylläpitoon. Idea bitcoinin takana on, että siihen kuuluvat tietokoneet ja käyttäjät tekevät toistensa kanssa kauppaa bitcoineja käyttäen (Franco 2014). Lohkoketjutekniikan perusteina toimiva tutkielma bitcoinista julkaistiin vuonna 2007 (BitcoinWebHosting.net 2018).

2.1 Bitcoin-lohkoketju

Bitcoinin ja koko lohkoketjutekniikan perustana pidetään tätä salaperäisen Satoshi Nakamoton verkkoon postittamaa dokumenttia *Bitcoin: A Peer-to-Peer Electronic Cash System*. Siinä kuvataan keskuspankeista riippumattoman bitcoin-kryptovaluutan toiminta tietoverkossa. Dokumentissa kuvataan, miten verkossa kuka tahansa voi luoda itselleen identiteetin ja alkaa tehdä kryptovaluutan siirtoja käyttäen Bitcoin-lohkoketjutekniikkaa (Nakamoto 2008).

Lohkoketju on hajautettu tietokanta ja viestinvälittäjä. Sana lohkoketju tulee siitä, että kaikista tietokannassa tapahtuvista transaktiosta muodostuu lohko edellisten päälle. Nämä tietokannan transaktiot ovat Bitcoinin tapauksessa kryptovaluutan siirtoja käyttäjältä toiselle (Meunier 2016). Välittäjän lohkoketjusta tekee se, että kuka tahansa voi liittyä eli ladata lohkoketjun, eli hajautetun tietokannan (Urban 2018). Bitcoin-tutkielmassa mainitaan lompakko. Se toimii identiteettinä, jota kukaan ei voi kahdentaa lohkoketjussa. Lompakkoa voidaan pitää käyttäjätunnuksena, jota ei voi väärentää (Nakamoto 2008). Huomionarvoista on, että toinen lompakon omistaja ei voi tietää, kuka toinen on, ellei hän paljasta oikeaa identiteettiään.

Lompakko on kaksiosainen salausavain, josta lompakon omistaja pitää itsellään avaimen salaisen osan ja jakaa avaimen julkista osaa lohkoketjussa (Nakamoto 2008). Salaisella avaimen osalla käyttäjä allekirjoittaa kaikki tekemänsä transaktiot, jotka voidaan todentaa kenen tahansa toimesta tekijän julkisella avaimella (Nakamoto 2008). Kryptovaluutan siirrot tehdään julkisen avaimen tarkastussummalle, joka toimii lompakon ilmentymänä Bitcoin-vertaisverkossa.

Kun kaksi lompakkoa haluaa vaihtaa bitcoin-kryptovaluuttaa keskenään, se tapahtuu erityisen prosessin avulla. Tämä tekee kryptovaluutta bitcoinista paremman kuin edeltäjänsä, sillä Bitcoin ratkaisee niin kutsutun kaksoismaksamisen ongelman (Nakamoto 2008).

Kaksoismaksaminen tulee ongelmaksi, kun joku oivaltaa voivansa käyttää hyvin lyhyen aikaikkunan sisällä saman valuutan kahteen tai useampaan kertaan. Bitcoin ratkaisee tämän ongelman louhinnalla (Khatwani 2018). Louhinta tarkoittaa sitä, että kun ketjuun halutaan liittää uusi transaktio, se lähetetään louhijoiksi ilmoittautuville tietokoneille (Nakamoto 2008). Ne alkavat transaktion saavuttua tehdä monimutkaisia laskutoimituksia, joilla ei ole tarkoitusta ja jotka kuormittavat louhivaa tietokonetta suhteettomasti. Nämä laskutoimitukset tehdään kaksoismaksamisen estämiseksi.

Louhija, joka ensimmäisenä saa laskennan valmiiksi julkaisee tuloksen, eli uuden lohkoketjun lohkon, ja se liitetään osaksi lohkoketjua (Nakamoto 2008). Esimerkiksi Bitcoin-vertaisverkossa louhinta kestää kymmenisen minuuttia. Ennen louhinnan loppua samalla rahalla voi maksaa niin monta kertaa, kuin haluaa, mutta kun joku louhijoista ensimmäisenä ilmoittaa, että valuutta on käytetty, estää se kyseessä olevan kryptovaluutan uudelleen käytön. Tämän jälkeen valuutta sijaitsee toisessa lompakossa, eikä näin ollen enää voi tulla osaksi toista kryptovaluutan transaktiota alkuperäiseltä omistajaltaan. Louhinnan suorittanut kone saa pienen määrän bitcoineja palkkioksi. Louhinnan jälkeen kaikki osapuolet lataavat louhitun transaktion osaksi omaa lohkoketjuaan käyttäen konsensus säännöstöä (Nakamoto 2008).

Aluksi lohkoketjut olivat marginaalinen ilmiö, mutta suosio nousi vuonna 2017 (Reinikainen 2017). Vaikka lohkoketju mielletään usein sidotuksi bitcoiniin ja sen suosioon, on sillä muitakin käyttökohteita. Tästä yksi esimerkki on kuljetusyhtiö MAERSKin ja IBM:n yhteinen projekti laivaston seurantaan, joka heidän mukaansa tuotti 40 %:n säästöt (Tivi 2017) sekä Nasdaqin kokeilut osakekaupasta IBM:n lohkoketjutekniikalla (Bajpai 2017). Näiden isojen lohkoketjutoteutuksen perusteella tekniikka alkaa näyttää potentiaalinsa.

Saadessaan suosiota Bitcoin synnytti muitakin lohkoketjutekniikkaa hyödyntäviä projekteja kuten Etherumin, jotka pyrkivät ratkaisemaan Bitcoinin puutteita tai toteuttamaan siinä puuttuvia ominaisuuksia.

2.2 Ethereum-alusta

Ethereum-alusta on Ethereum-yhteisön kehittämä lohkoketjusovellusten tekemiseen tarkoitettu ohjelmisto. Ethereum-alustan pohjana on ETH-lohkoketjuvaluutta, mutta silti valuutta ei varsinaisesti ole Ethereumissa pääasia. Ethereum-alusta pyrkii toteuttamaan lohkoketjun perustan, jonka Chris Dannen määrittelee seuraavasti: Lohkoketjun tärkeimmät osat ovat vertaisverkko, asymetrinen salaus ja kryptografisen pitävä tiiviste (Dannen 2017). Yksi päätavoitteista on mahdollistaa yksilöiden välinen luottamukseen perustuva transaktioiden suorittaminen (Wood 2018).

Ethereum pyrkii olemaan alusta, jolla ohjelmoija voi kehittää omia älykkäitä sopimuksiaan. Ethereum-alustan ytimenä on määritelty 256-bittinen virtuaalikonetietokone, eräänlainen globaalitietokone. Sen transaktiot ovat paikallisia jokaisella koneella, mutta ne ovat samalla kaikkien verkon koneiden näkyvillä (Dannen 2017). Ethereum-alustassa on pyritty luomaan lohkoketju ja sen päällä pyörivä tilakone sekä virtuaalikone, joka suorittaa ohjelmat (Dannen 2017). Ethereum-virtuaalikone on Turing-täydellinen. Se tarkoittaa, että sillä voi suorittaa mitä tahansa laskennallisia ongelmia ja niiden lopputulos on sama kuin millä tahansa muulla Turing-täydellisellä tietokoneella (McGrath 2016).

Ajatukset Ethereum-alustan takana ovat olleet

- yksinkertaisuus, joka tarkoittaa, että koneiden välisen kommunikaatioon käytettävän protokollan tulee olla yksikertaisin mahdollinen
- universaalius, joka tarkoittaa, että Ethereumissa ei ole ominaisuuksia vaan se tarjoaa Turing-täydellisen kielen niiden tekemiseen
- modulaarisuus, jonka tarkoittaa, että Ethereum-protokolla tulee olla laajennettavissa moduuleilla ja lisäosilla
- ketteryys, joka merkitsee, että protokollaa voidaan muuttaa tarpeen mukaan.

- syrjinnän estäminen, joka tarkoittaa, että Ethereum-protokollan tekijät eivät voi rajoittaa, mihin Ethereum-alustaa käytetään (Ray 2018a).

Ethereum-alustalla voi tehdä älykkäitä sopimuksia, jotka siis eivät ole sopimuksia vaan ohjelmia, joita ajetaan Ethereum-virtuaalikoneessa. Älykkäiden sopimusten kehittämiseen on kehitetty Solidity-kieli.

Lohkoketjuun liittyvä tiedonsiirtotapa eli protokolla määrittelee, miten tieto jäsennetään. Julkaisemattomassa kirjassa *Mastering Ethereum* lohkoketjulle määritellään seuraavat komponentit, jotka sen tulee toteuttaa: vertaisverkkoliikenne ketjuun kuuluvien tietokoneiden välillä, jolla julkaistaan muutokset ja ylläpidetään verkkoa, riitojen selvittämiseen tarvittavat säännöt, transaktioiden välittämiseen tarvittavat viestit, tilakone, joka ylläpitää transaktioita ja tapahtumia, sekä hajautettu tietokanta (Antonopoulos & Wood 2018b).

3 SOLIDITY – ÄLYKKÄIDEN SOPIMUSTEN KIELI

Solidity on uusi ohjelmointikieli, jolla voidaan kehittää Ethereum-virtuaalikoneelle älykkäitä sopimuksia. Ne ovat ohjelmia, joita Ethereum-virtuaalikone eli EVM suorittaa. Soliditya kuvaillaan korkean tason ohjelmointikieleksi, joka on saanut vaikutteita muiden muassa C++-, JavaScript- ja Python-kielistä (Ethereum 2018a). Ohjelmia ei ole pakko kirjoittaa Solidity-kielellä, mutta Solidity-kääntäjä tuottaa suoraan EVM:n ymmärtämää tavukoodia sekä tukee lohkoketjun vaatimia asioita kuten käyttäjän tunnistusta ja varmistussummien laskentaa.

EVM voi suorittaa ohjelmia, jotka on kirjoitettu Solidity-ohjelmointikielellä ja käännetty sille sopivalle konekielelle. EVM toimii kuin mikä tahansa tietokone, mutta ympäristö on tiukasti määritelty ja rajattu. Ominaisuudet, jotka toimivat kehittäjän koneella, toimivat samalla tavalla kaikissa Ethereum-verkon koneissa. EVM estää ohjelmien ulospääsyn, jolloin kehittäjä ei voi käyttää kuin resursseja, jotka ovat EVM:n hallinnassa ja määritelty sille kuuluviksi.

Solidity määrittelee älykkään sopimuksen kasaksi koodia ja dataa (Ethereum 2018b). Tämä tarkoittaa, että lohkoketjuun lisätään Solidityllä käännetty tavukoodi. Sen tekemät muutokset muuttujiin lisätään tilakoneeseen, eli ohjelman lopputulema tallennetaan Ethereum-lohkoketjuun transaktioksi kuten se tallennettaisiin normaalisti vaikkapa kovalevylle.

Kuka tahansa lohkoketjun osakas voi tarkastella näitä lopputulemia, jotka siis ovat tietokannan transaktioita, eli ne ovat julkisia kaikille verkon osakkaille. Tämän selventämiseksi työssä käydään läpi muutama Solidity-esimerkki, joita kehittäjäyhteisö tarjoaa dokumentaatiossaan, sekä yhtä tähän opinnäytetyöhän kirjoitettua luokkaa. Ne kirjoitettiin Solidity 0.4 -kielellä, joka tarkoittaa, että ei ole takuita että ne toimisivat tulevaisuudessa uudemmilla versioilla. Tämä on esimerkki siitä, miten uusi ja epävakaa kieli Solidity on.

Seuraava esimerkki on yksinkertainen mutta summaa kaikki edellä kuvatut asiat:

```
pragma solidity ^0.4.0;

contract SimpleStorage {
```

```
    uint storedData;

    function set(uint x) public {

        storedData = x;

    }

    function get() public view returns (uint) {

        return storedData;

    }
}
```

(Ethereum 2018b).

Solidity on olio-ohjelmointikieli, jonka ymmärtäminen edellyttää olio-ohjelmoinnin perusteiden tuntemista. Kuten näkyy, ohjelma itsessään on yksinkertainen eikä juuri kelpaa muuhun kuin Solidityn perusteiden esittelyyn. Ohjelman tarkoitus on tallettaa muuttujan storeDatan tila lohkoketjuun. Tätä tallennettua muuttujaa voi kysyä metodilla get() SimpleStorage-objektilta.

Toisessa esimerkkiohjelmassa, joka on liitteenä 1, hyödynnettiin monia Solidityn ominaisuuksia. Tässä esimerkissä on luokat Tilallinen (Farmer), Rekat (Trucks) ja Kasvikset (Vegetables). Esimerkkiohjelmassa Tilallinen-luokka kasvattaa Kasviksia, omistaa ne ja laittaa sitten Rekkaan, jolla niitä voidaan kuljettaa. Koska Solidityn kyky yhdistellä ja löytää ulkopuolisia luokkia on rajallinen, ovat nämä toisistaan täysin erilliset luokat samassa tiedostossa, jotta ne pystyy kääntämään Solidity-kääntäjällä. Omistajuuden seuraaminen tapahtuu luomalla 256-bittinen tarkistussumma lähettäjän tiedoista.

3.1 Älykkään sopimuksen suorittaminen

Edellä olevat esimerkkiohjelmat eivät ole suoraan ajettavissa, vaan ne on käännettävä Solidity-kääntäjällä tavukoodiksi, joka voidaan suorittaa sitten Ethereum-virtuaalikoneella. Solidity ei siis ole tulkattava kieli vaan pseudo-tavukoodikieli kuten Java-ohjelmointikieli. Ethereum-alusta asettaa myös omia vaatimuksiaan sille, miten ja kuinka se suostuu suorittamaan tavukoodia. Tämä on ratkaistu Ethereum-alustalla vaatimalla niin sanottua kaasumaksua, joka tulee suorittaa ETH-valuutalla (Dannen 2017).

Kaasumaksun ajatus on, että jokaisen ohjelman suorittaminen, lataaminen verkkoon ja säilyttäminen siellä maksaa tietyn hinnan, joka on nimetty kaasunhinnaksi. Ohjelman lohkoketjun osaksi haluavalla käyttäjällä tulee siis olla tarpeeksi Ethereum-alustan ETH-valuuttaa, että hän voi suorittaa ohjelmia (älykkäitä sopimuksia) sekä maksaa kaasun eli saada ohjelma pyörimään verkossa ja niiden tulokset tallennettua lohkoketjuun.

Maksu määräytyy EVM:n sisäisellä laskutoimituksella. Alkumaksu määräytyy sen perusteella, mitä tarpeita ohjelmalla on. Ethereum määrittelee jokaiselle toiminnolle hinnan, ja näin katsotaan, mitä pelkästään ohjelman ajaminen maksaa. Sen liittäminen osaksi lohkoketjua maksaa joka kerta aloitusmaksun ja tämän jälkeen siitä on maksettava kaasumaksu joka kerran, kun se ajetaan. Ohjelmaa siis suoritetaan niin kauan, että se loppuu tai maksajalta loppuvat rahat siihen tarvittavan kaasun maksamiseen (Ray 2018b).

Kaasumaksu vaikeuttaa ohjelmien ajamista Etherum-alustalla. Siksi esimerkiksi IBM on luonut oman versionsa Ethereum-virtuaalikoneen standardista. Fabric on tarkoitettu älykkäiden sopimusten alustaksi yksityisissä lohkoketjuissa. Ethereum-alusta on myös paineen alla ottanut käyttöön tämän tekniikan omassa järjestelmässään. Nykyisin lohkoketjun voi lohkoa pienempiin aliverkkoihin tai sillä voi luoda oman yksityisen täysin julkisesta Ethereum-lohkoketjusta erillisen lohkoketjun (Robinson 2018).

3.2 Ethereum-alustan työkalut

Solidityllä tehdyn ohjelman eli älykkään sopimuksen suorittamiseksi tarvitaan monta erilaista työkalua ja se on käsin tehtynä työlästä. Seuraavaksi käydään läpi, mistä Ethereum-alusta muodostuu tällä hetkellä.

Ethereum-alusta on nimensä mukaisesti alusta. Tämä tarkoittaa, että se ei ole yksi yhtenäinen ohjelma vaan löyhä kasa erillisiä ohjelmia, jotka toimivat toistensa kanssa yhteen. Ethereum-alustalla toimivan työkalun tai ohjelman tulee seurata spesifikaatiota (Wood 2018).

Spesifikaatiossa käydään tarkasti läpi tarvittava matematiikka ja termit. Silti Ethereum-alustan käyttö ei vaadi protokollan tai spesifikaation tuntemusta eikä

sen takana olevan matematiikan tietämistä. Spesifikaatio on tarkoitettu yrityksille, jotka haluavat tehdä oman version, joka on yhteensopiva Ethereum-alustan kanssa. Ethereum-alustan ohjelmat ovatkin tietyllä tapaa referenssitoteutus, johon verraten muut voivat tarkastaa omien tulkintojensa yhteensopivuuden.

Tällä hetkellä Ethereum-alustan kulmakivet ovat Go-pohjainen Geth-lohkoketju-komentoriviohjelma, Aleth-lohkoketju C++ toteutus ja Solidity. Muitakin Ethereum-alustan ohjelmia kehitetään yhteisöpohjaisesti Github-alustalla.

3.3 Solidity-komentorivikääntäjä

Solidity-koodin saaminen ajettavaan muotoon tapahtuu Solidity-kääntäjällä. Kääntäjän tuottamalle tavukoodille on ABI eli Application Binary Interface. ABI määrittelee, miten ohjelma tulee sijoittaa käännettynä tiedostoon, että sitä ajava järjestelmä ymmärtää, miten se ajetaan. Tämä merkitsee, että vaikka kieli muuttuu, sen tulee seurata tiettyä kaavaa, millaisia ohjelmien eli sopimusten tulee olla. Tätä muuttuvaista spesifikaatiota voi lukea Solidityn dokumentaatiosta kohdasta *Contract ABI Specification* (Ethereum 2018c).

3.4 Geth-komentoriviohjelma

Oikeastaan Ethereum-virtuaalikonetta ei ole olemassa missään kuten ei ole Java-virtuaalikonettakaan. Spesifikaatioon pohjaavia toteutuksia on muutamia. Go-kielinen versio on Ethereum-yhteisön mukaan kaikkein lähimpänä spesifikaatiota. Sitä kutsutaan nimellä Go-Ethereum eli Geth-komentoriviohjelma, jonka pääominaisuudet ovat seuraavat:

- louhia ETH-valuuttaa
- siirtää varoja
- tehdä sopimuksia ja lähettää transaktioita
- seurata lohkoketjun historiaa.

Geth-ohjelmalla voi myös liittää Solidity-käännettyjä tavukoodiohjelmia lohkoketjuun, vaikkakin työläästi, sekä tehdä muita asioita, kuten luoda yksityisen omaan käyttöön tarkoitetun Ethereum-lohkoketjun.

4 TYÖN SUORITTAMINEN

Opinnäytetyön ajatus lähti omalta alaltani: komponenttien seurantaongelmasta pohjamaanrakennusalalla. Pohjamaan rakentaminen rajoittuu kaikkeen, mikä tapahtuu, ennen kuin varsinainen rakenne asennetaan perustuksen tai vastaavan varaan. Komponentit, joista tässä puhutaan, ovat pääsääntöisesti erilaisia paaluja tai niihin verrattavia komponentteja. Alalla on selvä tarve löytää uusia innovatiivisia tapoja toteuttaa paalukomponenttien seuranta siten, että se on luotettavaa pitkällä aikajänteellä. Tärkeää on myös, etteivät mahdolliset muutokset toimijoissa aiheuta sitä, että koko edellinen tietokanta komponenteista ja niiden sijoittelusta muuttuu arvottomaksi tai se pitää tehdä kokonaan uudestaan. Päätimme yhdessä anonyymin yhteistyöyrityksen kanssa kokeilla mahdollisuutta käyttää Ethereum-alustaa, Go-pohjaista Geth-komentoriviohjelmaa ja Solidityä tiedon seurantaan.

Työn toteuttaminen jakaantui kahteen osaan, joista isotöisin oli vaatimus omasta yksityisestä lohkoketjusta. Tämän jälkeen pyrittiin saamaan yksinkertainen ohjelma eli älykäs sopimus osaksi lohkoketjua ja suoritettua sen kautta.

4.1 Lohkoketjun sopivuudesta seurantaan

Lohkoketju on hajautettu tietokanta, joka on yhtenevä kaikille osapuolilleen eikä keskitettyä palvelinta tarvita. Tästä on hyötyä rakennusalalla, jossa on monia eri toimijoita, kuten komponentin toimittaja, pääurakoija ja aliurakoija. Osapuolilla on samaan dataan erilainen tarve ja sen tuottamiseen erilainen motiivi.

Kun rakennuksia rakennetaan, komponenttien alkuperää tai asennushetkeä on usein mahdoton jäljittää sen jälkeen, kun ne on laitettu paikalleen. Pohjamaarakennuksessa lyödyn paalun paaluttaja tai valmistaja ei ole tällä hetkellä jäljitettävissä aukottomasti rakenteen jäädessä näkymättömiin. Tällä hetkellä paaluista tallennettavat tiedot kerätään manuaalisesti. On paalun valmistajan vastuulla, että hän toimittaa sovitun paalun, paalun asentajan vastuulla, että kyseinen paalu upotetaan paikkaan, johon se on suunniteltu, sekä tilaajan tai viranomaisen vastuulla, että valvonta suoritetaan asiallisesti. Ajatuksena olisi lohkoketjua hyödyntäen tallentaa tieto siten, että tarvittava metatieto, esimerkiksi paalun pi-

tuus, tyyppi ja valmistuspäivä, olisi saatavilla. Muut tiedot olisivat saatavilla ainoastaan tarkistesumman avulla.

Lohkoketjun käyttö takaisi, että kaikki osapuolet voidaan tunnistaa aukottomasti. Tämän varmistaa käytössä oleva lompakko-järjestelmä, jossa jokaisella toimijalla ketjussa on oma personoitu identiteettinsä. Sen voi periaatteessa varastaa, mutta tyhjästä sellaisten avainparien luominen, jotka vastaisivat jotain toista toimijaa, ei ole ainakaan vielä onnistunut julkisessa Ethereum-lohkoketjussa. Lohkoketjun eduiksi komponenttien seurannassa todettiin seuraavat:

- Keskitettyä valvojaa ei tarvita transaktioiden tekemiseen.
- Kaikki transaktiot pinotuvat päällekkäin eli historia on aukoton. Prosessin tulokset ovat siis läpinäkyviä.
- Osapuolet voivat tunnistaa toisensa aukottomasti ja uusien osapuolien lisääminen onnistuu ilman lohkoketjun uudelleen aloittamista.
- Ohjelmia voidaan ajaa, lisätä ja muuttaa tarpeen mukaan.

4.2 Yksityisen lohkoketjun luominen Gethillä

Tekniikan omaksumisen jälkeen ensimmäinen ja työn haastavin osuus oli tehdä yksityinen lohkoketju. Tämä siksi, että luonteensa vuoksi tiedot eivät sovellu julkiseen Ethereum-lohkoketjuun. Työkaluksi valittiin Geth-komentoriviohjelma, koska sille löytyvä dokumentaatio oli ajantasaisinta ja sen käyttö vaikutti selkeimmältä. Geth osaa kaiken tarvittavan: louhintaa, lompakon ylläpitoa ja yksityisen lohkoketjun juurilohkon luomista.

Geth toimii siten, että jokainen instanssi on oma itsenäinen noodinsa eli sillä voi muodostaa suljetun yhteisön. Oman yksityisen verkkonsa voi avata, kunhan siinä on yksi noodi, jolla on osoite, jonka muut vertaisverkossa olevat tietävät. Tämän perusteella se muodostaa käytössä olevan lohkoketjun käynnistyskoneelle.

Perustana yksityiselle lohkoketjulle toimii alkulohko, jota Ethereum-alustalla kutsutaan Genesis-lohkoksi. Tämä lohko määrittelee, millainen lohkoketju on. Genesis-lohkon jälkeen lohkoketju on valmis käyttöön ja sitä voi alkaa käyttää. Ge-

nesis-lohko on määriteltävä ja se on louhittava kuten kaikki muutkin lohkoketjun osat.

4.2.1 Genesis-lohkon määrittely

Ethereum-alustan lohkoketjun alkulohkon nimi on Genesis-lohko. Se määriteltiin genesis.json-nimisellä määritystiedostolla. Seuraava on kokeiluverkkoon tarkoitettu genesis.json-määritystiedosto.

```
{
  "nonce" : "0x0000000000000055",
  "mixHash": ”0x0000000000000000000000000000000000000000000000000000000000000000",
  ”parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "difficulty": "0x20000",
  "gasLimit": "0x800000",
  "timestamp" : "0x0",
  "extraData" : "",
  "coinbase": "0x0000000000000000000000000000000000000000",
  "alloc": {},
  "config": {
  "chainId": 100,
  "homesteadBlock": 0,
  "eip155Block": 0,
  "eip158Block": 0
  }
}
```

Genesis.json-määritystiedosto, jota käytettiin testauksessa, on tässä lähinnä esimerkkinä, mutta sen käyttö lohkoketjulle on määräävää. Nimittäin yksityinen lohkoketju luodaan tämän tiedoston perusteella. Kun se on tehty, sitä ei voi kesken enää muuttaa. Tässä määritystiedostossa olevat nonce- ja mixhash-parametrit määrittävät, miten vaikea uusia lohkoja on louhia difficulty-parametrin kanssa. Gaslimit-parametri taas kertoo, kuinka suuri kaasumaksu on.

Tarkemmin Genesis.json-määritystiedostoa on käsitelty blogissa *What does each genesis.json parameter mean* (Mohan 2106). Blogissa selitetään, miten

määritystiedoston eri parametrit vaikuttavat kaiken takana vaikuttavaan matematiikkaan.

Se, että ohjeet löytyvät blogikirjoituksista, on mielestäni osa Ethereum-alustan käytön vaikeutta. Dokumentaatiota on olemassa mutta hajallaan. Genesis-lohkon tekoa on selitetty myös E-kirjan *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners* luvussa *Creating Private Chains* (Dannen 2017).

4.2.2 Genesis-lohkon louhiminen Geth-ohjelmalla

Dokumentaatio yksityisen lohkoketjun tekemiseen löytyy Ethereum-alustan wikisivuilta *Private network* (Lange 2017). Silti dokumentaation ongelmana on, että vaikka seuraisi sitä kirjaimellisesti, ei sen perusteella saa tehtyä yksityistä lohkoketjua, vaan sen tekemiseen joutuu tekemään useampia erillisiä kokeiluja.

Genesis-lohkon laskemiseen tarvitaan hakemisto, jossa tämä alkulohko ja oikeastaan koko lohkoketju sijaitsee. Koska tässä opinnäytetyössä käytettiin Linux-ympäristöä, valittiin hakemisto /tmp/eth/data-private. Hakemisto voi siis olla ihan mikä vain, kunhan siellä alussa ei ole yhtään tiedostoa. Esimerkin genesis.json-määritystiedosto sijaitsi paikassa /tmp/eth/configs/genesis.json.

Genesis-lohkon laskeminen tapahtui seuraavasti:

```
geth --datadir /tmp/eth/data-private init /tmp/eth/configs/genesis.json
```

Tämän jälkeen kone laski eli louhi Genesis-lohkoa muutamista kymmenistä sekunneista aina muutamiin minuutteihin. Tässä lohkoketjussa ETH-valuuttaa ei tarvita, koska älykkäiden sopimusten tekeminen ei maksa mitään. Samalla näkee, miten lohkoketjuun pääsee käsiksi. Verkon tunnistenumeroksi valittiin sattumanvarainen numero 42 ja komentoportiksi portin numero 60303. Lohkoketju sijaitsi edelleen hakemistossa /tmp/eth/data-private.

```
geth --networkid 42 --port 60303 --rpc --lightkdf --cache 16 --datadir /tmp/eth/data-
private console
```

Tämä komento avasi konsolin, joka käyttää laskettua yksityistä lohkoketjua. Konsolissa tehtiin tähän lohkoketjuun lompakko komennolla:

```
personal.newAccount()
```

Tämä komento kysyy lompakon salasanaa ja tulostaa lompakon osoitteen. Molemmat tulee kirjata ylös, koska muuten lompakon sisältöä ei pääse enää tarkastamaan.

Tässä vaiheessa poistuttiin pois Geth-konsolista ja tehtiin siitä louhija, johon muut verkon osakkaat voivat liittyä. Geth-verkon osakkaan nimi oli TestMiner1 ja lompakon osoite 0x9d956ebf9d30c3f1a0c6282fdef1c4379d22d8f1.

```
geth --identity TestMiner1 --nodiscover --networkid 42 --port 60303 --maxpeers 10 --
lightkdf --cache 16 --rpc --rpccorsdomain "*" --datadir /tmp/eth/data-private --etherba-
se "0x9d956ebf9d30c3f1a0c6282fdef1c4379d22d8f1" --minerthreads 1 --mine
```

Mikäli verkossa ei ole yhtään louhijaa, älykkäitä sopimuksiakaan ei voi suorittaa, koska jotta transaktio saadaan lohkoketjuun, tulee louhijan louhia se. Mitä enemmän louhijoita, sen nopeammin transaktiot menevät lohkoketjuun.

4.3 Solidity-kääntäjän käyttäminen

Seuraavaksi käydään läpi, miten Solidity-kääntäjän ja ympäristön saa tehtyä, jotta saa käännettyä älykkään sopimuksen ja liitettyä sen omaan yksityiseen lohkoketjuun.

Solidityllе ei ole muuta graafista kehitysympäristöä kuin verkossa toimiva Remix. Tämä takia on perusteltua käyttää komentoriviversiota. Se on saatavilla monella käyttöjärjestelmälle mutta mikäli se ei ole saatavilla, joutuu kääntämään sen käsin alusta alkaen.

4.3.1 Solidity-kääntöympäristön kääntäminen

Solidity-ympäristö jaellaan Github-sivustolla ja siitä on saatavana lähdekoodipaketti. Tässä käytettiin 0.4-versiota, jonka korvasi Solidityn 0.5-versio vuoden 2018 lopussa.

Solidity käyttää kääntämiseensä Cmake-kääntötyökalua sekä eri alustoilla joko GNU Make- tai Windows-ympäristössä joko Nmake- tai Visual Studio -ympäristöä. CMake tarkastaa järjestelmässä tarvittavat kirjastot, jotta lähdekoodi voidaan kääntää C++-kääntäjällä. Tässä esimerkissä oletettiin, että loppukääntö tehdään käyttäen Make-työkalua.

Seuraava komento ajettiin Solidityn lähdekoodipaketin juurihakemistossa:

```
cmake -DCMAKE_BUILD_TYPE=Release -DBoost_USE_STATIC_LIBS=OFF
```

Cmake-kääntötyökalu tulostaa selkeän ulostulon siitä, voiko ohjelmaa kääntää kyseisellä koneella. Kun Cmake-kääntötyökalu menee läpi ilman ongelmia, kääntäminen tapahtuu seuraavasti komennolla:

```
make
```

Käännön tapahduttua solc-kääntäjän pitäisi olla saatavilla.

4.3.2 Solidity-kääntäjän kokeileminen Hello World -esimerkillä

Hello World eli Terve Maailma on esimerkkiohjelma, joka tulostaa Hello World- tai Terve Maailma -tekstin ruudulle tai käytetylle tulostuspinnalle. Solidity-kielellä se tehtiin seuraavasti:

```
pragma solidity ^0.4.4;

contract HelloWorldExampleContract  {

    function SayHello() public pure returns(string) {

        return "Hello World";

    }

}
```

Tämä esimerkki käännetään joko Ethereum Remix -ohjelmistolla tai sitten C++–solc-kääntäjällä. Jos käytetään solc-kääntäjää, se tehdään seuraavasti.

```
solc -o target --bin --abi Hello.sol
```

Komento tuottaa seuraavaan hakemistoon target-tiedostot HelloWorldExample-Contract.abi ja HelloWorldExampleContract.bin. ABI-tiedosto sisältää Application Binary Interface -tiedot. Tämän hetkinen ABI-määrittely tehdään JSON-notaatiolla ja edellä olevan koodin ABI näytti seuraavalta.

```
[
  {
     "inputs" : [],
     "payable" : false,
     "constant" : true,
     "name" : "SayHello",
     "type" : "function",
     "stateMutability" : "pure",
     "outputs" : [
        {
           "type" : "string",
           "name" : ""
        }
     ]
  }
]
```

Käytännössä tiedoston sisältö on siis SayHello-funktion määrittely eli mitä parametrejä se ottaa vastaan ja mitä se palauttaa. Binääritiedosto HelloWorldExampleContract.bin taas sisältää ajettavan koodin EVM-virtuaalikoneen ymmärtämässä muodossa. Tähän mennessä työssä on luotu Gethin avulla täysin oma yksityinen lohkoketju ja käännetty Solidityllä esimerkkiohjelma, jolla testattiin, että älykkään sopimuksen sai ajettua lohkoketjussa.

```
pragma solidity ^0.4.20;

// This one is copied from https://arvanaghi.com/blog/testing-smart-contracts-on-a-private-blockchain-with-Geth/

contract mortal {

    /* Define variable owner of the type address */

    address owner;



    /* This function is executed at initialization and sets the owner of the contract */

    function mortal() public { owner = msg.sender; }



    /* Function to recover the funds on the contract */

    function kill() public { if (msg.sender == owner) selfdestruct(owner); }

}

contract greeter is mortal {

    /* Define variable greeting of the type string */

    string greeting;



    /* This runs when the contract is executed */

    function greeter(string _greeting) public {

        greeting = _greeting;

    }



    /* Main function */

    function greet() public constant returns (string) {

        return greeting;

    }

}
```

Ohjelma on yksinkertainen esimerkiksi käyvä koodi. Älykkään sopimuksen saattaminen Geth-ohjelman kanssa ajettavaan muotoon ei ole käsin tehtynä kovin nopea eikä automaattinen prosessi. On tehtävä seuraavat asiat, että lohkoketju hyväksyy sopimuksen osakseen. Esimerkissä yllä oleva koodi oli Greeter.sol -nimisessä tiedostossa. Sen kääntäminen käyttäen ylempää esimerkkiä tapahtui seuraavasti.

```
solc --overwrite -o target --bin --abi Greeter.sol
```

Mikä käänsi yllä olevan koodin target-hakemistoon, jossa tulisi olla tiedostot greeter.abi/bin ja mortal.abi/bin. Tämän jälkeen käynnistettiin Geth-konsoli ja alettiin liittää näitä osaksi lohkoketjua.

```
greeterHex = "0x<Tähän Greeter.bin-tiedoston sisältö>"
```

Tämän jälkeen ABI-tiedostolle tehtiin sama.

```
greeterAbi = [{"constant":false,"inputs":[],"name":"kill","outputs":
[],"payable":false,"stateMutability":"nonpayable","type":"function"},
{"constant":true,"inputs":[],"name":"greet","outputs":
[{"name":"","type":"string"}],"payable":false,"stateMutability":"view","type":"function"
},{"inputs":
[{"name":"_greeting","type":"string"}],"payable":false,"stateMutability":"nonpayable","t
ype":"constructor"}]
```

Nyt Geth-ohjelmalla on tieto koodista ja että siinä on tämän kaltaiset muuttujat. Tämän jälkeen koodi ajettiin eli louhittiin osaksi lohkoketjua. Tämä tarvitsee edellä luotua lompakkoa:

```
personal.unlockAccount(eth.accounts[0])
```

Sitten avattiin älykkään sopimuksen louhiminen ja lisättiin muuttujaan greeting teksti This is the greeting with which we will instantiate the contract. Hi!.

```
greeterInterface = eth.contract(greeterAbi)

greeterTx = greeterInterface.new(

  "This is the greeting with which we will instantiate the contract. Hi!",

  {

    from: eth.accounts[0],

    data: greeterHex,

    gas: 1000000

  }

)
```

Tämä jälkeen saatiin tälle lohkolle kuuluva tarkastussumma, joka kannattaa ottaa ylös. Tämän jälkeen käynnistettiin louhinta.

```
greeterTxHash = greeterTx.transactionHash

miner.start()

null

miner.stop()

true
```

Lopuksi tarkastettiin, että louhinta liitti tämän sopimuksen osaksi lohkoketjua.

```
eth.getTransactionReceipt(greeterTxHash)
```

Se tulostaa tämän tyylisen ulostulon jos se onnistuu.

```
{

  blockHash: "0x3f1208262112c6a691f18b22fcf2592fb738f7668257875f38ade6a93ff53ae5",

  blockNumber: 1132,

  contractAddress: "0x99e8c28d96eb82bb90e14e0d016d85ba904dd7ad",

  cumulativeGasUsed: 684972,

  from: "0x9d956ebf9d30c3f1a0c6282fdef1c4379d22d8f1",

  gasUsed: 342486,

  logs: [],

  logsBloom:
"0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000",

  root: "0xb8195de407a0e5d80e211268e496fd07f508aa4fac50ee257b56b76bf38d395e",

  to: null,

  transactionHash: "0x7a63641b49140f4b0afcb4b0973568badd542183f1d9753ba7de70190fd13e72",

  transactionIndex: 1

}
```

Tämän jälkeen tätä sopimusta voidaan käyttää paikallisesti.

```
greeterInterface.at(publishedGreeterAddr).greet()
```

Komennon suorituksen jälkeen ruudulle ilmestyy ohjelman ajon lopputulema.

```
"This is the greeting with which we will instantiate the contract. Hi!"
```

Toisesta noodista, joka on liitetty verkkoon, voidaan tätä älykästä sopimusta käyttää seuraavasti. Tähän tarvitaan sopimuksen osoite, joka löytyy alkuperäiseltä noodilta komennolla:

```
publishedGreeterAddr = eth.getTransactionReceipt(greeterTxHash).contractAddress
```

Tämän jälkeen toisella noodilla ajettiin seuraavat komennot.

```
greeterAbi = <Tiedoston Greeter.abi sisältö>

greeterInterface = eth.contract(greeterAbi)

publishedGreeterAddr = <publishedGreeterAddr>

greeter = greeterInterface.at(publishedGreeterAddr)

greeter.greet()
```

Tämän tulisi tulostaa seuraava teksti.

```
"This is the greeting with which we will instantiate the contract. Hi!"
```

Edellä mainittu tapa sopii vain testaukseen, mutta kaikki kohdat ovat tarpeen, jos älykäs sopimus halutaan osaksi lohkoketjua Geth-ohjelmalla.

5 LOPPUSANAT

Opinnäytetyön tavoitteena oli, että pääsisimme yhteistyöyrityksen kanssa esimerkin myötä etenemään lohkoketjujen käytön suhteen paalukomponenttien seuraamisessa. Kaikilta osiltaan projekti ei ollut tyydyttävä, koska Ethereum-alustalla, ainakaan käyttämällä Geth-ohjelmaa, ei ole mahdollista saada riittävän automaattista älykkäiden sopimusten solmimista.

Tämä opinnäytteen perusteella saatiin kuitenkin yhteistyöyrityksen kanssa käsitys, missä ovat pahimmat kipupisteet, jos halutaan ottaa käyttöön lohkoketjutekniikka Ethereum-alustalla toteutettuna. Ongelmat olivat

- lohkoketjutekniikkaa koskevan tiedon hajanaisuus ja julkaisujen laatu
- lohkoketjutekniikan uutuus ja epävarmuus sen pysyvyydestä
- ohjelmointikielellä eli Solidityllä tehtyjen ohjelmien puute
- Solidityllе olevien ohjelmointikirjastojen puute
- integraation puute vanhoihin järjestelmiin.

Kehitysehdotuksena olisi valita C++-versio Ethereum-alusta implementaatiosta Go-kielisen sijasta. Tällöin päästäisiin käsin tekemisestä. Silti kokonaiskuvan kannalta Go-kielisen Geth-lohkoketjukomentoriviohjelman valinta oli oikea. C++-versiosta on vielä vähemmän dokumentaatiota kuin Go-kielisestä versiosta.

Yksi asiakkaan tavoite oli selvittää, miten lohkoketjun voi integroida olemassa olevaan tietokantaan ja tuotannon ohjaukseen. Ongelmaksi muodostui teknologiakuilu. Uuden, vasta kehittymässä olevan teknologian kuten älykkäiden sopimusten selittäminen on hankalaa, samoin kuin se, että lohkoketju itsessään on tietokanta. Totesimme, että integraatio vanhoihin järjestelmiin on mahdollista, mutta se vaatii lisäselvityksiä sekä ohjelmointityötä.

Omaan oppimiseen tämän opinnäytetyön tekeminen toi paljon. Periaatteessa minulla oli vapaat kädet kokeilla, miten Ethereum-alusta toimii. Huomasin, että

Ethehreum sitoo ohjelmoijan tekemään asiat tietyllä tavalla. Toisinaan esimerkiksi louhiminen yksityisessä verkossa on täysin turhaa.

Olen oppinut lohkoketjutekniikasta ja tullut vakuuttuneeksi siitä, että se tai sen jälkeen tuleva tekniikka tulee muuttamaan asioita. Silti lohkoketjun ongelma on sen määrittely. Myös tietotekniikkaa tuntematon asiakas voi tunnistaa tämän kaltaisesta asiasta innovaation. Tällä hetkellä lohkoketjutekniikan käyttö ei kuitenkaan yleistyne sen toteuttamisen vaikeuden takia.

Ethereum-alusta vaatii vielä kaupallisemman otteen, mikäli se haluaa nousta seuraavalle tasolle. Hyvä merkki on, että siitä kirjoitetaan kirjoja ja oppaita. Se merkitsee alustan vakiintumista, kun voidaan olettaa, että kirjojen tieto ei vanhene, ennen kuin ne julkaistaan.

Nähtäväksi jää, mikä lohkoketjutekniikoista lopulta on se, joka nousee laajamittaiseen käyttöön. Se voi olla Ethereum-alusta tai vaikkapa Linux Foundationin Hyperledger, johon mainittu IBM Fabric on lahjoitettu. Sen tukijat ovat moninaiset ja siinä on monia erilaisia lohkoketjuprojekteja nivottu yhteen.

Edellä on kuvattu ongelmia, mutta on myös paljon, mitä lohkoketju voisi ratkaista. Yksi esimerkki on saumaton kirjautuminen niin, että kaikilla olisi aukoton identiteetti verkossa. Tämä toisi mahdollisuuksia tietojen seurattavuuteen useilla aloilla, kuten potilastietojen käsittelyssä, sekä erilaisiin anturitietoihin, joiden tapahtumat pitää voida ajoittaa aukottomasti.

LÄHTEET

Antonopoulos, Andreas M. & Wood, Gavin 2018b. Mastering Ethereum. Saatavissa: https://github.com/ethereumbook/ethereumbook/blob/develop/what-is.asciidoc. Hakupäivä 23.09.2018.

Antonopoulos, Andreas M. & Wood, Gavin 2018a. Smart contracts and Solidity. Saatavissa: https://github.com/ethereumbook/ethereumbook/blob/develop/smart-contracts-solidity.asciidoc. Hakupäivä 23.09.2018.

Arvanaghi, Brandon 2018. Testing Smart Contracts Locally using Geth. Saatavissa: https://arvanaghi.com/blog/testing-smart-contracts-on-a-private-blockchain-with-Geth/. Hakupäivä 30.09.2018.

Bajpai, Prableen 2017. How Stock Exchanges Are Experimenting With Blockchain Technology. Saatavissa: https://www.nasdaq.com/article/how-stock-exchanges-are-experimenting-with-blockchain-technology-cm801802. Hakupäivä 23.09.2018.

Dannen, Chris 2017. Introducing Ethereum and Solidity. Saatavissa: http://proquest.safaribooksonline.com.ezp.oamk.fi:2048/9781484225356. Hakupäivä 29.09.2018.

Ethereum 2018c. Contract ABI Specification. Saatavissa: https://solidity.readthedocs.io/en/develop/abi-spec.html. Hakupäivä 30.09.2018.

Ethereum 2018b. Introduction to Smart Contracts. Saatavissa: https://solidity.readthedocs.io/en/v0.4.24/introduction-to-smart-contracts.html. Hakupäivä 29.09.2018.

Ethereum 2018a. Solidity. Saatavissa: https://solidity.readthedocs.io/en/v0.4.24/. Hakupäivä 30.09.2018.

Franco, Pedro 2014. Understanding Bitcoin: Cryptography, Engineering and Economics. West Sussex: John Wiley & Sons, Incorporated.

Gueiros, Solange 2017. The Geth's saga: setting up Ethereum private network on windows. Saatavissa: https://medium.com/@solangegueiros/https-medium-com-solangegueiros-setting-up-ethereum-private-network-on-windows-a72ec59f2198. Hakupäivä 30.09.2017.

historyofbitcoin.org 2018. History of Bitcoin. Saatavissa: http://historyofbitcoin.org/. Hakupäivä 17.11.2018.

Khatwani, Sudhir 2018. What is Double Spending & How Does Bitcoin Handle It?. Saatavissa: https://coinsutra.com/bitcoin-double-spending/. Hakupäivä 04.12.2018.

Lange, Felix 2017. Setting up private network or local cluster. Saatavissa: https://github.com/ethereum/go-ethereum/wiki/Setting-up-private-network-or-local-cluster. Hakupäivä 30.09.2018.

McGrath, Robert 2016. Ethereum Virtual Machine is Turing-complete. Saatavissa: https://robertmcgrath.wordpress.com/2016/03/31/ethereum-virtual-machine-is-turing-complete/. Hakupäivä 29.09.2018.

Meunier, Sebastien 2016. Blockchain technology—a very special kind of Distributed Database. Saatavissa: https://medium.com/@sbmeunier/blockchain-technology-a-very-special-kind-of-distributed-database-e63d00781118. Hakupäivä 04.12.2018.

Mohan, Nikhil 2016. What does each genesis.json parameter mean?. Saatavissa: https://lightrains.com/blogs/genesis-json-parameter-explained-ethereum. Hakupäivä 30.09.2018.

Ray, James 2018b. A Next-Generation Smart Contract and Decentralized Application Platform. Saatavissa: https://github.com/ethereum/wiki/wiki/White-Paper#ethereum-state-transition-function. Hakupäivä 29.09.2018.

Ray, James 2018a. Ethereum. Saatavissa: https://github.com/ethereum/wiki/wiki/White-Paper#ethereum. Hakupäivä 29.09.2018.

Reinikainen, Pauli 2017. Bitcoinin arvo nousi huippulukemiin: "Sijoittaminen erittäin riskialtista". Saatavissa: https://www.yrittajat.fi/uutiset/558758-bitcoinin-arvo-nousi-huippulukemiin-sijoittaminen-erittain-riskialtista. Hakupäivä 17.11.2018.

Robinson, Peter 2018. Requirements for Ethereum Private Sidechains. Saatavissa: https://github.com/ethereum/wiki/wiki/White-Paper#ethereum-state-transition-function. Hakupäivä 29.09.2018.

Satoshi Nakamoto 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. Saatavissa: https://bitcoin.org/bitcoin.pdf. Hakupäivä 4.12.2018.

Tietoviikko 2018. IBM ja Maersk luottavat lohkoketjuun – tuonut jo 40 prosentin edun. Saatavissa: https://www.tivi.fi/Kaikki_uutiset/ibm-ja-maersk-luottavat-lohkoketjuun-tuonut-jo-40-prosentin-edun-6735838. Hakupäivä 23.09.2018.

Urban, Holly 2018. Blockchain: What Is a “Distributed Ledger” and Why Is It Useful to Lawyers?. Saatavissa: https://www.lawtechnologytoday.org/2018/10/blockchain-and-why-is-it-useful-to-lawyers/. Hakupäivä 04.12.2018.

Wood, Gavin 2018. Ethereum: a secure decentralised generalised transaction ledger. Saatavissa: https://ethereum.github.io/yellowpaper/paper.pdf. Hakupäivä 17.11.2018.

LIITE 1, SOLIDITY LUOKKA ESIMERKKI

```
pragma solidity ^0.4.20;

/**
* This example shows how to make an Vegetable Dealer who has farmers
* and trucks to gather vegetables
*
* This is very brutal and unsecure example. It just shows example what
* smarts contracts has to offer and what EVM (Ethreum Virtual Machine) is
* good for.
*
*
* This image tries to come up idea how diffrent Contract partners deal
* each other
*
* - Dealer
*    +-Stock <-+
*    +-Trucks  |
*    | +-Truck |
*    |   +-Cargo <--+
*    +-Farmers      |
*       +-Farmer     |
*        +-Vegetables
*          +-Vegetable
*/

/* Dealer contract has farmers and trucks */
contract Dealer {
   address owner;
   Farmer[] farmers;
   Truck[] trucks;
   Vegetable[] vegetablesToDeal;

   /* Create Dealer contract */
   function Dealer() public {
       owner = msg.sender;
   }

   /**
    * Add farmer to dealer. Before this happens Dealer and Farmers has to make
    * succesful contract that Dealer can use Farmers stuff
    *
    * @param farm Farm to add
    */
   function addFarmer(Farmer farm) public {
       farmers.push(farm);
   }

   /**
    * Add truck to dealer. Before this happens Dealer and Truck owner has to
    * make succesful contract that Dealer can add cargo to truck
    *
    * @param truck Truck to add Dealers fleet
    */
   function addTruck(Truck truck) public {
       trucks.push(truck);
   }

   /**
    * Return how many farmers belogns to dealers potential
    *
    * @return amount of farmers or 0 if non
    */
   function getAmountOfFarmers() public view returns (uint) {
       return farmers.length;
   }

   /**
    * Return how many trucks belogns to dealers fleet
    *
    * @return amount of trucks or 0 if non
```

```
    */
   function getAmountOfTrucks() public view returns (uint) {
       return trucks.length;
   }

   /**
    * Return specific farm of thrown an error if there is not that
    * farm
    *
    * @return Farmer or throw a error
    */

   function getFarm(uint farm) public view returns (Farmer) {
      require(getAmountOfFarmers() < farm);
      return farmers[farm];
   }

   /**
    * Return specific truck of thrown an error if there is not that
    * truck
    *
    * @return Truck or throw a error
    */
   function getTruck(uint truck) public view returns (Truck) {
      require(getAmountOfTrucks() < truck);
      return trucks[truck];
   }

   /**
    * Return how much vegetables are available spefic Farmer
    *
    * @return Amount of vegetable or throw a error
    */
   function getVegetablesInFarm(uint farm) public  view returns (uint) {
      getFarm(farm).amountOfVegetables();
   }

   /**
    * Return how much vegetables are available spefic Truck
    *
    * @return Amount of vegetable or throw a error
    */
   function getVegetablesInTruck(uint truck) public view returns (uint) {
      getTruck(truck).getAmountOfCargo();
   }

   /**
    * Add cargo from specific Farmer to Truck. If there is not that much
    * vegtables we just add as much we can
    *
    * @param farm Farmer where we collect vegetables
    * @param truck truck to add these vegetables
    * @param amount how much we add
    */
   function addCargoToTruck(Farmer farm, Truck truck, uint amount) public {
       truck.addCargo(farm, amount);
   }

   function kill() public {
       if (msg.sender == owner) {
           selfdestruct(owner);
       }
   }
}

contract Truck {
   address owner;
   bytes20 truckId;
   Vegetable[] currentCargo;
   uint maxAmountOfVegetables = 100;

   /* Create Truck and generate unique id for it */
   function Truck() public {
       owner = msg.sender;
       truckId = bytes20(keccak256(msg.sender, block.blockhash(block.number - 1)));
```

```
   }

   /**
    * Add cargo from Farmer or throw a error if there is enough
    *
    * @param farm Farmer where to gather vegetables
    * @param amount who much we take
    */
   function addCargo(Farmer farm, uint amount) public {
      require(farm.amountOfVegetables() > 0);

      if(amount > farm.amountOfVegetables()) {
           amount = farm.amountOfVegetables() - 1;
      }

      for(uint x = 0; x < amount; x++) {
         if(currentCargo.length >= maxAmountOfVegetables)
         {
            break;
         }
         currentCargo.push(farm.decreaseVegetable());
      }
   }

   /**
    * Give cargo away of if truck doesn't have any throw a error
    *
    * @return Vegetable which is last in or throw a error if empty
    */
   function decreaseCargo() public returns (Vegetable)
   {
       require(currentCargo.length > 0);

       Vegetable veggie = currentCargo[currentCargo.length - 1];
       delete currentCargo[currentCargo.length - 1];
       return veggie;
   }

   /**
    * How much we have cargo in this truck
    *
    * @return Amount of cargo
    */
   function getAmountOfCargo() public view returns (uint)
   {
       return currentCargo.length;
   }

   function kill() public {
       if (msg.sender == owner) {
           selfdestruct(owner);
       }
   }
}

contract Farmer {
   address owner;
   bytes20 farmerId;
   Vegetable[] VegetablesGrow;

   /* Create farmer and generate farmer unique id */
   function Farmer() public {
       owner = msg.sender;
       farmerId = bytes20(keccak256(msg.sender, block.blockhash(block.number - 1)));
   }

   /**
    * Add vegetable to farm
    *
    * @param vType Which kind of vegetable to add
    */
   function addVegetable(Vegetable vType) public {
       VegetablesGrow.push(vType);
   }
```

```
    /**
     * Add multiple vegetables in same time
     *
     * @param vType Which kind of vegetable to add
     * @param amount How many vegetables we like to add
     */
    function addVegetable(Vegetable vType, uint amount) public {
        for(uint x = 0; x < amount; x++) {
           addVegetable(vType);
        }
    }

    /**
     * Get vegetable out of farm or throw a error
     *
     * @return Return last vegetable in or throw a error if there is non left
     */
    function decreaseVegetable() public returns (Vegetable) {
        require(VegetablesGrow.length > 0);

        Vegetable veggie = VegetablesGrow[VegetablesGrow.length - 1];
        delete VegetablesGrow[VegetablesGrow.length - 1];
        return veggie;
    }

    function amountOfVegetables() public view returns (uint) {
        return VegetablesGrow.length;
    }

    function kill() public {
        if (msg.sender == owner) {
            selfdestruct(owner);
        }
    }
}

contract Vegetable {
    address owner;
    bytes20 veggieId;
    enum vegetableTypes {
        Tomato,
        Cucumber,
        Carrot,
        Potato
    }
    vegetableTypes public typeOfVegetable = vegetableTypes.Cucumber;
    uint amountOfVegetables = 0;

    /* Vegetable base class also generate veggie unique id */
    function Vegetable() public {
        owner = msg.sender;
        veggieId = bytes20(keccak256(msg.sender, block.blockhash(block.number - 1)));
    }

    /**
     * Type of vegetable
     *
     * @param vType type of vegetable
     */
    function setType(vegetableTypes vType) public {
        typeOfVegetable = vType;
    }

    /**
     * Get type of vegetable
     *
     * @return Type of vegetable
     */
    function getType() public view returns (vegetableTypes) {
        return typeOfVegetable;
    }

    function kill() public {
        if (msg.sender == owner) {
            selfdestruct(owner);
```

```
        }
    }
}
```