Haaga-Helia
ammattikorkeakoulu Oy

Lohkoketjuteknologia esimerkin kautta

Joonas Järvenpää

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2020

Haaga-Helia ammattikorkeakoulu Oy

Tiivistelmä

12.11.2020

Tekijä(t)
Joonas Järvenpää

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön otsikko
Lohkoketjuteknologia esimerkin kautta

Sivu- ja liitesivumäärä
44 + 26

Opinnäytetyön otsikko englanniksi
Blockchain technology through example

Satoshi Nakamoton vuonna 2008 julkaiseman Bitcoin-virtuaalivaluutan myötä keksitty blockchain- eli lohkoketjuteknologia on mullistanut IT-alaa ja toi tavan toteuttaa sellaisia digitaalisia järjestelmiä, joiden luomisessa ilmenneiden ongelmien ratkaiseminen olisi ollut ennestään erittäin vaikeaa.

Vaikka useat tunteekin jo Bitcoinin nimeltä ja moni on kuullut sen nerokkuuden taustalta löytyvästä lohkoketjutekniikastakin puhuttavan, voi monelle lohkoketjuun tutustuneellekin olla hankala määritellä lohkoketjun toimintaperiaatteita.

Tämän opinnäytetyön aiheena oli tutkia lohkoketjua, sen toimintaperiaatteita ja toimintamekaniikkaa, sekä sisäistää syvällisemmin sen teoriaa soveltamalla sitä konkreettisella tasolla. Opinnäytetyössä selvitettiin, että miten lohkoketjun periaatteet näkyy käytännössä, sekä miten lohkoketjulla toimiva järjestelmä luodaan ja mitä se pitää sisällään.

Menetelmänä aiheen tutkimiselle toimi Bitcoin-virtuaalivaluutan lohkoketjujärjestelmää demonstroivan Java-kielisen ohjelman luominen itse.

Oman lohkoketjun periaatteiden mukaisesti toimivan järjestelmän luominen auttoi sisäistämään teoreettisen lohkoketjun toiminnan käytännön tasolle, ja antoi tarkemman käsityksen siitä, mitä lohkoketjulla toimivassa järjestelmässä tapahtuu abstraktien käsitteiden peittämän pinnan alla, sekä miten lohkoketjun voisi toiminnallisuuksineen luoda itse.

Asiasanat
Lohkoketju, Bitcoin, Hajauttaminen

Sisällys

1 Johdanto

Voi tuntua joskus vaikealta pysyä mukana alati kasvavalla vauhdilla kehittyvien teknologioiden aallonharjalla. Siitä on vasta 20 vuotta kun Nokia julkisti kuuluisan Nokia 3310 -puhelinmallinsa. (Nokia Museum 2020.) Verratessa sitä nykypäivän mitä villeimpiin innovaatioihin voimme todeta ihmiskunnan teknologisen kehittymisen olleen huikea. Vuonna 2008 mysteerinen Satoshi Nakamoto loi avoimeen lähdekoodiin perustuvan Bitcoin -virtuaalivaluutan, joka sen julkaistaessa vuonna 2009 toi käyttämälleen lohkoketjuteknologialle suurta näkyvyyttä. Tämän jälkeen lohkoketjutekniikkaa on käytetty useilla tavoilla monenlaisissa teknologisissa innovaatioissa ja trendinä lohkoketjujen käyttäminen kasvaa vuosi vuodelta suuremmaksi. Lohkoketjut ovatkin tänä päivänä kuuma keskusteluaihe ja monet teknologiaa seuraavat tuntevat Bitcoinin lisäksi monia muitakin lohkoketjuimplementaatioita.

Vaikka Bitcoin alkaakin olla jo valtaväestöllekin tuttu käsite vähintäänkin nimensä perusteella, voi olla silti vaikeaa hahmottaa sen käyttämän lohkoketjutekniikan toimintamekaniikkaa. IT-ala kehittyy kokoajan entistä monipuolisemmaksi kokonaisuudeksi ja lohkoketjuja käyttävien innovaatioiden määrä kasvaa jatkuvasti, joten tulevana IT-alan ammattilaisena olisi tulevaisuudenkin kannalta hyvä ymmärtää myös lohkoketjun toiminnan perusperiaatteet.

Olen itse huomannut oppivani asioita parhaiten käytännönläheisesti tekemällä ja tiedän monien muidenkin ihmisten olevan asiassa samanlaisia. Internet on pulloillaan tietoa lohkoketjujen toiminnasta abstraktilla tasolla selitettynä, mutta voi olla haasteellista soveltaa kyseistä tietoa käytännössä.

Tässä opinnäytetyössä yhdistetään lohkoketjun toimintaperiaatteiden teoreettinen oppiminen käytäntöön luomalla esimerkki lohkoketjun toiminnasta Java -ohjelmointikielellä. Määrittämällä ja toteuttamalla oma lohkoketju toimintoineen askel askeleelta auttaa yhdistämään abstraktit käsitteet konkreettisiin esimerkkeihin.

Lohkoketjutekniikkaa käsitellään käytännön näkökulmasta sivuten Bitcoin -virtuaalivaluutan käyttämän lohkoketjun toimintamallia. Opinnäytetyön tarkoituksena on helpottaa lohkoketjun toimintaperiaatteiden sisäistämistä käytännön tasolle. Suunnitelmana on perehtyä lohkoketjutekniikkaan, sekä luoda ja dokumentoida Java -kielellä ohjelmoitu konkreettinen esimerkki, joka noudattaa lohkoketjun pääasiallisia toimintaperiaatteita.

Henkilökohtainen tavoitteeni on oppia lisää lohkoketjuista, ymmärtää sen toimintaperiaatteet käytännön tasolla, soveltaa tietoa järjestelmän konkretisoimiseksi, sekä tarjota opinnäytetyön lukijalle tavanomaisesta poikkeava ja käytännönläheinen näkökulma, josta käsin lohkoketjun toiminnan perusperiaatteet voi sisäistää.

Projektin tavoite on ensin analysoida ja eritellä lohkoketjun periaatteelliset toiminnot, toteuttaa toiminnot loogisessa järjestyksessä Java-ohjelman muotoon, sekä toteuttaa projektin dokumentointi tavalla, joka voi auttaa myös lukijaa yhdistämään toiminnallisuuksien konkreettiset esimerkit abstrakteihin käsitteisiin.

Opinnäytetyön lopputuloksena valmistuu tuotos, joka demonstroi lohkoketjun yleisiä toimintaperiaatteita, ja jonka dokumentaatiota seuraamalla Java-ohjelmointikielen tarvittavalla tasolla osaava lukija voi halutessaan luoda oman Bitcoin-virtuaalivaluutan lohkoketjua mukailevan Java-kielisen suppean esimerkkijärjestelmän. Tuotos auttaa ymmärtämään ja sisäistämään lohkoketjun toimintaperiaatteet käytännön tasolla, antamalla abstrakteille konsepteille konkreettiset esimerkit.

Opinnäytetyön ohjaamiseksi olen asettanut seuraavat tutkimuskysymykset:

- Mikä on lohkoketju?
- Miten lohkoketju toimii?
- Miten lohkoketjun voi luoda?

Tämä opinnäytetyö keskittyy lohkoketjun käsittelemiseen yleisesti tekniikkana, soveltaen luotavassa esimerkissä lähtökohtaisesti Bitcoin -virtuaalivaluutan käyttämän lohkoketjumallin toimintaperiaatteita. Lohkoketjuja on käytetty laajassa määrin hyvin erilaisiin tarkoituksiin muutellen, joten tässä opinnäytetyössä luotavan esimerkin ei ole tarkoitus kattaa kaikkia mahdollisia erilaisissa lohkoketjuissa käytettäviä toiminnallisuuksia. Myös muita lohkoketjulla toimivia järjestelmiä käsitellään esimerkkeinä lohkoketjujen soveltamisesta erilaisiin tarkoituksiin, mutta muiden tässä opinnäytetyössä luotavaan esimerkkiin liittymättömien innovaatioiden syvällisempi kuvaaminen pyritään jättämään pois, keskittyen käyttämään Bitcoinia esimerkkinä lohkoketjun ydintoiminnoista.

Tarkoitus on, ettei lukija tarvitse aiempaa perehtyneisyyttä lohkoketjusta, ja että Java -ohjelmointikielen perusteet omaava lukija pystyy ymmärtämään ohjelmakoodia ja yhdistämään sen kyseisen vaiheen teoriaan relevantisti seuraamalla järjestelmän luomisprosessia.

Tämän opinnäytetyön keskeiset käsitteet ovat lohkoketju, hajauttaminen ja bitcoin.

Lohkoketju (engl. Blockchain), on tietorakennetekniikka, joka mahdollistaa luotettavan ja varmennettavan tietokannan ylläpitämisen hajautetusti. Lohkoketjulla tarkoitetaan digitaalista rekisteriä, jossa jokainen tallennettu tieto on varmennettu yhteisymmärryksessä järjestelmän osallisten kesken, ja jossa rekisteriin syötetyn tiedon muuttaminen jälkeenpäin huomaamatta ei ole mahdollista. (Laurence 2017.)

Hajauttamisella (engl. decentralization) lohkoketjussa tarkoitetaan sitä, että järjestelmän toiminnasta ei vastaa luotettu keskeinen järjestelmää hallinnoiva taho, vaan järjestelmä ja sen toiminta on jaettu verkolle itsenäisiä osallistujia. (Laurence 2017.)

Bitcoin on suoraan käyttäjältä käyttäjälle toimiva, elektroninen valuuttajärjestelmä, joka toimii lohkoketjutekniikalla hajautetusti. Bitcoin on hajautettu järjestelmä, eli sitä ei hallinnoi mikään keskitetty taho kuten keskuspankki. Tiedon tallentaminen tapahtuu kryptografian turvin tavalla, jonka ansiosta käyttäjät pystyvät luottamaan sokeasti toisiinsa sekä tallennetun tiedon eheyteen, ilman tarvetta järjestelmän luotettavuuden ja toiminnan takaavalle keskitetylle taholle. (Nakamoto 2008.)

2 Lohkoketju

2.1 Historiaa

Mysteerisen Satoshi Nakamoton toimesta vuonna 2009 julkaistu Bitcoin -nimeä kantava virtuaalivaluutta toi kasvunsa myötä lohkoketjutekniikalle mainetta ja kunniaa ympäri maailman, suurin osa sen alustaneista teknisistä ideoista keksittiin monia vuosia aiemmin. Lohkoketju on tietynlainen hajautettu tietokanta, ja sen idea taas juontaa juurensa ainakin 1970-luvulle. Yleisesti ottaen idea kirjanpidosta juontaa juurensa ainakin muinaiseen Mesopotamiaan saakka. (Sherman, Javani, Zhang & Golaszewski 2018.)

Vuonna 1979 Ralph Merklen väitöskirja avasi idean tietoja sisältävien lohkojen tallentamisesta muuttumattomaan ketjuun hyödyntämällä kryptografista hajautusfunktiota (engl. hash function). Väitöskirja esitti, miten informaatio tallennetaan puuta muistuttavaan rakenteeseen (engl. tree structure), joka tunnetaan tänä päivänä nimellä ”Merkle hash tree”. (Sherman, Javani, Zhang & Golaszewski 2018.)

Myöhemmin vuonna 1991 Stuart Haber ja W. Scott Stornetta ideoivat tekniikan käyttämistä aikaleimadokumenteissa (Sherman, Javani, Zhang, Golaszewski 2018) ja myöhemmin Dave Bayerin kanssa julkaisemassaan artikkelissaan he tiivistävät ongelmaa siten, että paperimuotoisen, päivätyn tiedon muuttaminen on hankalampaa, ja että niiden ominaisuuksien vuoksi on mahdollista tutkia, onko dokumentteja muuteltu myöhemmin. Tämä eroaa digitaalisen muodon omaavasta päivätystä tiedosta, eikä digitaaliseen dokumenttiin välttämättä jää merkkejä peukaloinnista. (Bayer, Haber, Stornetta 1992). Vaikka Haberin, Stornettan ja Bayerin ideoimassa ensimmäisessä lohkoketjun esiasteessa on hyvin paljon samankaltaisuuksia Bitcoininkin käyttämän lohkoketjun toimintaperiaatteisiin, ei se kuitenkaan pidä sisällään kaikkia lohkoketjun elementtejä. (Sherman, Javani, Zhang & Golaszewski 2018.)

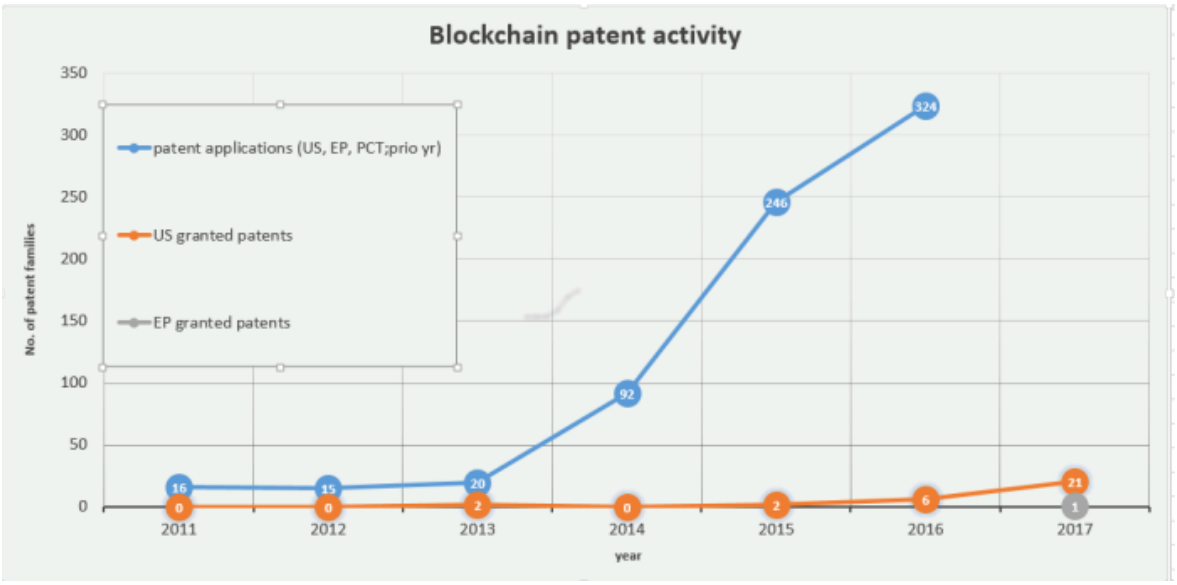
Yksi näistä elementeistä on konsensuksen, eli yksimielisyyden saavuttaminen talletettavan tiedon aikaleiman eheydestä. Vuonna 2002 Adam Back julkisti Hashcash -järjestelmän, jonka alkuperäinen tarkoitus on rajoittaa roskapostitusta sähköpostissa sekä palvelunestohyökkäyksiä tarjoamalla ”proof-of-work” -konsensusmekanismin, jota myös sivutaan Satoshi Nakamoton julkaisemassa Bitcoinin dokumentaatiossa. (Back 2002.)

Vuonna 2008 Satoshi Nakamoto -nimellä esiintyvä pseudonyymi henkilö tai taho julkisti dokumentin ”Bitcoin: A Peer-to-Peer Electronic Cash System”, tuoden julki ainutlaatuisen järjestelmän, joka käyttää lohkoketjutekniikkaa tavallaan. Järjestelmän julkaisun jälkeen

vuonna 2009, se alkoi keräämään tunnustusta myös kryptografian alan ulkopuolella. Hajautettu ei-kenenkään hallitsema elektroninen valuutta, jossa jokainen käyttäjä pystyi luottaa toisiinsa nähtiin jopa ”maailman mullistavana”. (Wallace 2011.)

Tämä virallisista tahoista riippumaton suora loppukäyttäjien muodostama luottamusverkko olikin Bitcoinin pääperiaate. Bitcoin-julkaisun yhteydessä olevan Satoshi Nakamoton kirjoitelman mukaan motiivina Bitcoinin luomiseen oli vuosien 2007 – 2009 tapahtuneen finanssikriisin aiheuttama suuttumus. Nakamoton mukaan juuriongelma tavanomaisissa valuutoissa on kaikki se luottamus, mitä ne tarvitsevat toimiakseen. Käyttäjien on luotettava, ettei keskuspankki heikennä valuuttaa, mutta Fiat-rahan historia on täynnä tämän sen luottamuksen loukkauksia. (Feuer 2013.)

Seuravana askeleena historiassa on itse lohkoketjun löytäminen. Bitcoinin julkaisun jälkeen ei mennyt kauaa kun huomattiin, että sen pohjalla olevaa lohkoketjutekniikkaa voidaan sitä muokkaamalla käyttää virtuaalisten valuuttojen lisäksi muunkinlaisiin järjestelmiin. (Gupta 2017.) Vuonna 2013 Yhdysvalloissa haettiin 20 lohkoketjuun liittyvää patenttia ja vuonna 2016 hakemuksia lähetettiin jo 324 kappaletta (kuvio 1). (PatentlyGerman 2018.)



Kuvio 1. Lohkoketjupatenttien määrän kehitys. (PatentlyGerman 2018.)

Lohkoketjutekniikan käytön eriydyttyä bitcoinista, keksittiin seuraavana merkittävänä innovaationa ”smart contracts” eli älysopimukset, Ethereum-nimisen seuraavan sukupolven lohkoketjujärjestelmän muodossa. Ethereumissa suoraan lohkoketjuun oli rakennettu pieniä ohjelmia, jotka mahdollistivat rahainstrumenttien kuten lainojen ja joukkovelkakirjojen käytön pelkän Bitcoinin kaltaisen valuutan sijaan. (Gupta 2017.)

Tänä päivänä lohkoketjuja käytetään mahdollistamaan lukuisten erilaisten järjestelmien toimintaa ennennäkemättömällä tavalla ja lohkoketjun hyödyntäminen uusissa innovaatioissa jatkaa alati kasvuaan.

2.2 Lohkoketjun käsite

Bitcoinin myötä keksitty lohkoketjutekniikka mahdollistaa verkossa ennestään mahdottomien tai äärimmäisen vaikeasti toteutettavien ominaisuuksien luomisen. Lohkoketjun nerokkuuden taustalla piilee matemaattiset algoritmit, joiden avulla voidaan luotettavasti ratkaista tunnetut ongelmat tietotekniikan ja matematiikan saralla, kuten ”Double spend problem**”** (digitaalisen rahan käyttö kahteen kertaan), sekä **”**Byzantine generals problem**”** (bysanttilaisten kenraalien ongelma). Edellä mainitut ovat olleet rajoitteina mm. verkossa toimivien digitaalisten maksujärjestelmien luomisessa. (Viitala 2016.)

Syvimmillään lohkoketjun käsite ei kuitenkaan nojaa yksinomaan Bitcoinin tapaan käyttää sitä, vaan sen taustalla olevaan tekniikkaan. Lohkoketju on pohjimmillaan digitaalinen hajautettu tietokanta, eli julkinen rekisteri kaikista järjestelmän käyttäjien kesken tapahtuneista transaktioista tai tapahtumista, jossa jokainen tieto on varmistettu yhteisymmärryksessä (konsensus, engl. consensus) osallistujien kesken, ja jossa rekisteriin syötetyn tiedon muuttaminen jälkeenpäin huomaamatta ei ole mahdollista. (Laurence 2017.)

Lohkoketju on rekisteri, joka sisältää luotettavan ja varmennettavan luettelon jokaisesta sinne syötetystä transaktiosta tai tapahtumasta, aina ensimmäiseen merkittyyn tapahtumaan saakka. (Crosby, Nachiappan, Pattanayak, Verma & Kalyanaraman 2016.)

Lohkoketju mahdollistaa sen, että hajautettu verkko itsenäisiä osallistujia voi hallita rekisteriä luotettavasti poistaen tarpeen minkään keskeisen tahon valtuuttamiseksi rekisterin hallintaan. (Laurence 2017.)

Alla lueteltuna on pähkinänkuoressa lohkoketjun ominaisuudet, myös joiden ansioista jotkut pitävät sitä jopa yhtenä tärkeimmistä keksinnöistä internetin ja sähkön jälkeen: (Metry 2017.)

- Järjestelmän hajautus; ei jää yhtä hallinnoivaa tahoa, jonka toimintaan tai rehellisyyteen luottaa. Jäljelle jää kaikille verkon osallisille jaettava yhteinen totuus, eikä järjestelmän toiminta ole yhden heikon kohdan varassa, vaan jaettu sen käyttäjille.

- Tallennetun tiedon eheys ja pysyvyys; tietoa ei voida väärentää huomaamatta. Tieto tallennetaan tavalla, jonka ansiosta sitä ei voida huomaamattomasti muokata eikä poistaa enää jälkeenpäin.

- Konsensusmekanismi; tallennettavalle tiedolle saavutetaan hajautetussa, itsenäisten tietokoneiden muodostamassa verkostossa kryptografiaa hyödyntämällä yhteisymmärrys ja luottamus tiedon aitoudesta. Bitcoinin käyttämä protokolla konsensuksen saavuttamiseen on ”Proof-of-work” protokolla. (Nakamoto 2008.)

Lohkoketju luo pysyvän rekisterin kaikista järjestelmän historian tapahtumista. Kuitenkin koska mikään ei ole ikuista, pysyvyys pohjautuu tietenkin järjestelmää ylläpitävän verkon pysyvyyteen. Kun tieto on tallennettu lohkoketjuun, sitä on äärimmäisen vaikea muokata tai poistaa sieltä. Lisättäessä tietoa rekisteriin, järjestelmän osallistujien täytyy varmentaa tiedon oikeellisuus yhteisymmärryksessä. (Laurence 2017.)

Lohkoketjun ominaisuuksista myös hajauttamisen ymmärtäminen on tärkeää. Se, että lohkoketju on hajautettu tarkoittaa sitä, että järjestelmästä vastaa yhden hallinnoivan tahon sijaan suuri verkko itsenäisiä osallistujia, joita kutsutaan usein termillä ”node”. Nodet ovat tietokoneita, jotka ajavat lohkoketjun toiminnan takana olevaa algoritmia. Koska järjestelmän toiminnan mahdollistavat tietokoneet eivät ole yhdessä paikassa, myöskään järjestelmään ei synny yhtä kriittistä heikkoa kohtaa, parantaen sen virhealttiutta. (Viitala 2016.)

Nämä verkostoituneet itsenäiset tietokoneet käyttävät laskentatehoa ja sähköä ylläpitäessään järjestelmän toimintaa ja esimerkiksi bitcoin-verkossa niille maksetaan työstä palkkana Bitcoineja. Tätä rahaa vastaan laskentatehon valjastamista verkon ylläpitämiseen kutsutaan myös ”kaivamiseksi”, (engl. mining). (Laurence 2017.)

Lohkoketjuja on kuitenkin myös erityyppisiä ja niiden ominaisuudet voivat poiketa toisistaan, vaikka ne pääpiirteittäin jakavatkin yhteiset toimintaperiaatteet;

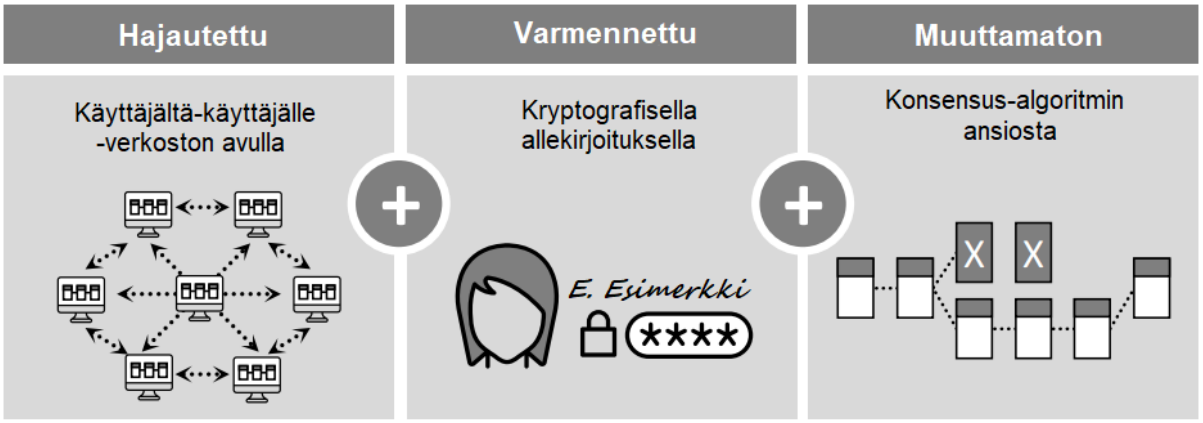
- **Julkiset lohkoketjut**; muun muassa kuten Bitcoin. Julkinen lohkoketju toimii suuren hajautetun käyttäjäverkon voimin, ja siihen osallistuminen, sen käyttö sekä sen lähdekoodi ovat avoinna kaikille. Käyttää järjestelmässä omaa sisäistä rahayksikköä, jota nodet saavat palkaksi verkon ylläpitämisestä.

- **Luvanvaraiset lohkoketjut:** Luvanvaraisessa lohkoketjussa sen omistava taho voi määritellä hajautetun verkon käyttäjille rooleja, joiden velvoitteiden sekä oikeuksien mukaan he voivat verkossa toimia. Käytetään myös sisäistä rahayksikköä verkkoa ylläpitäville nodeille maksamiseksi. Lähdekoodi voi olla avoin tai suljettu.

- **Yksityiset lohkoketjut:** Yksityiset lohkoketjut ovat usein kooltaan pienempiä, ja verkostoon osallistuvia hallitaan tarkasti. Yksityiset lohkoketjut ovat yleisiä yksityisten organisaatioiden välisissä järjestelmissä, joissa jaetaan luottamuksellista tietoa. Yleensä ei käytössä järjestelmän sisäistä rahayksikköä. (Laurence 2017.)

2.3 Bitcoinin toimintamekaniikka

Vaikka eri lohkoketjujärjestelmien sisältämät toiminnallisuudet voivat erota toisistaan, ovat perusperiaatteet kuitenkin lähellä toisiaan (kuvio 2). Esimerkkinä lohkoketjun mekaniikasta voidaan käyttää Bitcoinia sen ollessa ensimmäinen lohkoketjulla toimiva järjestelmä.



Kuvio 2. Lohkoketjun ominaisuudet. (Hackius & Petersen 2017.)

Bitcoin on suoraan käyttäjältä käyttäjälle toimiva elektroninen valuuttajärjestelmä, jota ei hallinnoi mikään keskitetty taho kuten keskuspankki. Bitcoinin ollessa virtuaalinen valuutta, lohkoketju on sen toiminnan mahdollistava moottori sekä tietynlainen kirjanpitojärjestelmä. Bitcoinin lohkoketjuun on tallennettu merkintänä historian jokainen järjestelmän käyttäjältä toiselle tapahtunut rahansiirto, eli transaktio. Tiedon tallentaminen tapahtuu kryptografian turvin tavalla, jonka ansiosta käyttäjät pystyvät luottamaan toisiinsa sekä tallennetun tiedon eheyteen ilman tarvetta keskitetylle taholle, joka varmistaa järjestelmän luotettavuuden ja toiminnan. (Nakamoto 2008.)

2.3.1 Hajautettu osallistujien verkosto

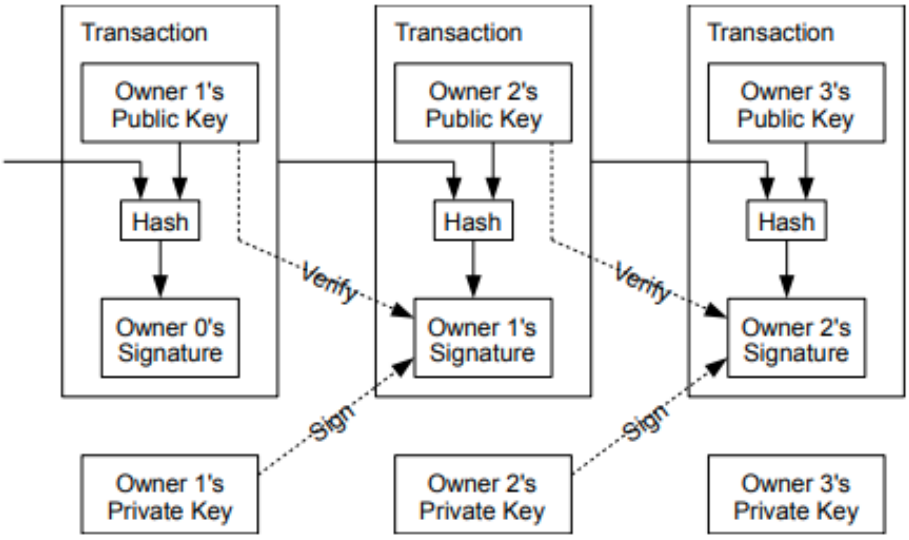
Lohkoketju on yksinkertaistettuna tietokanta tai tapahtumakirja, joka on hajautettu verkostolle itsenäisiä osallistujia. Yksittäistä verkon osallistujaa kutsutaan englanniksi termillä ”node”. Verkoston osallistujat, eli nodet ovat algoritmia ajavia tietokoneita, joilla kaikilla on hallussaan sama rekisteri kaikista koskaan lohkoketjuun tallennetuista tapahtumista. (Laurence 2017.)

Nodet voivat sijaita kaikkialla ympäri maailman, sellaista voi operoida kuka tahansa. Noden ylläpitäminen on työlästä ja kallista, joten koko järjestelmän toimintaa ajava algoritmi kätkee sisäänsä toiminnon, jolla nodeille maksetaan palkkiota palveluksestaan. Palkkio on yleensä järjestelmän oma raha, kuten esimerkiksi Bitcoinin järjestelmässä palkkiona työstään nodet saavat Bitcoineja. (Laurence 2017.)

2.3.2 Käyttäjien väliset transaktiot

Bitcoinin tapauksessa lohkoketjuun tallennetaan tietoja käyttäjien välillä tapahtuvista transaktioista. Miten kuitenkaan voimme luottaa siihen, ettei kirjanpitoon lisättävä merkintä transaktiosta ole väärä?

Bitcoinin ratkaisu ongelmaan perustuu julkisen avaimen salaukseen (kuvio 3). Rahan omistaja lähettää rahaa seuraavalle siten, että lähetettävän kolikon edellisen siirtotapahtuman tiiviste ja seuraavan omistajan julkinen avain allekirjoitetaan lähettäjän omalla salaisella avaimellaan ja allekirjoitus lisätään lähetettävän kolikon tietoihin. (Nakamoto 2008.)



Kuvio 3. Bitcoin-transaktion allekirjoitusmekaniikka. (Nakamoto 2008.)

Täten maksun saaja voi varmentaa allekirjoituksen, ja näin ollen myös kolikon ”omistajuuden ketjun”. Ongelmana kuitenkin vielä on, että maksun saaja ei voi varmentaa sitä, etteikö joku edellisistä omistajista olisi käyttänyt kyseistä rahaa jo kertaalleen. Yleisen käytännön mukaan tässä kohtaa tulisi kyseeseen keskeinen luotettu taho, joka varmentaisi, ettei rahaa olisi käytetty aiemmin. Bitcoinin tavoitteena on kuitenkin nimenomaan poistaa tarve tällaiselle keskeiselle auktoriteetille järjestelmässä. (Nakamoto 2008.)

Jotta voidaan varmistua, ettei rahaa ole käytetty kertaalleen, tulee olla tietoinen kaikista aiemmista tapahtumista. Bitcoinin ratkaisu tähän on, että jokainen käyttäjien välinen transaktio julkaistaan kaikille verkon osallistujille eli nodeille ja jokaisella nodella on täydellinen kopio kaikista ennestään tapahtuneista transaktioista. (Nakamoto 2008.)

Nodet taas toimivat järjestelmässä, joka mahdollistaa sen, että kaikki siihen osallistuvat voivat olla yhtä mieltä transaktioiden järjestyksestä ja historiasta. Järjestelmän on näin tarkoitus antaa maksunsaajalle todiste, ettei hänen saamaansa rahaa ole käytetty jo ennestään. (Nakamoto 2008.)

Tätä osallistujien verkon yhteisen hyväksynnän mahdollistavaa mekanismia kutsutaan konsensusmekanismiksi (engl. consensus protocol). Mekanismi mahdollistaa sen, että kaikki verkon osapuolet voivat varmistua siitä, että tietokanta on ajantasainen ja sinne tallennetut tiedot ovat kaikkien verkon osapuolten hyväksymiä. (Viitala 2016.)

2.3.3 Lohkot ketjussa

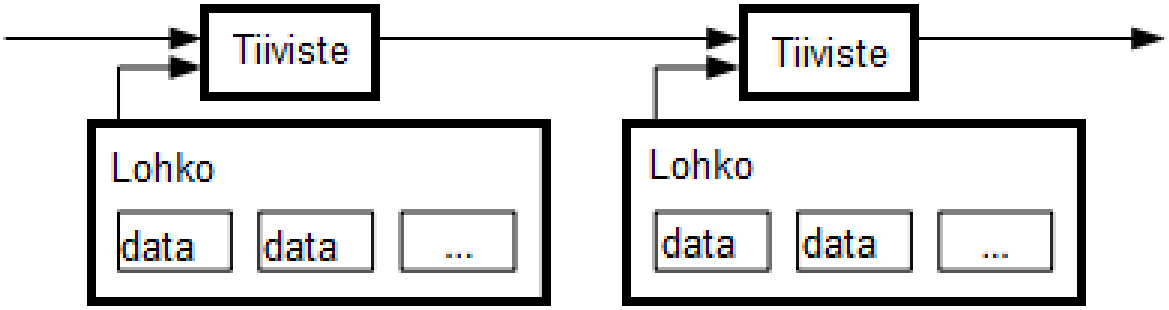
Nimi ”lohkoketju” juontaa juurensa malliin, jolla tieto on siihen tallennettu. Tieto nimittäin tallennetaan ”lohkoihin” ja jokainen lohko on tallennettu ”ketjuun” peräjälkeen kiinni toisiinsa kronologisessa järjestyksessä. (Laurence 2017.)

- **Lohko** on tietynlainen säilö, joka pitää sisällään rekisteriin kirjattuja tapahtumia. Lohkojen määritelty koko, ajanjakso sekä laukaisevat tekijät lohkon lisäämiseksi ketjuun, vaihtelevat järjestelmittäin. Yksi lohko sisältää siis useita toisistaan erillisiä tietoja ja sitä voisi ajatella vaikka kokonaan omana listanaan kirjattuja tapahtumia.

- **Ketju** on lohkojen välillä oleva yhdistävä tekijä, joka sitoo lohkot matemaattisesti toisiinsa. Tämä toiminnallisuus on syy siihen, että tieto pysyy luotettavana. Yksinkertaistettuna ketjuun liitettävälle lohkolle muodostetaan oma sormenjälkensä, josta sen tunnistaa muista ketjun lohkoista. Lohkolle luotava sormenjälki koostuu kaikesta kyseisen lohkon sisältämästä tiedosta sekä lisäksi edellisen lohkon sormen-

jäljestä. Sormenjälki siis toimii sen uniikkina tunnisteena ja lukitsee sen myös ketjuun paikalleen. (Laurence 2017). Kun tämä prosessi toistuu, on jokaisen lohkon sormenjäljen muodostamiseen siis käytetty myös edellisten lohkojen sormenjälkiä.

Bitcoinin ratkaisussa transaktioita sisältävästä lohkosta (engl. block) lasketaan kryptografisen hash-funktion avulla tiiviste, joka sisältää myös aikaleiman. Tiivisteessä oleva aikaleima todistaa, että kyseiset tiedot sisältävä kyseinen lohko on ollut kyseisessä muodossa olemassa aikaleimattuun aikaan. Tiedot sisältävä lohko saadaan liitettyä ketjuun siten, että ensimmäisestä lohkosta lähtien, uuden lohkon sormenjäljeksi luotavaan tiivisteeseen lisätään lohkon sisältämien tietojen lisäksi mukaan myös edellisen ketjussa olevan lohkon tiiviste kuvion 4 mukaisesti. (Nakamoto 2008.) Näin toimimalla jokaisessa ketjuun lisätyn lohkon tiivisteessä on tavallaan yhteen liitettynä myös järjestyksessä kaikkien edellisten lohkojen tiiviste.



Kuvio 4. Lohkojen ketjuttaminen tiivisteen avulla. (Nakamoto 2008.)

Bitcoinissa lohkon tiiviste lasketaan kryptografisella ”SHA-256” hash-funktiolla. Funktio laskee syötetystä tiedosta ulos tietyn mittaisen tiivisteen. Olennaista funktiossa on se, ettei luodusta tiivisteestä kykene laskemaan enää ”takaisinpäin” ja siten saada sen luomista varten syötettyä tietoa. (Mycryptopedia 2018.)

Esimerkiksi merkkijonosta ”A, B, C, D” laskettu sha-256 tiiviste olisi:
”4AA3EDAC2D0393ABDC2BABB0D8AD0EF9F5F6FF02CEB2A230BE2C6D5E3FCEE6C2”.
Jos muuttaisimme syötettyä merkkijonoa vähänkään, tiiviste muuttuu kokonaan.
Esimerkiksi merkkijonon ”a, b, c, d” laskettu tiiviste eroaa jo olennaisesti alkuperäisestä:
”85641E641B2BB9D4EAD1F84ADD137D48099AC71F24DA8B24BB8DE05F5FAF8B91”.
Tiivisteen laskemiseen käytettiin sha256 web-työkalua osoitteessa passwordsgenerator.com.

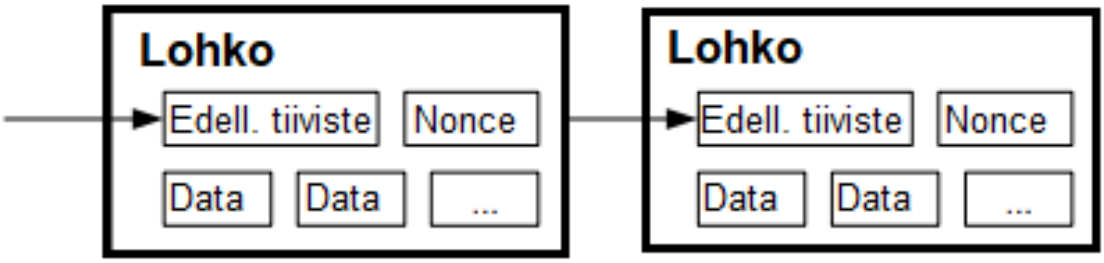
Koska jokaisen ketjussa olevan lohkon tiivisteen laskemiseen on käytetty myös edellisen lohkon tiivistettä, muuttamalla tietoa yhdessäkään aiemmassa lohkossa, muuttaa koko sitä seuraavan lohkoketjun viimeisintä tiivistettä. Näin ollen mitään tietoa ei voida muuttaa muiden osallisten sitä huomaamatta.

2.3.4 Konsensusmekanismi

Lohkoketjun osallistujalta-osallistujalle (peer-to-peer) -toimintamalli vaatii sen, että jokaisen verkon osallisen tulee voida varmistua siitä, että rekisteriin lisättävät tapahtumat ovat oikeita, eikä esimerkiksi lähetettävää rahaa ole jo lähetetty aiemmin muualle. Yksinkertaistettuna tämä tarkoittaa, että kaiken lohkoketjussa olevan tiedon tulee saada koko verkoston yhteinen hyväksyntä eli saavuttaa sille konsensus. (Laurence 2017.)

Lohkoketjujärjestelmän ollessa hajautettu verkostolle itsenäisiä, algoritmia ajavia tietokoneita eli nodeja, on tärkeää, että ne kaikki toimivat verkostossa yhteisesti sovitulla tavalla. Tässä tuleekin kyseeseen bysanttilaisten kenraalien ongelma; miten hajautetut, toisistaan riippumattomat tahot voivat luottaa toisiinsa siinä, ettei joku tahoista muuttaisi sovittua toimintatapaa salassa muilta omaksi hyödykseen (Campbell 2015.). Bitcoinin tapauksessa konsensus saavutetaan proof-of-work -järjestelmällä, jonka Satoshi Nakamoto kertoo saaneen vaikutteita Adam Back:n luomasta ”Hashcash”-järjestelmästä. (Nakamoto 2008.)

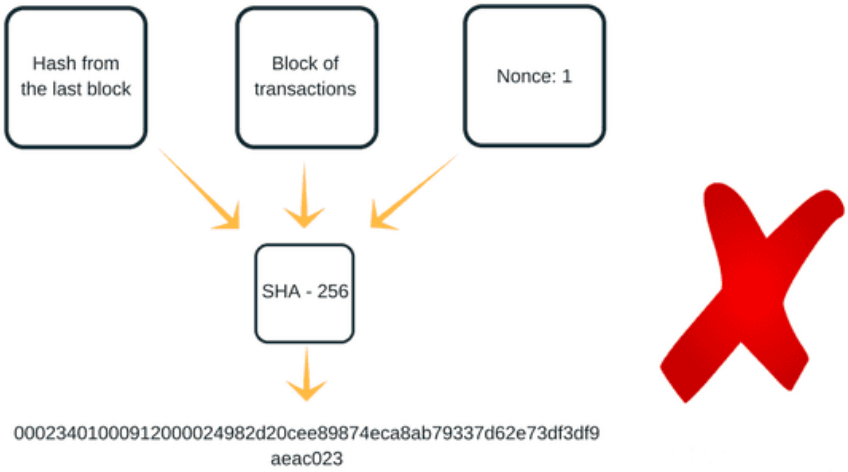
Bitcoin käyttää tiivisteiden (hash) laskemiseen ”SHA-256” -tiivistemetodia siten, että lohkosta lasketun tiivisteen tulee omata tietty määrä merkkijonon ensimmäisistä merkeistä numeroita 0. Kuviossa 5 kuvattua lohkon sisältämää ”nonce” attribuuttia, bitcoinin tapauksessa satunnaista kokonaislukua arvataan ja tiivistettä lasketaan yhä uudelleen ja uudelleen niin kauan, kunnes tiivisteen alussa on haluttu määrä nollia. (Nakamoto 2008.)



Kuvio 5. Lohkojen sisältö. (Nakamoto 2008.)

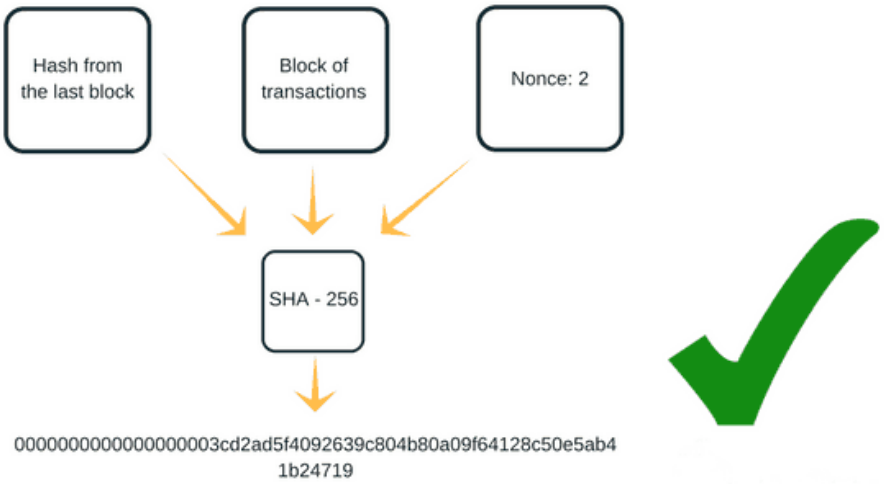
Tätä haluttua nollien määrää tiivisteen laskennassa Satoshi Nakamoto kutsuu järjestelmän ”vaikeustasoksi”, tai englanniksi ”difficulty”. Vaikeustasoa nostetaan suhteessa koko verkon yhteenlasketun laskentatehomäärään sen mukaisesti, että lohkoja saadaan lasket-

tua haluttu määrä tunnissa. Vaikeustaso siis nousee sen mukaan, mitä enemmän verkostossa on nodeja. Mitä useampi nolla lohkon tiivisteen alussa tulee olla, sitä enemmän laskentatehoa sen laskemiseksi vaaditaan, sillä jokaista arvattua satunnaista numeroa kohden tulee laskea lohkon tiiviste uudelleen. Nollien määrän lisääntyminen nostaa vaadittua laskentatehoa eksponentiaalisesti. (Nakamoto 2008.) Esimerkiksi jos vaikeustaso olisi 18, ei tämä satunnaisluvulla ”1” laskettu tiiviste kävisi sen omatessa ainoastaan 3 nollaa merkkijonon alussa (kuvio 6).



Kuvio 6. Epäonnistunut tiivisteen laskeminen. (Asynclabs blog 2018.)

Satunnaislukuja kokeiltaisiin yhä uudelleen ja uudelleen lohkon tiivistettä laskiessa niin kauan, kunnes oikea satunnaisluku tiivisteen laskemiseksi löytyy (kuvio 7).



Kuvio 7. Onnistuneesti laskettu proof-of-workin mukainen tiiviste. (Asynclabs blog 2018.)

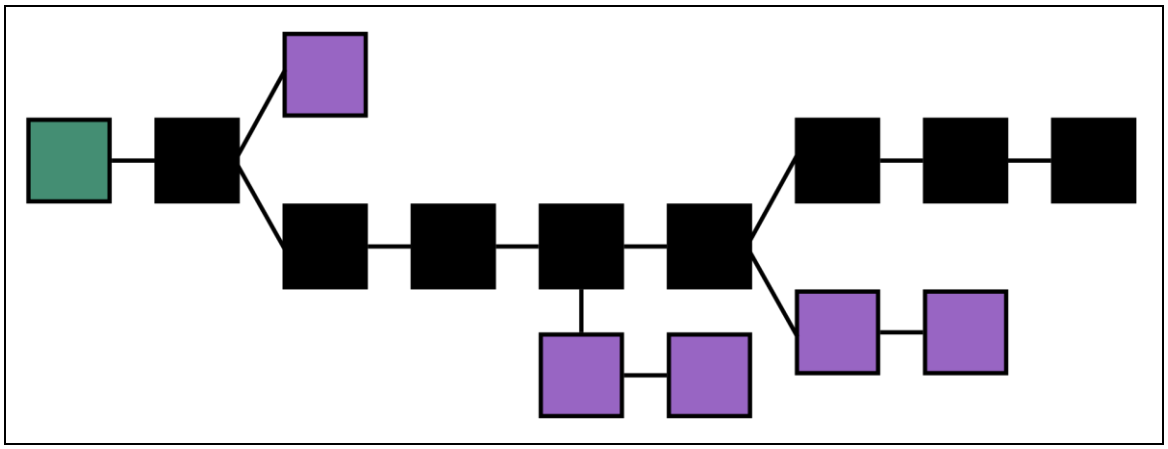
Kun haluttu määrä nollia on saavutettu ja laskentatehon käytön määrä kattaa halutun proof-of-work:in, lohkoa ei voida enää muuttaa tekemättä työtä uudelleen. Kun myöhemmin lohkoja lisätään ketjuun, aiemman lohkon sisältämien tietojen muuttamiseksi joutuisi laskemaan myös jokaisen kyseistä lohkoa seuraavan lohkon tiivisteet uudelleen. (Nakamoto 2008.)

Hyökkääjän halutessa muuttaa jonkin aiemman lohkon tietoja, tulisi sen kaikkien seuraavien lohkojen uudelleentyöstämisen lisäksi tehdä se nopeammin, kuin koko muu verkosto laskee ja liittää uusia lohkoja ketjuun. Tämä vaatisi yksittäisen hyökkääjän omaavan 51 % koko verkoston laskentatehosta, joten on erittäin epätodennäköistä, että näin käy. Verkoston nodet voivat olettaa pisimmäksi kasvaneen ketjun viimeisimmän lohkon olevan enemmistön päätöksellä hyväksytty, koska siihen on käytetty myös eniten laskentatehoa. (Nakamoto 2008.)

Bitcoinin verkosto toimii seuraavassa järjestyksessä:

1. Uudet, kirjattavat transaktiot julistetaan verkoston nodeille
2. Jokainen node kerää tapahtumat lohkoon
3. Jokainen node työskentelee saavuttaakseen proof-of-work:in lohkolleen.
4. Kun joku nodeista saavuttaa proof-of-workin lohkolleen (eli löytää oikean nonce-numeron tiivisteen laskemiseksi), se julistaa lohkon muille verkon nodeille.
5. Nodet hyväksyvät lohkon, mikäli kaikki sen sisältämät transaktiot ovat oikeellisia, eikä siirrettäviä rahoja ole jo siirretty entuudestaan.
6. Nodet osoittavat hyväksyntänsä lohkolle alkamalla työstämään seuraavaa lohkoa ketjuun, käyttäen tämän aiemman hyväksytyn lohkon tiivistettä osana uudesta lohkosta laskettavaa tiivistettä. (Nakamoto 2008.)

Nodet uskovat aina pisimmäksi kasvaneen ketjun olevan oikea ja ne jatkavat uusien lohkojen työstämistä siihen. On kuitenkin mahdollista, että kaksi eri nodea päätyvät julistamaan verkostolle samanaikaisesti eriävät versiot uudesta lohkosta, jolloin ketju haarautuu. Kuviossa 8 vihreä neliö kuvastaa ketjun ensimmäistä lohkoa, mustat lohkot muodostavat pääketjun ja violetit lohkot ovat viallisia lohkojen haarautumia ketjun ulkopuolella. Jotkin nodeista voi saada tiedon väärennetystä tai viallisesta lohkosta ennen oikeaa, jolloin alkaessa työstämään seuraavaa lohkoa väärän lohkon perään, node tallentaa ketjujen haarautuvan kohdan siltä varalta, että toisesta ketjusta tuleekin myöhemmin pidempi. (Nakamoto 2008.)



Kuvio 8. Haarautumia ketjussa.

Vaikka jotkin nodeista alkaisivatkin työstämään epärehellisen noden ilmoittamaa lohkoa, jos ei kyseisen lohkon työstämiseen ole valjastettu 51 % koko verkon laskentatehosta, tulee sen perään muodostuva ketju jäämään jälkeen oikeellisesta pääketjusta. Pidemmän ketjun huomatessaan, väärennetyn lohkon perään lohkoa työstänyt node siirtyy työskentelemään takaisin pisimmän ketjun viimeisimpään ilmoitettuun lohkoon, koska siihen on käytetty yhteensä eniten laskentatehoa. (Nakamoto 2008.)

3 Esimerkkejä lohkoketjujen käytöstä

Sen jälkeen, kun Bitcoinin pohjalta löydettiin sen järjestelmän toiminnan mahdollistanut lohkoketjuteknologia, on lohkoketjua alettu käyttää moniin muihinkin eri tarkoituksiin.

Alla luetellaan esimerkkeinä Bitcoinin-virtuaalivaluutan lisäksi muitakin tapoja ja tarkoituksia lohkoketjujen soveltamiselle. Käsitellyistä järjestelmistä saa osviittaa kaikesta siitä, mihin lohkoketjuja voidaan käyttää mielikuvituksen ollessa ainoana rajana. Koska lohkoketju on loistava tekniikka järjestelmän hajauttamiseen sekä tiedon tallentamiseen varmennettavasti, sillä on erittäin suuri potentiaali tulevaisuudessakin.

3.1 Ethereum

Ethereum -alustan julkaisu vuonna 2016 toi lohkoketjuteknologian seuraavalle tasolle. Ethereumin tarkoituksena oli kehittäjiensä mukaan luoda vaihtoehtoinen protokolla hajautettujen sovellusten rakentamiseksi tarjoamalla erilaisia ratkaisuja, joiden he uskovat olevan hyödyllisiä suurelle määrälle hajautettuja sovelluksia, painottaen erityisesti tilanteita joissa nopea kehitysaika, turvallisuus, sekä erilaisten sovellusten tehokas toimiminen keskenään ovat tärkeitä. (Buterin 2013.)

Käytännössä Ethereum-järjestelmä on abstrakti, pohjalla oleva lohkoketjulla toimiva alusta, jonka avulla kuka tahansa voi luoda älysopimuksia ja hajautettuja sovelluksia nopeasti, yksinkertaisesti, modulaarisesti, universaalisesti sekä ketterästi. Ethereum-lohkoketjun oman valuuttayksikön nimi on Ether. (Buterin 2013.)

3.2 Älysopimukset

Mikä älysopimus sitten oikein on? Älysopimukset ovat kuin tavallisia kahden tahon välisiä sopimuksia, mutta siirrettynä toimimaan lohkoketjussa.

Älysopimusten kolme avainominaisuutta ovat:

- Ne toimivat lohkoketjussa, joten ne tallennetaan julkiseen kirjanpitoon, josta niitä ei voi enää muuttaa jälkikäteen.
- Älysopimuksen määrittelemät transaktiot prosessoidaan lohkoketjussa, eli ne voivat tapahtua automaattisesti ja turvallisesti ilman tarvetta kolmannelle osapuolelle.
- Määritellyt transaktiot tapahtuvat ainoastaan silloin, kun sopimuksen ehdot täyttyvät. Koska mukana ei ole kolmatta osapuolta, ei ole myöskään ongelmia luottamuksessa. (Laura 2020.)

Ethereum mahdollisti ensimmäisenä älysopimusten toimimaan alustallaan ja ne kirjoitetaan käyttämällä Ethereumin omaa ohjelmointikieltä. (Buterin 2013.) Esimerkkinä älysopimuksesta voi toimia vaikka asunnon myyminen. Se missä normaalilla sopimuksella kaupat tehtäessä vaaditaan useita luotettuja välikäsiä kaupan myymiseen sekä vahvistamiseen, voidaan se älysopimuksella karsia.

Kaikkein yksinkertaisimmillaan älysopimus voi olla esimerkiksi ”Kun Herra A maksaa Rouva B:lle 300 etheriä, Herra A saa talon omistajuuden.” Kun sopimus on tallennettu, ei sitä lohkoketjusta voida enää muuttaa. Herra A voi siis turvallisin mielin maksaa Rouva B:lle 300 etheriä talosta, koska lohkoketjuteknologia mahdollistaa luottamuksen heidän välillä. (Laura 2020.)

Älysopimuksien käytölle löytyy lukemattomia mahdollisuuksia. Niitä käytetään jo muun muassa finanssi- ja pankkialalla, luottoyhtiöissä sekä vakuutusalalla, käytön laajetessa muidenkin alojen keskuudessa kokoajan enenevässä määrin. (Laura 2020.)

3.3 Hajautetut sovellukset, ”Dapps”

Hajautetut sovellukset (engl. Decentralized applications, ”Dapps”) eroavat tavallisista sovelluksista siinä, että se toiminnasta vastaa normaalin keskitetyn järjestelmän sijaan hajautettu verkko itsenäisiä tietokoneita eli nodeja.

Lohkoketjussa toimivan hajautetun sovelluksen vahvuuksiin lukeutuu:

- Tallennettu data on luotettavaa ja läpinäkyvää, koska se on lohkoketjussa.
- Järjestelmän toiminta on hajautettu, joten siihen ei jää yksittäistä heikkoa kohtaa. (Raval 2016.)

Esimerkki hajautetusta sovelluksesta on Gems -niminen sosiaalinen viestintäsovellus, jonka tarkoituksena on luoda ”reilumpi” toimintamalli, kuin WhatsApp. Gems:llä on oma valuutta, jota käyttäen mainostajat voivat maksaa suoraan sovelluksen käyttäjälle tämän datasta poistaen rahallisesti hyötyvän välikäden. (Raval 2016.) Vaikka Ethereum oli ensimmäinen alusta, joka teki hajautettujen sovellusten rakentamisesta helppoa, ei se tarkoita, etteikö hajautettua sovellusta voisi luoda ilman sitä, rakentaen sille kokonaan oman lohkoketjun.

3.4 Hajautettu rahoitus, ”DeFi”

Hajautettu rahoitustoiminta (engl. Decentralized Finance, ”DeFi”) on yksi nopeiten kasvavista julkisten lohkoketjujen käyttökohteista. Se käyttää älysopimuksia helpottaakseen

vakaiden markkinoiden ylläpitämistä, jossa käyttäjät voivat muun muassa lainata rahaa toisilleen ja tuottaen rahallaan korkoa sekä lyödä vetoa ilman keskitettyä koordinoivaa tahoa. (Chainlink 2019.)

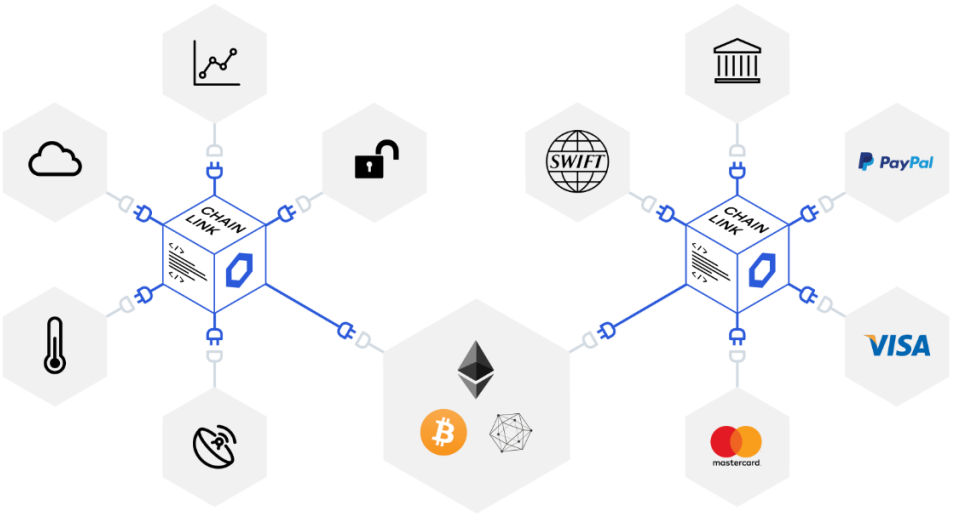
3.5 Oraakkelit ja ChainLink

Jotta älysopimuksia voitaisiin solmia myös lohkoketjun ulkopuolella olevaa dataa käyttäen, tarvitaan tapa tuoda lohkoketjun ulkopuolinen data lohkoketjussa toimivaan muotoon. Vaikeus ulkoisen tietolähteen yhdistämiseksi lohkoketjussa toimivaan älysopimukseen on suurin rajoite sille, kuinka laajasti älysopimuksia voidaan hyödyntää. (Gemini 2020.)

Tässä kohtaa mukaan tulee oraakkelit (engl. oracle). Oraakkeli on oikean maailman ja lohkoketjun välissä toimiva ohjelmisto, joka muuttaa oikean maailman tiedot lohkoketjussa toimivaan muotoon ja takaisin, mahdollistaen lohkoketjun ulkopuolella olevan tiedon yhdistämisen älysopimukseen. (Gemini 2020.)

Ongelmana kuitenkin on, että älysopimusten tarkoitus on mahdollistaa sopimusten hajauttaminen, poistaen keskitetyn heikon kohdan, mutta hajauttamaton välikätenä toimiva oraakkeli luo uuden keskeisen heikon kohdan. Tämä tarkoittaa, että koska tieto kulkee oraakkelin kautta, koko järjestelmä *joutuu luottamaan* yksittäiseen oraakkeliin. Jos oraakkeli on viallinen tai haavoittuvainen, mistä voimme tietää sen läpi kulkevan datan olevan luotettavaa? (Gemini 2020.)

Chainlink on projekti, joka pyrkii ratkaisemaan tämän keskitetyn oraakkelin luoman luottamusongelman. Chainlink kuvailee rakentamaansa ratkaisua ”hajautetuksi oraakkeleiden verkostoksi”. Sen on tarkoitus mahdollistaa lohkoketjun ulkopuolisten resurssien liittäminen lohkoketjuun turvallisesti, käyttäen oraakkeleina toimivia hajautettua verkostoa (kuvio 9). (Ellis, Juels & Nazarov 2017.)



Kuvio 9. ChainLink-oraakkelia havainnollistava kuvitus. (Chainlink 2019.)

Chainlink toimii Ethereum -alustalla ja sillä on oma raha, ”Link”, jota käytetään nodejen operaattoreille maksamiseen. Chainlink -järjestelmää kehitetään jatkuvasti ja sen päämääränä on pystyä yhdistämään älysopimuksiin tarvittavat tietoresurssit luotettavasti lohkoketjujen ulkopuolelta. (Chainlink 2019.)

3.6 Muita käyttökohteita

Lohkoketjujen käyttökohteita keksitään jatkuvasti lisää ja lohkoketjuja käytetään jo nyt mitä mielikuvituksellisempiin innovaatioihin:

- Lohkoketjuteknologia tarjoaa suuren potentiaalin prosessien ja toimintamallien parantamiseksi logistiikassa ja toimitusketjun hallinnassa. Sen ansiosta voidaan mm. helpottaa kauppatavaran seurantaa ja alkuperän jäljittämistä, parantaa kansainvälisen rahtiliikenteen asiakirjojen käsittelyä, kuin myös tunnistaa tuoteväärennöksiä esimerkiksi lääkkeiden toimitusketjuissa. (Hackius & Petersen 2017.)

- Lohkoketjuja voidaan käyttää myös terveydenhuollossa, mm. potilastietojen tallentamiseen siten, että terveydenhuollon ammattilaisilla olisi pääsy niihin, mutta tiedot pysyvät kuitenkin turvassa. Esimerkki tällaisesta järjestelmästä on Gem -nimisen yhdysvaltalaisen startup-yrityksen Gem Health Network -järjestelmä. (Mettler 2016.)

- Vuonna 2018 #MeToo -liikkeen myötä inspiroituneena Hollantilainen startup-yritys julkaisi LegalFling -nimeä kantavan sovelluksen, jonka tarkoitus on tarjota tapa

varmistaa suostumus seksuaaliseen kanssakäymiseen. Tahojen väliset suostumukset tallennetaan lohkoketjuun, joten niitä ei voida myöhemmin muokata. (Belanger 2018.)

- Follow My Vote -järjestelmän tarkoituksena on tarjota keino järjestää äänestyksiä elektronisesti, mutta läpinäkyvästi ja turvallisesti. Inspiraation järjestelmän luomiseksi on tuonut syytökset vilpillisistä vaaleista kautta historian. Järjestelmä tallentaa ihmisten antamat äänet lohkoketjuun, jonka ansiosta voidaan läpinäkyvästi varmistua, ettei vaalivilppiä pääse tapahtumaan. (FollowMyVote 2020.)

4 Lohkoketjun luominen

Jotta voidaan syventyä lohkoketjun toiminnallisuuteen käytännössä, tulee tutkimismenetelmänkin olla kytköksissä toiminnallisuuksiin käytännön tasolla. Kaikkein lähimpänä käytännön tasoa on itse käytäntö, joten menetelmäksi valikoitui lohkoketjun luominen itse.

Lohkoketjun toiminnan periaatteiden tutkimiseksi luodaan siis oma prototyyppi toimivasta digitaalisesta lohkoketjusta Java -ohjelmointikieltä käyttäen. Tässä osiossa luotava lohkoketju määritellään käyttäen hyväksi kerättyä tietoperustaa ja sen periaatteiden mukaisen toiminnan vaatimat toiminnallisuudet jaotellaan erilleen omiksi luotaviksi kokonaisuuksiksi.

Konkreettisen lohkoketjuesimerkin luomiseen käytettäväksi ohjelmointikieleksi valikoitui Java, sillä Javan syntaksi on mielestäni tarpeeksi yksiselitteinen, vaikka se kielenä olisikin ”tönkkö” verrattuna vaikka JavaScriptiin tai Pythoniin. Esimerkiksi JavaScriptissä ja Pythonissa voidaan asettaa arvoja muuttujiin ilmoittamatta muuttujan tietotyyppiä etukäteen, joka tietysti tekee ohjelmoinnista ”joustavampaa”. Koin kuitenkin hyödyllisemmäksi järjestelmän luomisprosessin seuraamisen kannalta Javan yksiselitteisyyden, koska järjestelmäksi konkretisoitavaa tietoperustaa on paljon ja sen käsittely saattaa olla muutenkin haasteellista. Toinen Javan valitsemiseen vaikuttanut asia on oma kokemukseni siitä, että jos osaa Javaa, on muidenkin olio-ohjelmointikielten opetteleminen helppoa. Joten päättelin myös, että jos järjestelmä luodaan Javalla, on sen luominen helppoa myös muillakin ohjelmointikielillä myöhemmin.

Periaatteiden mukaisen toiminnan määrittelemiseksi käytetään Bitcoin-järjestelmää esimerkkinä, koska Bitcoin oli ensimmäinen järjestelmä, joka sovelsi lohkoketjutekniikkaa. Luotavasta esimerkistä kuitenkin jätetään Bitcoinin yleispiirteisten toimintaperiaatteiden ulkopuoliset toiminnot pois, jottei esimerkistä tulisi liian vaikeaselkoinen ja massiivinen.

4.1 Projektisuunnitelma

Lohkoketjuprototyypin luominen tapahtuu seuraavassa järjestyksessä:

1. Määritellään vaatimukset järjestelmälle siten, että se vastaa toiminnaltaan yleispiirteittäin lohkoketjun tietorakennetta.
2. Määritellään lohkoketjun toiminnalliset osat yksittäisiksi toteutettaviksi kokonaisuuksiksi vastaamaan olio-ohjelmoinnin käytäntöjä. Puretaan toteutettava järjestelmä siis yksittäisiin olioihin.

3. Määritellään olioiden toiminnallisuudet ja yhteydet toisiinsa siten, että järjestelmän osat toimisivat halutulla tavalla.
4. Toteutetaan suunniteltu järjestelmä.
5. Testataan järjestelmää ja selvitetään vastaako se sille asetettuihin vaatimuksiin.

4.2 Toteutus ja tuotos

Seuraavaksi aletaan toteuttamaan järjestelmää suunnitelman mukaisesti. Järjestelmään luodut osat ovat dokumentoitu pääsääntöisesti liitteiksi. Lukijan näkökulmasta voi olla helpompaa ymmärtää luomisprosessin tapahtumia seuraamalla liitteitä samanaikaisesti, esimerkiksi tulostamalla liitteet erikseen.

Kokonaan valmis ohjelmakoodi luokkakohtaisesti on dokumentoitu liitteisiin 23 – 36;

- Transaktio-luokka, liitteet 23 ja 24.
- Lohko-luokka, liitteet 25 ja 26.
- Lohkoketju-luokka, liite 27.
- Node-luokka, liitteet 28, 29, 30 ja 31.
- Kryptografia-luokka, liitteet 32 ja 33.
- Main-luokka testaamiseen, liitteet 34, 35 ja 36.

4.2.1 Järjestelmän vaatimukset ja jakaminen osiin

Jotta on mahdollista tutkia lohkoketjun toimintaa ohjelmoidun esimerkin kautta, tulee esimerkkijärjestelmän olla yksinkertainen ja lähdekoodin olla itsensä selittävää.

Koska tarkoituksena on sivuta Bitcoinin tapaa käyttää lohkoketjua tietorakenteena, määritellään, että sen tulee toimia virtuaalivaluutan tavoin. Asetetaan sille järjestelmänä seuraavat vaatimukset Bitcoinin mukaisesti:

1. Sillä tulee olla valuutta, jota käyttäjät voivat lähettää toisilleen ja jolla myös samalla maksetaan verkkoa ylläpitäville nodeille.
2. Käyttäjien väliset transaktiot tulee tallentaa rekisteriin lohkoketjun periaatteen mukaisesti, sekä tavalla jolla varmistetaan, ettei niitä voida muokata tai poistaa.
3. Järjestelmän tulee varmistaa, että käyttäjä ei voi lähettää samaa rahaa useampaan kertaan, eli estää ”Double-Spend problem”.
4. Järjestelmän tulee varmistaa, ettei toinen käyttäjä voi käyttää toisen käyttäjän varoja salaa.

5. Järjestelmään tulee omata proof-of-work protokolla, jolla varmistetaan osallisten välinen konsensus tallennetulle tiedolle.
6. Useamman noden tulee voida käyttää järjestelmää ja tehdä näin prototyypistä ”hajautettu”.

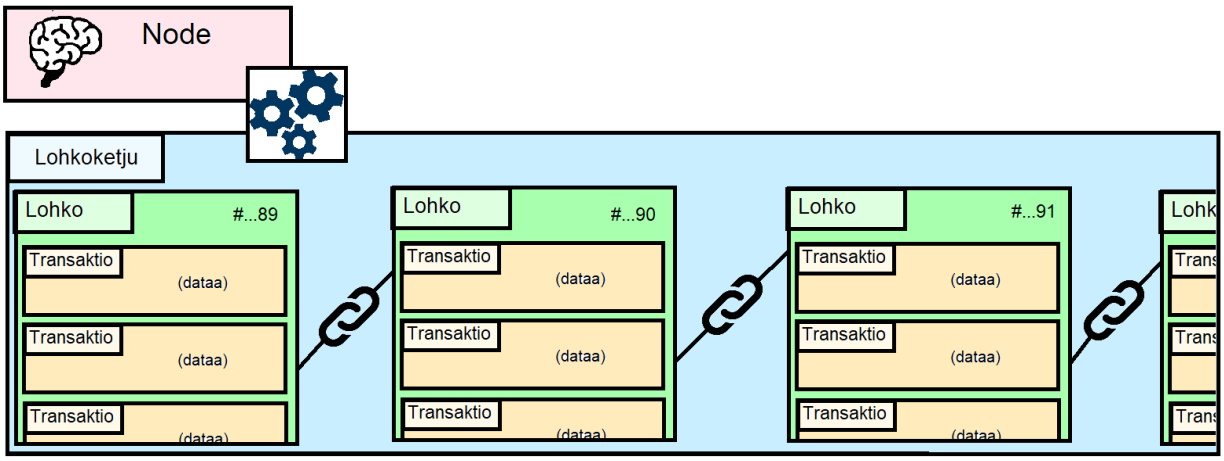
Toteutettavaa lohkoketjua voitaisiin lähteä purkamaan osiin miettimällä aluksi, että mitä tietoa lohkoketjuun tallennetaan. Koska tässä demonstroidaan lohkoketjun toimintaa mukaillen virtuaalivalutta Bitcoinin toimintaa, rekisteriin halutaan tallentaa järjestelmän käyttäjien välisiä transaktioita, eli rahansiirtotapahtumia. Ensimmäiseksi omaksi osakseen voidaan määritellä siis ”**transaktio**”-olio, joka sisältää datan käyttäjien välisestä tapahtumasta.

Lohkoketjun perusperiaatteen mukaisesti rekisteriin lisättävät transaktiot tallennetaan tietorakenteen nimen mukaisesti lohkoihin. Lohkot siis sisältävät listan transaktioita sivun 12 kuvion 5 mukaisesti. Erotellaan siis omaksi osakseen transaktioita sisältävä ”**lohko**”-olio. Nimensäkin mukaisesti tietorakenne koostuu siis ”lohkoista ketjussa”. Koska olemme määritelleet jo transaktio-oliota sisältävän lohko-olion omaksi kokonaisuudekseen, olisi seuraava looginen kokonaisuus tämä ketju. Ketjun tulee siis sisältää lista peräkkäin liitetyistä lohko-olioista. Määritellään siis seuraavaksi omaksi kokonaisuudekseen ”**lohkoketju**”-olio.

Lohkoketju on nyt määritelty olioksi, joka pitää sisällään listan lohko-olioita, jotka kukin taas sisältävät listan käyttäjien välisiä tapahtumia, eli transaktio-oliota. Järjestelmän toiminta vaatii kuitenkin erillisen osansa, joka pitää huolen järjestelmän toiminnasta. Lohkoketjuissa järjestelmän toiminta on jaettu verkolliselle nodeja.
Nodet ajavat siis lohkoketjun omaa ohjelmakoodia yhdessä muiden kanssa. Erotetaan järjestelmästä omaksi osakseen ”**node**”-olio, jonka vastuulla on huolehtia järjestelmän toiminnasta.

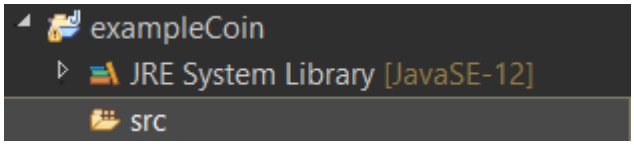
Kuviossa 10 on havainnollistettuna järjestelmän osat, sekä niiden välinen suhde.

- Node on järjestelmän osa, joka suorittaa lohkoketjun vaativat toiminnallisuudet.
- Lohkoketju on olio, joka sisältää listan yhteen liitettyjä lohkoja.
- Lohko on olio, joka pitää sisällään listan transaktio-olioita.
- Transaktio on olio, joka pitää sisällään tiedon käyttäjien välisestä tapahtumasta.



Kuvio 10. Järjestelmän toiminnalliset osat.

Seuraavaksi aletaan määrittelemään ja luomaan järjestelmän osia. Luodaan Eclipse IDE:ssä uusi Java-projekti nimeltä ”ExampleCoin”, johon alamme järjestelmää pala kerrallaan rakentamaan (kuvio 11).



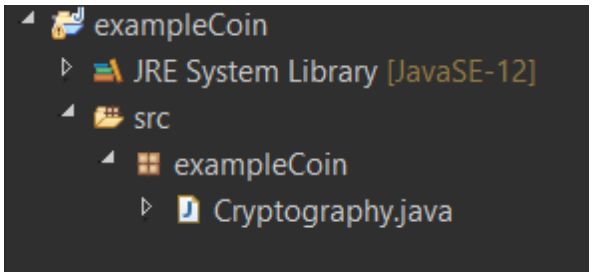
Kuvio 11. Järjestelmän pohja luotuna Eclipsessä.

4.2.2 Käyttäjät ja lompakot

Ensimmäiseksi määritellään, että miten käyttäjät otetaan huomioon järjestelmässä. Kuten kuvio 3 sivulla 9 osoittaa, käyttäjien väliset transaktiot allekirjoitetaan julkisen avaimen salauksella siten, että lähettävä osapuoli allekirjoittaa ilmoitettavasta transaktiosta lasketun tiivisteen omalla salaisella avaimellaan. Tämän julkisen avaimen salauksen toteuttamiseksi käytetään Javan tarjoamaa java.security -kirjastoa, joka tarjoaa suoraan luotavan protokollan tarvitsemat toiminnallisuudet ja luokat.

Kirjastoa hyväksikäyttäen käyttäjä voi generoida itselleen julkisen sekä salaisen avaimen järjestelmää käyttääkseen. Julkinen avain toimii siis käyttäjän ”osoitteena”, johon viitataan, kun rahaa lähetetään eteenpäin ja salainen avain on transaktion allekirjoittamista varten. Bitcoinissa avainpari generoidaan käyttäen elliptisen käyrän salausmenetelmiin kuuluvaa ”ECDSA”-algoritmia ”secp256k1” -käyrällä (Bitcoin Developer 2020).
Java.security -kirjasto mahdollistaa avainparin luomisen kyseisellä metodilla, joten sitä käytetään myös tässä järjestelmässä.

Järjestelmän oikeanlainen toiminta vaatii useita erilaisia kryptografisia menetelmiä, joten projektin juurikansioon alustetaan luokka ”Cryptography.java”, johon tarvittavat kryptografiset metodit asetetaan talteen (kuvio 12).



Kuvio 12. Kryptografia-luokka.

Kryptografia-luokkaan lisätään metodi avainparin, eli java.security -luokan KeyPair objektin generoimiselle, sekä metodit PrivateKey- ja PublicKey -objektien luomiselle avaimen byte-array -muodosta (Mishra 2019). Liitteessä 1 on esitetty lähdekoodi kokonaisuutena.

Simuloidaksemme Bitcoinia, tässä järjestelmässä luotavien tiivisteiden merkkijonot ovat heksadesimaalisessa muodossa. Kryptografia-luokkaan lisätään siis metodit byte-arrayn muuttamiseksi heksadesimaaliseksi merkkijonoksi, sekä heksadesimaalisen merkkijonon muuttaminen takaisin byte-array -muotoon (liite 2).

Koska tulemme käyttämään järjestelmässä avaimia heksadesimaalisessa merkkijonomuodossa, luomme myös metodin KeyPair-objektin luomiseksi suoraan siitä (kuvio 13).

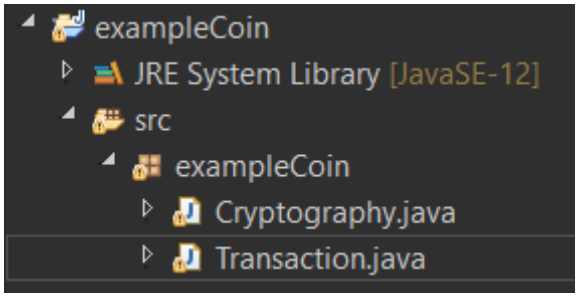
```
74    public static KeyPair generatePairFromHex(String publicHex, String privateHex) throws Exception {
75
76        PublicKey pu = generateFromPublic(hexStringToByteArray(publicHex));
77        PrivateKey pr = generateFromPrivate(hexStringToByteArray(privateHex));
78
79        return new KeyPair(pu, pr);
80
81    }
82
```

Kuvio 13. Metodi avainpari-olion luomiseksi heksadesimaalisista merkkijonoista

Nyt järjestelmässä on kryptografia-luokka, joka sisältää tarvittavat toiminnallisuudet julkisen avaimen salauksen käyttämiselle.

4.2.3 Transaktio

Koska nyt järjestelmässä voidaan varmentaa käyttäjien välinen transaktio julkisen avaimen salausmenetelmällä, määritellään seuraavaksi itse transaktio-olio. Kuten Bitcoinissa, tulee transaktion sisältää tiedot käyttäjien välisistä transaktioista. Luodaan projektin juurikansioon siis uusi luokka ”Transaction.java” (kuvio 14).



Kuvio 14. Transaktio-luokka projektissa.

Transaktion on tarkoitus sisältää tieto käyttäjien välisestä tapahtumasta. Transaktioon siis talletetaan rahaa lähettävän käyttäjän julkinen avain, eli ”lompakon osoite”, vastaanottajan julkinen avain, sekä lähetettävä rahamäärä. Transaktiosta tulee voida laskea tiiviste, jonka lähettäjä allekirjoittaa, joten oliossa tulee myös olla paikka tälle. Allekirjoituksen avulla transaktion oikeellisuus voidaan varmentaa.

Jotta transaktiolle saadaan laskettua tiiviste, oliolle määritellään metodi calculateHash(). Tiiviste lasketaan Bitcoinin tapaan kryptografisella SHA-256 -algoritmilla, jonka toimittava metodi voidaan siis hyvin sijoittaa Kyrptografia-luokkaan (liite 3).

Tiiviste lasketaan transaktion osapuolista, summasta, sekä aikaleimasta (kuvio 15). Transaktion tiiviste allekirjoitetaan lähettävän käyttäjän toimesta, joten oliolle määritellään metodi tiivisteen allekirjoittamiselle (kuvio 16).

```
public class Transaction {

    private String fromAddress;
    private String toAddress;
    private int amount;
    private String signature;
    private String hash;

    public Transaction(String fromAddress, String toAddress, int amount) {
        super();
        this.fromAddress = fromAddress;
        this.toAddress = toAddress;
        this.amount = amount;
        this.hash = this.calculateHash();
    }


    private String calculateHash() {
        Instant now = Instant.now();
        return Cryptography.generateHash(this.fromAddress + this.toAddress + this.amount + now.toString());
    }
```

Kuvio 15. Transaktio-luokan attribuutit ja metodi tiivisteen laskemiseksi

```
public void signTransaction(KeyPair keypair) throws Exception {
    String publicKeyHex = Cryptography.toHexString(keypair.getPublic().getEncoded());

    if (!publicKeyHex.equals(this.fromAddress)) {
        throw new Exception("Cannot sign transaction for someone else's wallet.");
    }

    Signature ecdaSign = Signature.getInstance("SHA256withECDSA");
    ecdaSign.initSign(keypair.getPrivate());
    ecdaSign.update(this.hash.getBytes());
    String signature = Cryptography.toHexString(ecdaSign.sign());
    this.signature = signature;
}
```

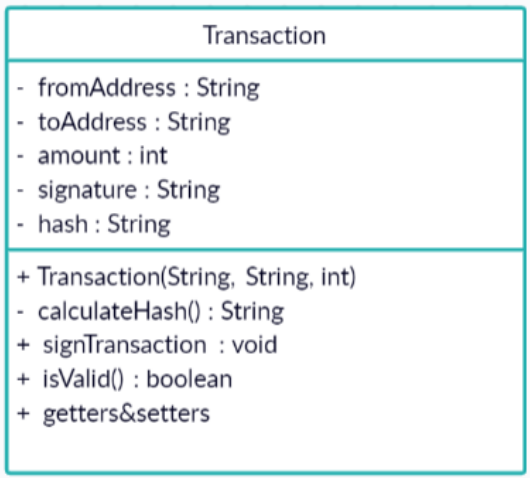
Kuvio 16. Luokan metodi transaktion allekirjoittamiseksi.

Transaktion oikeellisuus tulee myös voida varmentaa. Muutenhan allekirjoituksella ei olisi merkitystä, jos allekirjoitusta ei tarkistettaisi myöhemmin. Määrittelemme oliolle siis vielä metodin isValid(), joka tarkistaa transaktion oikeellisuuden (kuvio 17).

```
public boolean isValid() throws Exception {

    if (this.fromAddress == null) {
        return true;

    } else if (this.signature == null || this.signature.length() == 0) {
        throw new Exception("No signature");

    } else {

    Signature ecdsaVerify = Signature.getInstance("Sha256withECDSA");
    PublicKey publicKey = Cryptography.generateFromPublic
            (Cryptography.hexStringToByteArray(this.fromAddress));
    ecdsaVerify.initVerify(publicKey);
    ecdsaVerify.update(this.hash.getBytes());
    boolean result = ecdsaVerify.verify(Cryptography.hexStringToByteArray(this.signature));


    return result;

    }
}
```

Kuvio 17. Metodi transaktion tarkastamiseksi.

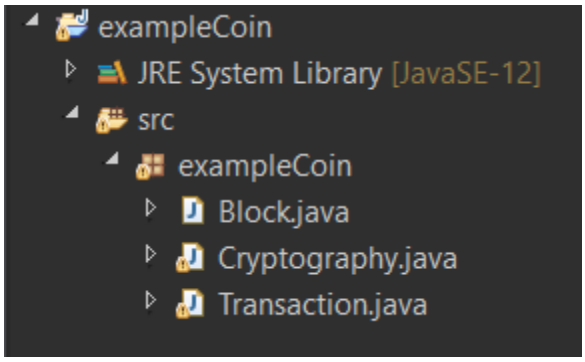
Määrittelyn jälkeen voidaan kirjoittaa luokalle loputkin ohjelmakoodista (liite 4) kuvion 18 luokkakaavion mukaisesti.



Kuvio 18. Transaktio-luokkakaavio.

4.2.4 Lohko

Kun transaktio-luokka on määritelty, tarvitsemme suunnitelman mukaisen paikan, johon transaktiot voidaan säilöä. Kuten aiemmin määritelty, transaktiot asetetaan lohkoon. Luodaan siis seuraavaksi projektiin lohko-luokka, ”Block.java” (kuvio 19).



Kuvio 19. Lohko-luokka ”Block.java” lisättynä projektiin.

Lohkon tarkoitus on säilöä käyttäjien väliset transaktiot, joten lohkossa on Transaktio-olioita hyväksyvä ArrayList yhtenä attribuuttina. Lohkot tallennetaan lohkoketjun periaatteen mukaisesti ”ketjuun”. Bitcoinin esimerkkiä seuraten lohkot liitetään toisiinsa niistä lasketun tiivisteen avulla. Lohkon sormenjälkenä toimiva tiiviste muodostetaan sivun 11, kuvion 4 mukaisesti laskemalla yhteen lohkon sisältö (kuvio 20) hash-funktiolla. Tällä tavoin toimien, kuten luvussa 2.3 on osoitettu, lohkon tiiviste muuttuu, jos mikään ketjussa aiemmin tallennettu tieto muuttuu. Näin tietoa ei siis voida muokata huomaamattomasti.

Yhteenlaskettavaan sisältöön lukeutuu:

- Kronologisesti aiemman ketjussa olevan lohkon tiiviste.
- Lohkolle asetettava aikaleima.
- Kaikki lohkon sisältämät Transaktio-oliot.
- Nonce-numero.

```
public class Block {

    private String timestamp;
    private ArrayList<Transaction> transactions;
    private String previousHash;
    private String hash;
    private int nonce;
```

Kuvio 20. Lohko-luokan sisältämät attribuutit.

Jotta lohkolle voidaan asettaa aikaleima, luodaan lohko-luokalle metodi aikaleiman luomista varten, jota kutsutaan olion luonnin yhteydessä (kuvio 21).

```
public Block(ArrayList<Transaction> transactions, String previousHash) {
    super();
    this.timestamp = timestamp();
    this.transactions = transactions;
    this.previousHash = previousHash;
    this.nonce = 0;

    String hash = calculateHash();
    if (hash != "") {
        this.hash = hash;
    }
}
```

Kuvio 21. Lohko-olion konstruktori.

Tiivisteen laskemiseksi lohkolle luodaan metodi, joka käyttää kryptografia-luokan generateHash() -metodia tiivisteen laskemiseksi (kuvio 22).

```
public String timestamp() {
    Instant now = Instant.now();
    return(now.toString());
}


public String calculateHash() {
    String hash = "";
    hash = Cryptography.generateHash(this.previousHash
            + this.timestamp
            + this.transactions
            + this.nonce);
    return hash;
}
```

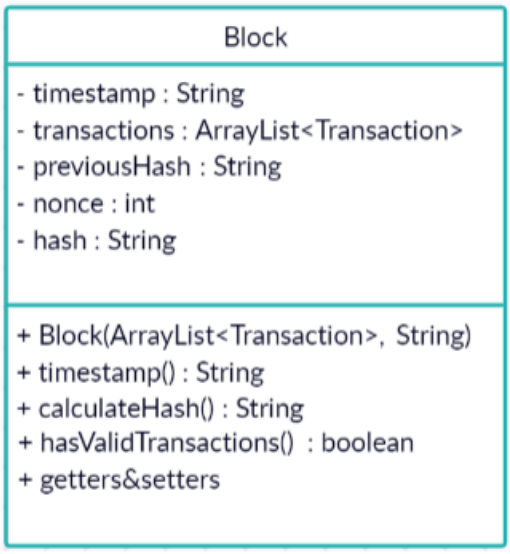
Kuvio 22. Lohko-luokan metodit aikaleiman ja tiivisteen muodostamiseksi.

Lohkolle on nyt määritelty sen sisältö sekä metodit tiivisteen laskemiselle ja sen aikaleimaamiselle. Lisäämme vielä luokalle metodin, jonka avulla voidaan varmistaa sen sisältämien transaktio-olioiden oikeellisuus käyttäen hyväksi transaktio-olion isValid() -metodia (kuvio 23).

```
public boolean hasValidTransactions( ) throws Exception {
    for (Transaction tx : this.transactions) {
        if (!tx.isValid()) {
            return false;
        }
    }

    return true;
}
```

Kuvio 23. Metodi lohkon sisältämien transaktioiden tarkistamiseksi.

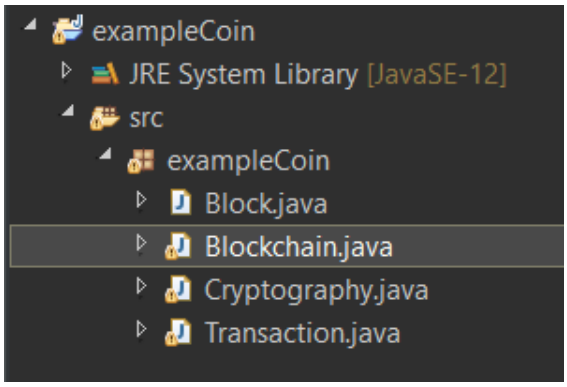
Määrittelyn jälkeen sen loput ohjelmakoodista voidaan toteuttaa Java-luokkaan kuvion 24 luokkakaavion mukaisesti (liite 5).



Kuvio 24. Lohko-luokkakaavio.

4.2.5 Lohkoketju

Järjestelmään on nyt määritelty transaktio-luokka ja lohko-luokka. Seuraava askel on tallentaa lohkot ketjun lailla kiinnitettynä toisiinsa tiivisteiden avulla. Järjestelmään tulee siis määritellä lohkoketju-luokka ”Blockchain.java” (kuvio 25), jonka tehtävänä on säilöä lohkot sisäänsä.



Kuvio 25. Projektin tiedostorakenne lohkoketju-luokan lisäämisen jälkeen.

Lohkoketjun tulee sulkea sisäänsä tallennettavat lohkot, joten määritellään lohkoketju-luokalle lohko-olioita sisältävä lista. Vaikka rekisteriin lisättävät lohkot voidaankin luoda linkitettynä toisiinsa tiivisteiden avulla, ei se tarkoita, että voisimme luottaa jokaisen lohkoketjua operoivan osallistujan tekevän sen samalla tavalla.
Tässä kohtaa tuleekin tarpeen määritellä järjestelmälle Bitcoinin tapaan proof-of-work protokolla, joka on esitelty luvussa 2.3.4.

Käytännössä tämä tarkoittaa, että lohkon lisäämiseksi järjestelmään sen tulee täyttää proof-of-work, joka saavutetaan laskemalla lohkon tiivistettä uudelleen ja uudelleen vaihtaen nonce-arvoa joka välissä, niin kauan, kunnes halutun tasoinen vaiva on nähty sen laskemiseen. Tämä näkyy ”vaikeustason” mukaisena määränä nollia lohkon sormenjälkenä toimivassa tiivisteessä. Lohkoketju-luokkaan lisätään vaikeustason kertova attribuutti, jota käytetään lohkon louhimisessa.

Vaikeustaso määräytyy verkostossa olevien osallistujien yhteisen laskennallisen tehon mukaan. Kun laskentateho nousee, lohkojen louhiminen nopeutuu. Bitcoin pyrkii pitämään lohkon laskemiseksi tarvittavan ajan vakiona, eli 10 minuuttia, jottei hyökkääjät voi yhtäkkisellä laskentatehon lisäämisellä manipuloida järjestelmää hyödykseen. (Siriwardena 2017.) Lohkoketju-oliolle asetetaan tämän simuloimiseksi attribuutti, joka kertoo tavoiteajan lohkon laskemiselle.

Proof-of-work -protokollan tekeminen aloitetaan lisäämällä lohko-luokkaan proof-of-work:in laskemisen, eli lohkon ”louhimisen” (engl. mining) mahdollistava metodi, joka saa arvokseen lohkoketjun senhetkisen vaikeustason (kuvio 26).

```
public void mineBlock(int difficulty) {

    String difficultyGoal = "";
    for (int i = 0; i < difficulty; i++) {
        difficultyGoal += "0";
    }

    System.out.println("Mining block with difficulty " + difficulty);

    while (!this.hash.substring(0, difficulty).equals(difficultyGoal)) {
        this.nonce++;
        this.hash = this.calculateHash();
    }
    System.out.println("Block " + this.hash + " succesfully mined.");
}
```

Kuvio 26. Lohko-olion louhimisen mahdollistava metodi.

Kun joku verkoston nodeista on laskenut lohkon proof-of-workin, lohko voidaan lisätä ketjuun. Jos muut osallistujat tarkastettuaan lohkon oikeellisuuden hyväksyvät sen, ne ryhtyvät työstämään seuraavaa lohkoa käyttäen tätä viimeisimmän lohkon tiivistettä uuden lohkon tiivisteen laskemisessa ja lohkon laskenut node saa palkinnon vaivasta. Näin ollen lohkoketjun tulee myös omata yhtenäinen tieto louhintapalkkiosta, joten louhintapalkkio määritellään yhdeksi lohkoketju-luokan attribuuteista.

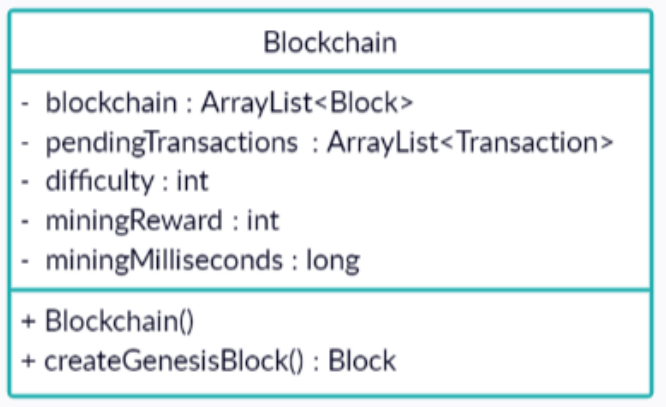
Järjestelmän tarkoituksena on tallettaa lohkoketjuun käyttäjiensä transaktioita luotettavalla tavalla. Järjestelmässä tulee siis olla myös jonkinlainen säiliö lohkoihin kerättäville transaktioille. Lohkoketju-luokalle määritellään siis ArrayList, joka ottaa ilmoitettavia transaktio-oliota vastaan, odottamaan lisäämistään lohkoihin.

Lohkoketju-luokka on näin valmis ottamaan vastaan luotettavia ja varmennettavia lohko-olioita. Ensimmäisen lohkon lisäämisessä tulee kuitenkin ongelma, sillä järjestelmässä kenelläkään ei ole vielä varallisuutta, joten transaktioita ei voida tehdä. Lohkoketjuun tulee kuitenkin saada talletettua ensimmäinen lohko, jonka tiivistettä seuraava lohko käyttää omansa laskennassa. Luokalle määritellään metodi, joka luo ensimmäisen ketjun lohkon, eli niin sanotun ”genesis”-lohkon (kuvio 27).

```
public Block createGenesisBlock() {
    this.pendingTransactions.add(new Transaction(null, null, this.miningReward));
    return new Block(
            new ArrayList<Transaction>()
            , Cryptography.toHexString(Cryptography.getSHA("Ei ole edellistä")));
}
```

Kuvio 27. Metodi genesis-lohkon luomiseksi lohkoketjuun.

Määrittelyn jälkeen luokkakaavio lohkoketjusta voisi näyttää kuvion 28 mukaiselta. Lohkoketju-olion määrittely on valmis, joten lohkoketju-luokan loput ohjelmakoodista voidaan toteuttaa luokkakaavion mukaisesti (liite 6).



Kuvio 28. Lohkoketju-luokkakaavio.

4.2.6 Node

Tietorakenteen mukaisesti eroteltavissa olevat osat on määritelty. Järjestelmä tarvitsee kuitenkin toimiakseen osallistujia, eli nodeja, jotka suorittavat järjestelmän toiminnallisuudet. Erotellaan node projektissa omaksi luokakseen, johon erittelemme järjestelmän vaatimat toiminnallisuudet metodeiksi (kuvio 29).

Kuvio 29. Node.java lisättynä projektiin.

Node on lähtökohtaisesti järjestelmän ajamisen toimittava osa. Node voi ajaa vain yhtä lohkoketjua kerrallaan, joten sen ajama lohkoketju-olio asetetaan noden attribuutiksi sen konstruktorissa luomishetkellä (kuvio 30).

```
public class Node {

    private Blockchain blockchain;

    public Node(Blockchain blockchain) {
        super();
        this.blockchain = blockchain;
    }
```

Kuvio 30. Node-luokan konstruktori.

Koska node käytännössä suorittaa koko järjestelmää, sen tulee myös omata kaikki tarvittavat toiminnallisuudet lohkoketjun turvallisuuden varmentamiseen, toiminnallisuuksien suorittamiseen sekä käyttäjän tarpeisiin. Esimerkkinä toimivan Bitcoinin verkosto toimii luvussa 2.3.4. esitetyssä järjestyksessä.

Node-olion työjärjestys tässä järjestelmässä on seuraavanlainen:

1. Node kerää lohkoketju-olion siirtoa odottavien transaktioiden listalla olevat transaktio-oliot ja tarkastaa jokaisen siltä varalta, että käyttäjillä on vaadittava varallisuus.
2. Node tarkastaa koko lohkoketjun ja käyttää viimeisimmän oikeellisen lohkon tiivistettä luodessaan uuden lohko-olion transaktioita varten.
3. Node hylkää vialliset transaktiot ja lisää hyväksytyt transaktiot luomaansa uuteen lohkoon.
4. Node laskee luomallensa lohkolle vaaditun vaikeustason mukaisen tiivisteen, täyttäen näin proof-of-workin.
5. Onnistuessaan tiivisteen laskemisessa node lisää lohkon lohkoketjuun. Verkoston nodet tarkistavat uuden lohkon. Nodet näyttävät hyväksyntänsä lisätylle lohkolle alkamalla työstää uutta lohkoa ketjuun sisällyttäen lisätyn lohkon tiivisteen oman, uuden lohkonsa tiivisteeseen.

Node tarvitsee siis metodit odottavien transaktioiden käsittelemiseen eli uuden lohkon louhimiseen (liite 7), lohkoketjun tarkastamiseen (liite 8), viimeisimmän oikeellisen lohkon löytämiseen ketjusta (liite 9), sekä myös metodi käyttäjien varallisuuden tarkistamiseen (liite 10). Toiminnallisuus lohkon louhimiseen käytetyn ajan mittaamiseksi ja säätämiseksi sijoitetaan louhimisen mahdollistavaan metodiin.

Koska noden metodia käyttäjän varallisuuden tarkistamiseksi tarvitaan Transaction-olion tasolla sen isValid() -metodissa, tehdään noden metodista staattinen ja otetaan transaktio-luokan metodissa vastaan koko lohkoketju parametrina (kuvio 31). Tämän lisäksi täytyy tehdä myös muutos lohko-luokan hasValidTransactions() -metodiin (kuvio 32).

```
public boolean isValid(Blockchain chain) throws Exception {

    if (this.fromAddress == null) {
        return true;

    } else if (this.signature == null || this.signature.length() == 0) {
        throw new Exception("No signature");

    } else if (Node.getBalanceOf(this.fromAddress, chain) < this.amount) {
        throw new Exception("Insufficient balance");
    } else {
```

Kuvio 31. Muutokset transaktio-luokan isValid() -metodissa.

```
public boolean hasValidTransactions(Blockchain chain) throws Exception {
    for (Transaction tx : this.transactions) {
        if (!tx.isValid(chain)) {
            return false;
        }
    }

    return true;
}
```

Kuvio 32. Muutokset lohko-luokan metodissa.

Myöskin uudet transaktiot julkistetaan lohkoketjuun nodejen kautta, joten node tarvitsee metodin transaktion lisäämiseen odottavien listalle (liite 11).

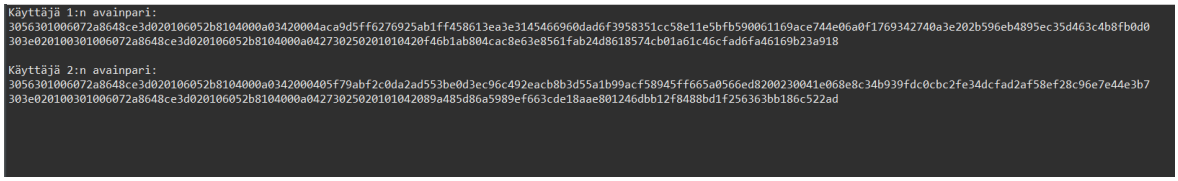
Järjestelmän testaamisen helpottamiseksi node-luokalle luodaan myös metodi, jolla koko lohkoketjun saa tulostettua lohko kerrallaan konsoliin tarkkailtavaksi (kuvio 33.)

```
public void printBlockchain() {

    int index = 1;

    for (Block b : this.blockchain.getBlockchain()) {

        System.out.println("Block #" + index);
        System.out.println(b);
        System.out.println();
        index++;

    }
}
```

Kuvio 33. Metodi lohkoketjun tulostamiseksi konsoliin.

4.2.7 Järjestelmän testaaminen

Seuraavaksi järjestelmää testataan sille asettettujen vaatimusten mukaisesti. Järjestelmää varten projektiin luodaan Main-luokka, jossa testikoodia ajetaan. Ensin luodaan kryptografia-luokan avulla kaksi uutta avainparia, joilla demonstroidaan käyttäjiä järjestelmässä (liite 12). Avainparit otetaan talteen testien ajaksi (kuvio 34).



Kuvio 34. Generoidut avainparit tulostettuna konsoliin.

Testaamista varten järjestelmään luotiin yksinkertainen konsolissa toimiva käyttöliittymä, jonka avulla järjestelmän toimintoja voi käyttää (liite 13). Käytön alussa asetetaan käytettävän lompakon osoitteeksi ensimmäinen julkinen avain, jonka jälkeen käyttöliittymä avautuu ja pyytää toimintoa.

Ensimmäiseksi testiksi kokeiltiin louhia muutama lohko, jotta käyttäjälle saadaan varallisuutta lähetettäväksi. Lohkojen louhimisen toiminnallisuus vaikeustason nostamiseksi ja laskemiseksi havaittiin toimivan. Muutaman lohkon louhittua käyttäjän varallisuus nousi louhittujen lohkojen verran (liite 14).

Seuraavaksi kokeiltiin lähettää 50 rahayksikköä toiseen alussa generoiduista osoitteista. Ensin yritettiin lähettää varat väärällä salaisella avaimella allekirjoitettuna, mutta järjestelmä osasi estää sen kuten kuuluukin (liite 15). Seuraavaksi 50 rahayksikköä lähetettiin oikean salausavaimen turvin ja se onnistui (liite 16). Myöskään rahaa ei kyennyt lähettämään enempää, kuin oli varallisuutta (liite 17). Voimme siis todeta, että tällä testillä järjestelmä täyttää osion 4.2.1. vaatimukset 1, 3, 4, ja 5.

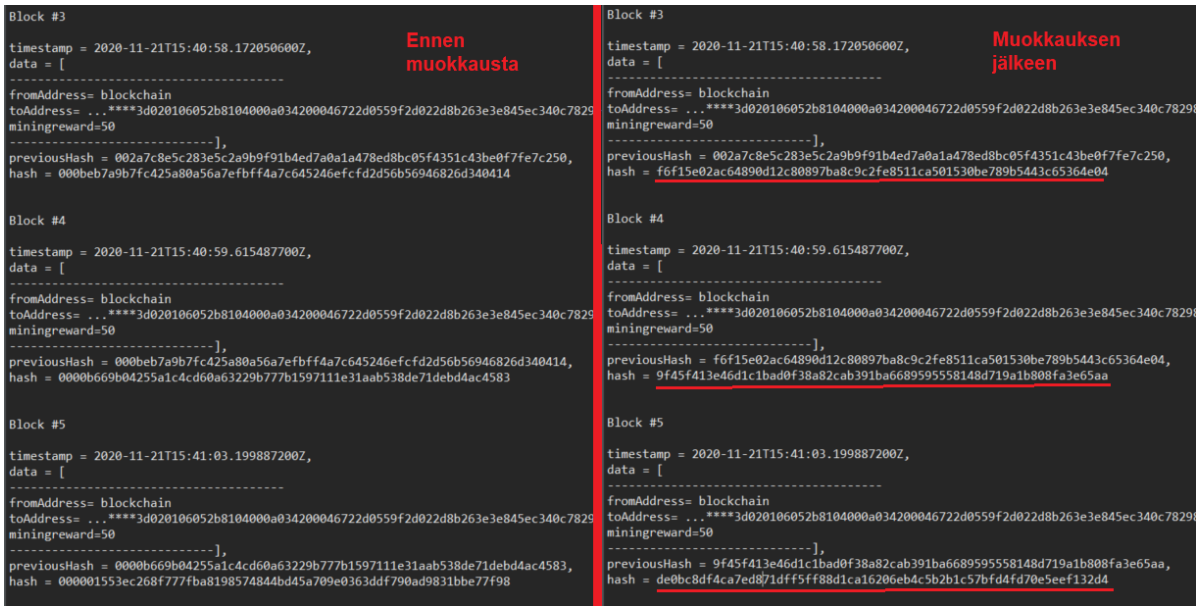
Tärkeintä järjestelmän testaamisessa on kuitenkin vaatimus numero 2, sillä järjestelmän kaikkein tärkein tarkoitus oli alun perin demonstroida lohkoketjua tietorakenteena. Tätä tiedon tallenustapaa kokeillaan louhimalla muutama lohko ja sitten yrittämällä manipuloida ketjussa olevaa lohkoa, jonka jälkeen verrataan viimeisen lohkon tiivistettä aiemman ajankohdan tiivisteeseen.

Louhittua muutaman lohkon, lohkoketju tulostettiin ja testattiin sen oikeellisuus (liite 18). Seuraavaksi kolmannen lohkon nonce-numeroa muutettiin ja verrattiin sen jälkeisten lohkojen tiivisteitä edellisen ajankohdan tiivisteisiin. Testatessa lohkoketjun eheyttä, tulokseksi tuli epätosi (liite 19). Huomasin, että lohkoketjusta puuttuu metodi, joka päivittää ketjun lohkojen tiivisteet, joten kyseinen metodi lisätään Node-luokkaan (kuvio 35).

```java
public void refreshChain(Blockchain chain) {
    String previousHash = "f52776a20666d540a951cb14ad3c5f25033c74e8a845fb1f508cfc5f5ab6e0ab";
    for (Block b : chain.getBlockchain()) {
        b.setPreviousHash(previousHash);
        b.setHash(b.calculateHash());
        previousHash = b.calculateHash();
    }
}
```

Kuvio 35. Node-luokan metodi ketjun päivittämiseksi.

Lohkoketjun manipuloinnin jälkeen järjestelmä huomasi lohkoketjun korruptoituneen, joten tiedon muuttaminen jälkeenpäin ei onnistu huomaamatta. Tulostamalla päivitetyn ketjun näemme, että kaikkien muutetun, ja sitä seuraavien lohkojen tiivisteet ovat muuttuneet (kuvio 36).



Kuvio 36. Tulosteet ennen ja jälkeen ketjun manipuloinnin.

Seuraavaksi kokeillaan järjestelmän ”hajauttamista” asettamalla sama lohkoketju useammalle nodelle. Tätä varten luotiin kolme erillistä nodea, jotka operoivat samaa lohkoketjua. Kaksi nodea laitettiin vuorotellen louhimaan kun välissä kolmas node ilmoitti allekirjoittamattoman transaktion, jonka jälkeen nodet jatkoivat louhimista (kuvio 37).

```
Blockchain ketju = new Blockchain();

Node node = new Node(ketju);
String rewardAddress1 = "3056301006072a8648ce3d020106052b8104000a034200046722d0559f2d022

Node node2 = new Node(ketju);
String rewardAddress2 = "3056301006072a8648ce3d020106052b8104000a03420004f16dfb7e1ad8214

Node node3 = new Node(ketju);

node.minePendingTransactions(rewardAddress1);
node.minePendingTransactions(rewardAddress1);
node2.minePendingTransactions(rewardAddress2);
node.minePendingTransactions(rewardAddress1);
Node.getBalanceOf(rewardAddress1, ketju);
Node.getBalanceOf(rewardAddress2, ketju);

node3.addTransaction(new Transaction(rewardAddress1, rewardAddress2, 100));

node2.minePendingTransactions(rewardAddress2);

node.printBlockchain();
```

Kuvio 37. Ohjelmakoodi, jolla simuloidaan järjestelmän ”hajauttaminen”

Nodet kuitenkin tunnistivat varmentamattoman transaktion (liite 20), eivätkä lisänneet sitä lohkoon (liite 21). Järjestelmä siis mahdollistaa noden kyvyn tunnistaa toisen noden epärehellisyys ja olla välittämättä siitä.

Viimeisenä testinä hajautuksen ja konsensusprotokollan toiminnan varmistamiseksi kokeilemme sitä, että yksi nodeista lisää suoraan lohkoketjuun väärennetyn lohkon, jossa on väärennetty transaktio, jonka jälkeen toisella nodella tarkistetaan ketju (kuvio 38).

```
System.out.println("Muutetaan ketjua:");

node3.getBlockchain().getBlockchain().add(
        new Block(
                new ArrayList<Transaction>(Arrays.asList(
                        new Transaction(rewardAddress1, rewardAddress2, 100)))
                , node3.findLastValidBlock().getHash()));

try {
    System.out.println(node.isChainValid());
} catch (Exception e) {
    System.out.println(e.getMessage());
}


System.out.println("\nTulostetaan päivitetty ketju:\n");
node.printUpdatedBlockchain();
```

Kuvio 38. Testikoodi lisättynä edellisen jatkoksi.

Liitteestä 22 huomaa, että ajamalla uudistetun koodin, konsolissa on ilmoitus, että node oli ketjua tarkastaessaan havainnut väärennetyn lohkon ketjusta (ilmoitus ”invalid transaction”) ja näin ollen sivuuttaisi sen alkaessaan työstämään seuraavaa. Konsoliin tulostetusta ketjusta huomaa myös, että myös seitsemännen lohkon tiivisteestä puuttuu proof-of-work, eli riittävä määrä nollia.

5 Pohdinta

Tälle opinnäytetyölle oli asetettu tutkimuskysymykset mikä on lohkoketju, miten lohkoketju toimii ja miten lohkoketjun voi luoda.

Ensimmäiseen tutkimuskysymykseen vastattiin luvuissa 2 ja 3, joissa käsiteltiin lohkoketjun keksimiseen johtanutta historiaa, lohkoketjun itsensä käsitettä sekä esimerkkejä lohkoketjujen käytöstä. Näissä opinnäytetyön tietoperustan määrittelevässä luvussa todettiin lohkoketjun olevan digitaalinen hajautettu tietokanta, jonka rakenne mahdollistaa sen sisältämän tiedon varmennettavuuden ja luotettavuuden, sekä käyttäjienvälisen luottamuksen konsensusmenekanismin avulla.

Toista tutkimuskysymystä käsiteltiin luvussa 2, eritoten osassa 2.3, jossa käytettiin Bitcoinin toimintamekaniikkaa esimerkkinä lohkoketjun toiminnasta. Bitcoin toimi hyvänä esimerkkinä lohkoketjun toiminnoista sen ollessa ensimmäinen järjestelmä, joka käytti lohkoketjua, antaen sille alkuperäisen määritelmän. Lohkoketjun toiminta käsiteltiin siten, että se vastaa tutkimuskysymykseen ja antaa perustan lähteä toteuttamaan tutkimusta käytännön tasolla, opinnäytetyön johdannossa määritellyn menetelmän mukaisesti.

Kolmatta tutkimuskysymystä ”miten lohkoketjun voi luoda” käsiteltiin luvussa 4, jossa toteutettiin lohkoketjun toimintaa demonstroiva järjestelmä Java -ohjelmointikielellä, Bitcoinin järjestelmää esimerkkinä käyttäen. Luvussa käytettiin hyväksi kerättyä tietoperustaa lohkoketjusta ja sen toiminnasta siten, että luotava järjestelmä oli mahdollista määritellä ja toteuttaa. Järjestelmän toteutus ja dokumentointi on tapahtunut sellaisessa järjestyksessä, että luku 4 vastaa tutkimuskysymykseen esimerkin lailla.

Opinnäytetyön luvussa 4 määriteltyä ja luotua esimerkkiä lohkoketjujärjestelmästä verrataan kerrytetyn tietoperustan analysoimisen perusteella kerättyyn listaan järjestelmän vaatimuksista, jotka lohkoketjujärjestelmä tarvitsee ollakseen toimintaperiaatteiden mukainen, tutkimusmenetelmä on siis kvalitatiivinen. Järjestelmän testaaminen luvussa 4.2.7 osoitti, että lohkoketjudemonstraatio on vaatimusten mukainen.

Henkilökohtaisena tavoitteenani opinnäytetyössä oli oppia lisää sekä syventää tietämystäni lohkoketjuista, ymmärtää sen toimintaperiaatteet ja soveltaa niitä käytännön tasolle. Lisäksi tavoitteenani oli myös tarjota lukijalle tavanomaisesta poikkeava, käytännönläheinen näkökulma lohkoketjun periaatteiden ja toiminnan sisäistämiseksi.

Opinnäytetyössä rakennettu järjestelmä vaati laajaa teoreettista analysointia lohkoketjuista tietorakenteena ja yleisesti, kuin myös järjestelmänä teknisestikin. Lohkoketjun konkreettisen luomisen vaatima tietoperustan kartuttaminen antoi pohjan lohkoketjun periaatteiden ymmärtämiselle, mutta varsinkin järjestelmää rakentaessani ymmärrykseni lohkoketjun toiminnasta todellisesti syventyi. Henkilökohtainen tavoitteeni oppimisesta ja tietämykseni syventämisestä aiheeseen liittyen toteutui.

Järjestelmän konkreettinen luominen antoi myös täysin erilaisen perspektiivin teoriaan suhteutettuna. Järjestelmän toiminnallisuuksien kartoittaminen herätti lohkoketjujen teknisestä toiminnasta sellaisia kysymyksiä, joita en olisi osannut aiemmin ajatella, kuten kysymyksen siitä, miten lohkoketjujärjestelmä saadaan toimimaan halutulla tavalla. Selvittäessäni kysymyksiin ratkaisua syventymällä Bitcoinin toimintaan, loi se sellaisen asetelman, jossa tietoa toiminnasta haki syventävästi ja konkreettisiin käyttötarkoituksiin. Kun haettua tietoa käytti järjestelmän rakentamiseen, oli helppoa myös soveltaa haettu tieto konkreettiseksi ohjelmakoodiksi.

Asetettujen vaatimusten suhteen, hajauttaminen tässä järjestelmässä ei kuitenkaan toteudu lohkoketjun periaatteiden mukaisesti. Koska luotavan järjestelmän tarkoitus oli toimia lohkoketjun toimintaa demonstroivana esimerkkinä, ei tämä järjestelmä toimi oikeassa hajautetussa verkossa, vaan useamman noden verkostoa ainoastaan simuloidaan. Transaktioita ei siis lähetetä verkon kautta muille nodeille, vaan tässä järjestelmässä sen sijaan transaktiot ilmoitetaan suoraan noden kautta lohkoketjun odottavien transaktioiden listalle. Verkoston simuloiminen vaikutti myös järjestelmän lohkojen louhimiseen. Esimerkiksi Bitcoinissa nodet omaa lohkoa louhiessaan jatkuvasti kuuntelevat, jos vaikka jokin muu verkoston node ehtii ratkaisemaan omansa ensin, ja alkaa tarkastettuaan lohkon työstämään uutta lohkoa sen perään. Koska verkostoa simuloiva järjestelmämme toimii yhdellä suorittimella, todellisuudessa vain yksi node voi louhia lohkoa kerrallaan, eikä node näin ollen voi kuunnella samanaikaisesti muiden nodejen louhinnan valmistumista.

Edellä mainitut seikat järjestelmästä oli kuitenkin tiedossa järjestelmää luodessa ja se selviää myös kerätystä tietoperustasta. Koska asia tuli ottaa järjestelmää luodessa huomioon, tuli myös ymmärtää oikea toimintamekaniikka, vaikka tässä siitä hieman poikettiinkin käytännöllisistä syistä. Näin ollen näen kuitenkin henkilökohtaisten tavoitteiden toimintaperiaatteiden ymmärtämisestä sekä soveltamisesta täytetyksi.

Opinnäytetyön tarkoituksena oli myös helpottaa lohkoketjun toimintaperiaatteiden sisäistämistä käytännön tasolle dokumentoimalla lohkoketjun pääasiallisten toimintaperiaatteiden mukaisen konkreettisen esimerkin luontiprosessi. Tämän tavoitteen mittaaminen on

hankalaa, sillä asia on hyvin subjektiivinen. Luulisin kuitenkin, että opinnäytetyössä on kerätty tarpeellinen tietoperusta ja dokumentointi sen käyttämisestä sillä tasolla, että uskon opinnäytetyön helpottavan tätä tarkoitukseksi asetettua sisäistämisprosessia. Kysymys jää, että kuinka paljon se sisäistämistä helpottaa.

Vaikka opinnäytetyön tuotoksena syntyi järjestelmä, joka mukailee Bitcoinin tapaa käyttää lohkoketjua nimenomaan virtuaalivaluutan toiminnan mahdollistamiselle, kun lohkoketjun on kerran luonut käytännössä, on myös lisäksi paljon parempi ymmärrys siitä, mihin muihin tarkoituksiin sitä voi käyttää ja miten sen voisi toteuttaa myös mahdollisesti muilla ohjelmointikielillä.

Luomisprosessi oli muutenkin mielenkiintoinen kokemus oppimisesta, sillä järjestelmää luodessa abstraktia tietoa toimintaperiaatteista piti soveltaa käytäntöön, ja etsittävällä tiedolla oli valmiiksi käyttötarkoitus. Uskoisin sen olevan myös monille muille itseni kaltaisille tekemisen kautta oppiville ihmisille hyödyllinen tapa oppia uutta ja syventää valmista tietoperustaa mistä tahansa muustakin aiheesta.

Opinnäytetyön teko oli kaiken kaikkiaan erittäin tyydyttävää itselleni, sillä koin tekeväni jotain hieman tavallisista opinnäytetöistä poikkeavaa, joka sattui kaiken lisäksi olemaan itselleni mielenkiintoistakin. Haasteita kuitenkin oli muutamia, joiden luulisin johtuvan muun muassa subjektiivisesta tavasta toteuttaa opinnäytetyö luomalla ohjelmakoodia, sekä koko aiheen tuoreudesta informaatioteknologian alalla. Ensimmäisenä haasteena koin kerätyn ja tietyllä tavalla käsittämäni abstraktin tiedon mallintamisen siihen muotoon, että opinnäytetyön lukijakin ymmärtää. Lisää haastetta toi tietolähteet, jotka ovat tästä aiheesta lähtökohtaisesti suurimmilta osin vielä englanninkielisiä, eikä suomen kielen määritelmiä käsitteille ole vielä välttämättä vakiinnutettu. Monessa tilanteessa jouduin miettimään kirjoitusasua, jottei lohkoketjuihin liittyvä suomenkielinen käsite aiheuta väärinymmärryksiä lukijalle.

Koska lohkoketjun toimintaperiaatteiden sisäistämisen tueksi etsin myös tietoperustaan esimerkkejä tavoista soveltaa lohkoketjuja muunkinlaisiin järjestelmiin, jatkotutkimusaiheena voisi olla kartoittaa tapoja hyödyntää lohkoketjuja ja vertailla niiden toimintoja keskenään, lopulta erotellen yhteiset toiminnallisuudet, joita voidaan pitää loppujen lopuksi lohkoketjun toiminnallisina yleisperiaatteina.

Lähteet

Asynclabs blog. 2018. Proof of Work – What it Is and How Does it Work? Luettavissa: https://www.asynclabs.co/blog/blockchain-development/proof-of-work-what-it-is-and-how-does-it-work/. Luettu: 11.11.2020.

Back, A. 2002. Hashcash – A Denial of service Counter-Measure Luettavissa: http://www.hashcash.org/papers/hashcash.pdf. Luettu: 11.11.2020.

Bayer, D., Haber, S. & Stornetta, W.S. 1992. Improving the Efficiency and Reliability of Digital Time-Stamping. Luettavissa: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.71.4891&rep=rep1&type=pdf. Luettu: 11.11.2020.

Belanger, L. 2018. 15 Crazy and Surprising Ways People Are Using Blockchain. Luettavissa: https://www.entrepreneur.com/slideshow/308004. Luettu: 11.11.2020.

Bitcoin Developer 2020. Developer guides – Transactions. Luettavissa: https://developer.bitcoin.org/devguide/transactions.html. Luettu: 11.11.2020.

Bonnadonna, E. 2013. Bitcoin and the Double-Spending Problem. Luettavissa: https://blogs.cornell.edu/info4220/2013/03/29/bitcoin-and-the-double-spending-problem/. Luettu: 11.11.2020.

Buterin, V. 2013. Ethereum Whitepaper. Luettavissa: https://ethereum.org/en/whitepaper/. Luettu: 11.11.2020.

Campbell, D. 2015. The Byzantine generals' problem. Luettavissa: https://www.dugcampbell.com/byzantine-generals-problem/. Luettu: 11.11.2020.

Chainlink. 2019. Driving Demand for Enterprise Smart Contracts Using the Trusted Computation Framework and Attested Oracles via Chainlink. Luettavissa: https://blog.chain.link/driving-demand-for-enterprise-smart-contracts-using-the-trusted-computation-framework-and-attested-oracles-via-chainlink/. Luettu: 11.11.2020.

Crosby, M., Nachiappan, Pattanayak, P., Verma, S. & Kalyanaraman, V. 2016. Blockchain technology: Beyond Bitcoin. Luettavissa: https://j2-capital.com/wp-content/uploads/2017/11/AIR-2016-Blockchain.pdf. Luettu: 11.11.2020.

Ellis, S., Juels, A. & Nazarov, S. 2017. ChainLink – A Decentralized Oracle Network. Luettavissa: https://link.smartcontract.com/whitepaper. Luettu: 11.11.2020.

Feuer, A. 2013. The bitcoin ideology. Luettavissa: https://web.archive.org/web/20180701222302/https://www.nytimes.com/2013/12/15/sunday-review/the-bitcoin-ideology.html. Luettu: 11.11.2020.

Gemini. 2020. What is Chainlink in 5 Minutes. Luettavissa: https://gemini.com/learn/what-is-chainlink-and-how-does-it-work. Luettu: 11.11.2020.

Gupta, V. 2017. A brief history of blockchain. Luettavissa: https://hbr.org/2017/02/a-brief-history-of-blockchain. Luettu: 11.11.2020.

Hackius, M & Petersen, N. 2017. Blockchain in Logistics and Supply Chain: Trick or Treat? Luettavissa: https://www.econstor.eu/bitstream/10419/209299/1/hicl-2017-23-003.pdf. Luettu: 11.11.2020.

Laura, M. 2020. What Is a Smart Contract and How Does it Work? Luettavissa: https://www.bitdegree.org/crypto/tutorials/what-is-a-smart-contract. Luettu: 11.11.2020.

Laurence, T. 2017. Blockchain for dummies. John Wiley & Sons, Inc. New Jersey.

Metry, M. 2017. Blockchain technlogy most significant invention sinve internet and electricity. Luettavissa: https://medium.com/@markymetry/blockchain-technology-is-the-most-significant-invention-since-the-internet-and-electricity-f2d44a631ef6. Luettu: 11.11.2020.

Mettler, M. 2016. Blockchain technology in healthcare: The revolution starts here. Ladattavissa: https://ieeexplore.ieee.org/document/7749510/. Luettu: 11.11.2020.

Mishra, D. 2019. ECDSA Digital Signature Verification in Java. Luettavissa: https://metamug.com/article/security/sign-verify-digital-signature-ecdsa-java.html. Luettu: 12.11.2020.

Mycryptopedia. 2018. What is sha-256 and how it’s related to bitcoin. Luettavissa: https://www.mycryptopedia.com/sha-256-related-bitcoin/. Luettu: 11.11.2020.

Nakamoto, S. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. Luettavissa: https://www.bitcoin.com/bitcoin.pdf. Luettu: 11.11.2020.

Nokia Museum. 2020. Category archives: 2000. Luettavissa: https://nokiamuseum.info/category/launching-year/2000/. Luettu: 11.11.2020.

PatentlyGerman. 2018. Blockchain patent activity increases – but granted patents are still rare in this field. Luettavissa: https://patentlygerman.com/2018/02/14/blockchain-patent-activity-increases-but-granted-patents-are-still-rare-in-this-field/. Luettu: 11.11.2020.

Raval, S. 2016. Decentralized Applications O'Reilly Media, Inc. Englanti.

Sherman, A., Javani, F., Zhang, H. & Golaszewski, E. 2018. On the Origins and Variations of Blockchain Technologies. Luettavissa: https://arxiv.org/ftp/arxiv/papers/1810/1810.06130.pdf. Luettu: 11.11.2020.

Siriwardena, P. 2017. The Mystery Behind Block Time. Luettavissa: https://medium.facilelogin.com/the-mystery-behind-block-time-63351e35603a. Luettu: 12.11.2020.

Viitala, J. 2016. Mikä on lohkoketju? Luettavissa: https://juhaviitala.com/2016/12/20/mika-on-lohkoketju/. Luettu: 11.11.2020.

Wallace, B. 2011. The rise and fall of bitcoin. Luettavissa: https://web.archive.org/web/20131031043919/http://www.wired.com/magazine/2011/11/mf_bitcoin. Luettu: 11.11.2020.

Liitteet

Liite 1. Kryptografia-luokan metodit avainten generoimiselle

```java
package demoCoin;

import java.nio.charset.StandardCharsets;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.spec.ECGenParameterSpec;
import java.security.spec.EncodedKeySpec;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;


public class Cryptography {


    public static KeyPair generateKeypair() {

        try {
            ECGenParameterSpec ecSpec = new ECGenParameterSpec("secp256k1");
            KeyPairGenerator g = KeyPairGenerator.getInstance("EC");
            g.initialize(ecSpec, new SecureRandom());
            KeyPair keypair = g.generateKeyPair();
            return keypair;
        } catch (Exception e) {
            System.out.println(e);
        }

        return null;

    }

    public static PrivateKey generateFromPrivate(byte[] privateKey) {

        try {

        KeyFactory f = KeyFactory.getInstance("EC");
        EncodedKeySpec spec = new PKCS8EncodedKeySpec(privateKey);
        PrivateKey key = f.generatePrivate(spec);

        return key;

        } catch (Exception e) {
            System.out.println(e);
        }

        return null;

    }

    public static PublicKey generateFromPublic(byte[] publicKey) {

        try {

        KeyFactory f = KeyFactory.getInstance("EC");
        EncodedKeySpec spec = new X509EncodedKeySpec(publicKey);
        PublicKey key = f.generatePublic(spec);

        return key;

        } catch (Exception e) {
            System.out.println(e);
        }

        return null;

    }
```

Liite 2. Kryptografia-luokan metodit heksadesimaalisten merkkijonojen käsittelyyn.

```
public static String toHexString(byte[] hash)
{

    StringBuilder hexString = new StringBuilder();

    for (byte b : hash) {
        hexString.append(String.format("%02x", b));
    }



    return hexString.toString();
}



public static byte[] hexStringToByteArray(String s) {
    int len = s.length();
    byte[] data = new byte[len / 2];
    for (int i = 0; i < len; i += 2) {
        data[i / 2] = (byte) ((Character.digit(s.charAt(i), 16) << 4)
                             + Character.digit(s.charAt(i+1), 16));
    }
    return data;
}
```

Liite 3. Kryptografia-luokan metodit sha-256 tiivisteen laskemiselle

```

public static String generateHash(String input) {
    return toHexString(getSHA(input));
}


public static byte[] getSHA(String input)
{
    MessageDigest md;
    try {
        md = MessageDigest.getInstance("SHA-256");

        return md.digest(input.getBytes(StandardCharsets.UTF_8));

    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }

    return null;
}
```

Liite 4. Loput transaktio -luokan koodista.

```
public String getFromAddress() {
    if (this.fromAddress != null) {
        return fromAddress;
    }
    return null;
}

public void setFromAddress(String fromAddress) {
    this.fromAddress = fromAddress;
}

public String getToAddress() {
    if (this.toAddress != null) {
        return toAddress;
    }
    return null;
}

public void setToAddress(String toAddress) {
    this.toAddress = toAddress;
}

public int getAmount() {
    return amount;
}

public void setAmount(int amount) {
    this.amount = amount;
}


@Override
public String toString() {
    if (this.fromAddress != null && this.toAddress != null) {
        return "\n----------------------------------------"
                + "\nfromAddress= ...****" + fromAddress.substring(20)
                + "\ntoAddress= ...****" + toAddress.substring(20)
                + "\namount=" + amount
                + "\nsignature=" + signature
                + "\nhash=" + hash + "\n----------------------------";
    } else if (this.toAddress != null) {
        return "\n----------------------------------------"
                + "\nfromAddress= blockchain"
                + "\ntoAddress= ...****" + toAddress.substring(20)
                + "\nminingreward=" + amount
                + "\n----------------------------";
    }
    return "\n----------------------------------------"
    + "\ngenesis"
    + "\n----------------------------";
}
```

Liite 5. Loput lohko -luokan koodista.

```
public String getTimestamp() {
    return timestamp;
}



public void setTimestamp(String timestamp) {
    this.timestamp = timestamp;
}


public ArrayList<Transaction> getTransactions() {
    return transactions;
}


public void setTransactions(ArrayList<Transaction> transactions) {
    this.transactions = transactions;
}


public int getNonce() {
    return nonce;
}


public void setNonce(int nonce) {
    this.nonce = nonce;
}


public String getPreviousHash() {
    return previousHash;
}



public void setPreviousHash(String previousHash) {
    this.previousHash = previousHash;
}



public String getHash() {
    return hash;
}



public void setHash(String hash) {
    this.hash = hash;
}



@Override
public String toString() {
    return "\ntimestamp = " + timestamp
            + ", \ndata = " + transactions
            + ", \npreviousHash = " + previousHash
            + ", \nhash = " + hash + "\n";
}
```

Liite 6. Loput lohkoketju -luokan koodista.

```
public class Blockchain {

    private ArrayList<Block> blockchain;
    private ArrayList<Transaction> pendingTransactions;
    private int difficulty;
    private int miningReward;
    private long miningMilliseconds;

    public Blockchain() {
        super();
        this.difficulty = 2;
        this.miningReward = 50;
        this.blockchain = new ArrayList<>();
        this.pendingTransactions = new ArrayList<>();
        this.blockchain.add(this.createGenesisBlock());
        this.miningMilliseconds = 60000;
    }

    public Block createGenesisBlock() {
        this.pendingTransactions.add(new Transaction(null, null, this.miningReward));
        return new Block(
                new ArrayList<Transaction>()
                , Cryptography.toHexString(Cryptography.getSHA("Ei ole edellistä")));
    }
```

Liite 7. Noden -luokan metodi odottavien transaktioiden käsittelyyn

```
public void minePendingTransactions(String rewardAddress) throws Exception {

    ArrayList<Transaction> transactions = new ArrayList<>();

    for (Transaction t : this.blockchain.getPendingTransactions()) {    // Tarkastaa transaktiot ja
                                                                        // käyttäjien varallisuuden
        if (t.getFromAddress() != null) {
            if (Node.getBalanceOf(t.getFromAddress(), this.blockchain) > t.getAmount()) {
                transactions.add(t);
            }
        }
    }

    Block block = new Block(transactions, this.findLastValidBlock().getHash()); // Etsii viimeisimmän
                                                                                 // validin lohkon
    long start = System.currentTimeMillis();    // Ajanlaskutoiminnallisuus

    block.mineBlock(this.blockchain.getDifficulty());   // Aloittaa lohkon louhinnan

    long end = System.currentTimeMillis();
    long elapsedTime = end - start;

    this.blockchain.getBlockchain().add(block);     // Louhinnan valmistuttua lisää lohkon ketjuun
    System.out.println("Block added to chain.");

    this.blockchain.setPendingTransactions(new ArrayList<Transaction>()); // Tyhjentää listan

    this.blockchain.getPendingTransactions().add(   // Lisää uudelle listalle louhintapalkkion
            (new Transaction(null, rewardAddress, this.blockchain.getMiningReward())));

    if (elapsedTime < (this.blockchain.getMiningMilliseconds() * 0.2)) {    // Toiminnallisuus louhinnan
        this.blockchain.increaseDifficulty();                               // ajan säätämisen
        System.out.println("Took " + (elapsedTime / 1000)                   // simuloimiseksi
                + " seconds. \nDifficulty increased to " + this.blockchain.getDifficulty());
    } else if (elapsedTime > (this.blockchain.getMiningMilliseconds() * 1.2)) {
        this.blockchain.decreaseDifficulty();
        System.out.println("Took " + (elapsedTime / 1000)
                + " seconds. \nDifficulty decreased to " + this.blockchain.getDifficulty());
    } else {
        System.out.println("Took " + (elapsedTime / 1000) + " seconds.");
    }

}
```

Liite 8. Node -luokan metodi lohkoketjun eheyden tarkastamiseen.

```
public boolean isChainValid() throws Exception {

    for (int i = 1; i < this.blockchain.getBlockchain().size(); i++) {

        Block current = this.blockchain.getBlockchain().get(i);
        Block previous = this.blockchain.getBlockchain().get(i - 1);


        if (!current.getHash().equals(current.calculateHash())) {
            System.out.println("Invalid block hash : " + i);
            System.out.println(current.getHash());
            System.out.println(current.calculateHash());
            System.out.println();
            return false;
        }

        if (!current.getPreviousHash().equals(previous.getHash())) {
            return false;
        }

        if (!current.hasValidTransactions(this.blockchain)) {
            return false;
        }
    }
    return true;
}
```

Liite 9. Node -luokan metodi viimeisimmän eheän lohkon löytämiseksi ketjusta

```
public Block findLastValidBlock() throws Exception {

    for (int i = 1; i < this.blockchain.getBlockchain().size(); i++) {

        Block current = this.blockchain.getBlockchain().get(i);
        Block previous = this.blockchain.getBlockchain().get(i - 1);

        if (!current.getHash().equals(current.calculateHash())) {           // Tarkistetaan onko lohkon
            System.out.println("Invalid block hash at chain index: " + i);  // tiiviste muodostettu oikei
            System.out.println(current.getHash());
            System.out.println(current.calculateHash());
            System.out.println();
            return previous;
        }

        if (!current.getPreviousHash().equals(previous.getHash())) {    // Tarkistetaan täsmääkö lohkon
            return previous;                                            // lohkon "edellinen tiiviste"
        }                                                               // edellisen lohkon tiivisteeseen

        if (!current.hasValidTransactions(this.blockchain)) {       // Tarkistetaan lohkon transaktiot
            return previous;
        }

    }

    return this.blockchain.getBlockchain().get(this.blockchain.getBlockchain().size() - 1);

}
```

Liite 10. Node -luokan metodi käyttäjän varallisuuden tarkastamiseksi.

```
public static int getBalanceOf(String address, Blockchain chain) {

    int balance = 0;

    for (Block b : chain.getBlockchain()) {     // Lasketaan saldo lohkoketjun transaktioista
        for (Transaction t : b.getTransactions()) {
            if (t.getFromAddress() != null) {
                if (t.getFromAddress().equals(address)) {
                    balance -= t.getAmount();
                }

                if (t.getToAddress().equals(address)) {
                    balance += t.getAmount();
                }
            } else if (t.getToAddress() != null){
                if (t.getToAddress().equals(address)) {
                    balance += t.getAmount();
                }
            }
        }
    }

    for (Transaction t : chain.getPendingTransactions()) {  // Lasketaan saldo myös
        if (t.getFromAddress() != null) {                   // odottavista transaktioista
            if (t.getFromAddress().equals(address)) {
                balance -= t.getAmount();
            }

            if (t.getToAddress().equals(address)) {
                balance += t.getAmount();
            }
        } else if (t.getToAddress() != null){
            if (t.getToAddress().equals(address)) {
                balance += t.getAmount();
            }
        }
    }

    return balance;
}
```

Liite 11. Node -luokan metodi transaktion lisäämiseksi järjestelmään.

```
public void addTransaction(Transaction tx) throws Exception {

    try {
        if (tx.getFromAddress() == null                 // Varmistetaan, että lisättävään transaktioon
            || tx.getFromAddress().length() == 0        // on merkitty lähettäjä ja vastaanottaja
            || tx.getToAddress() == null
            || tx.getToAddress().length() == 0 ) {

            throw new Exception("Must include from and to address");

        } else if (!tx.isValid(this.getBlockchain())) {     // Käytetään transaktio-luokan
            throw new Exception("Invalid transaction");     // isValid() -metodia
                                                            // allekirjoituksen tarkistamiseen.

        } else if (Node.getBalanceOf(tx.getFromAddress(), this.blockchain) < tx.getAmount()) {
                                                            // Varmistetaan, että transaktion
            throw new Exception("Insufficient wealth");     // lähettäjällä on tarvittava varallisuus

        } else {
            this.blockchain.getPendingTransactions().add(tx);   // Jos kaikki on ok,
                                                                // lisätään transaktio
                                                                // lohkoketjuun odottavien listalle.
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

Liite 12. Avainparin luominen testejä varten.

```
KeyPair k1 = Cryptography.generateKeypair();
KeyPair k2 = Cryptography.generateKeypair();

System.out.println("Käyttäjä 1:n avainpari:");
System.out.println(Cryptography.toHexString(k1.getPublic().getEncoded()));
System.out.println(Cryptography.toHexString(k1.getPrivate().getEncoded()));

System.out.println();

System.out.println("Käyttäjä 2:n avainpari:");
System.out.println(Cryptography.toHexString(k2.getPublic().getEncoded()));
System.out.println(Cryptography.toHexString(k2.getPrivate().getEncoded()));
```

Liite 13. Yksinkertaisen konsolikäyttöliittymän koodi

```
public class Main {

    public static void main(String[] args) throws Exception {

        Blockchain ketju = new Blockchain();

        Node node = new Node(ketju);


        Scanner s = new Scanner(System.in);
        boolean kaynnissa = true;


        String walletAddress = "";

        while (kaynnissa) {

            if (walletAddress == "") {
                System.out.println("Enter wallet address: ");
                walletAddress = s.next();
            }

            System.out.println();
            System.out.println("Functions:\n"
                    + "1: Add a transaction\n"
                    + "2: Mine pending transactions\n"
                    + "3: Check account balance\n"
                    + "4: Check if chain is valid\n"
                    + "5: Test altering chain\n"
                    + "6: Print blockchain\n"
                    + "\n"
                    + "9: Logout current wallet\n"
                    + "0: Quit\n");
            System.out.print("\nSelect function: ");


            int input = s.nextInt();

            System.out.println();

            switch (input) {

              case 0:
                  kaynnissa = false;
                  break;
```

```
            case 1:
                try {

                    System.out.println("\nEnter target address: ");
                    String to = s.next();
                    int balance = Node.getBalanceOf(walletAddress, node.getBlockchain());
                    System.out.println("\nYour balance: " + balance);
                    System.out.println("Enter the amount: ");
                    int amount = s.nextInt();
                    System.out.println("Enter your private key: ");
                    String pvtKey = s.next();
                    System.out.println("\nAre you sure? y / n");

                    if (s.next().equals("y")) {
                        KeyPair kp = Cryptography.generatePairFromHex(walletAddress, pvtKey);
                        Transaction tx = new Transaction(walletAddress, to, amount);
                        tx.signTransaction(kp);
                        node.addTransaction(tx);
                        break;
                    }

                } catch (Exception e) {
                    System.out.println(e.getMessage());
                    break;
                }
                break;


            case 2:
                String rewardTo = walletAddress;
                node.minePendingTransactions(rewardTo);
                break;

            case 3:
                System.out.println("\nYour balance: " + Node.getBalanceOf(walletAddress,
                        node.getBlockchain()));
                break;

            case 4:
                System.out.println(node.isChainValid());
                break;

            case 5:
                System.out.println("Enter block index: ");
                int index = s.nextInt();

            case 6:
                node.printBlockchain();

            case 9:
                walletAddress = "";
                break;


            default:
                System.out.println("unknown command");
                continue;

        }
    }
```

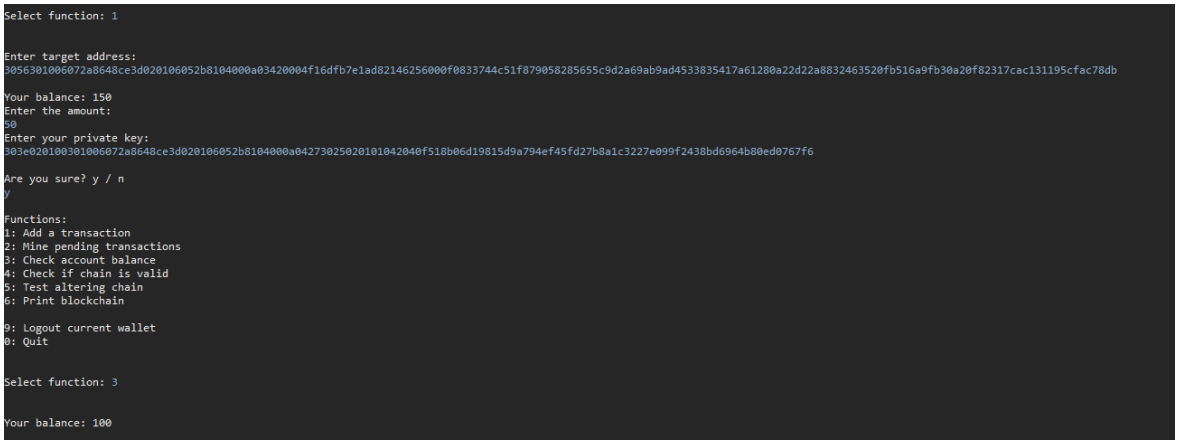
Liite 14. Testilouhinnan tulos, testikäyttäjän varallisuus

```
Mining block with difficulty 4
Block 0000b56455b86803a832edeebf3e843df89e087804268d51368f30cb7ff75350 succesfully mined.
Block added to chain.
Took 0 seconds.
Difficulty increased to 5

Functions:
1: Add a transaction
2: Mine pending transactions
3: Check account balance
4: Check if chain is valid
5: Test altering chain
6: Print blockchain

9: Logout current wallet
0: Quit


Select function: 3


Your balance: 150

Functions:
1: Add a transaction
2: Mine pending transactions
3: Check account balance
4: Check if chain is valid
5: Test altering chain
6: Print blockchain

9: Logout current wallet
0: Quit

Select function:
```

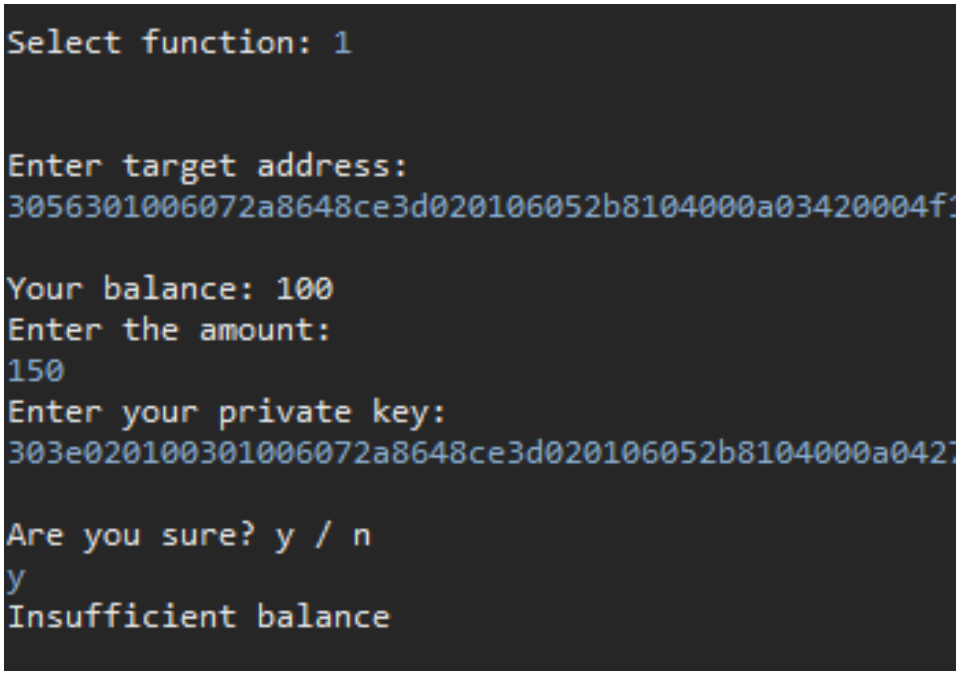
Liite 15. Testitulos, transaktion lähettäminen väärällä avaimella.

```
Select function: 1


Enter target address:
3056301006072a8648ce3d020106052b8104000a03420004f16dfb7e1ad8214625600f0833744c51f879058285655c9d2a69ab9ad4533835417a61280a22d22a8832463520fb516a9fb30a20f82317cac131195cfac78db

Your balance: 150
Enter the amount:
50
Enter your private key:
303e020100301006072a8648ce3d020106052b8104000a042730250201010420848787 38cb9dfb798848547e6339fcb1d44102ab26804344dae022c90140aa32

Are you sure? y / n
y
Invalid transaction

Functions:
1: Add a transaction
2: Mine pending transactions
3: Check account balance
4: Check if chain is valid
5: Test altering chain
6: Print blockchain

9: Logout current wallet
0: Quit
```

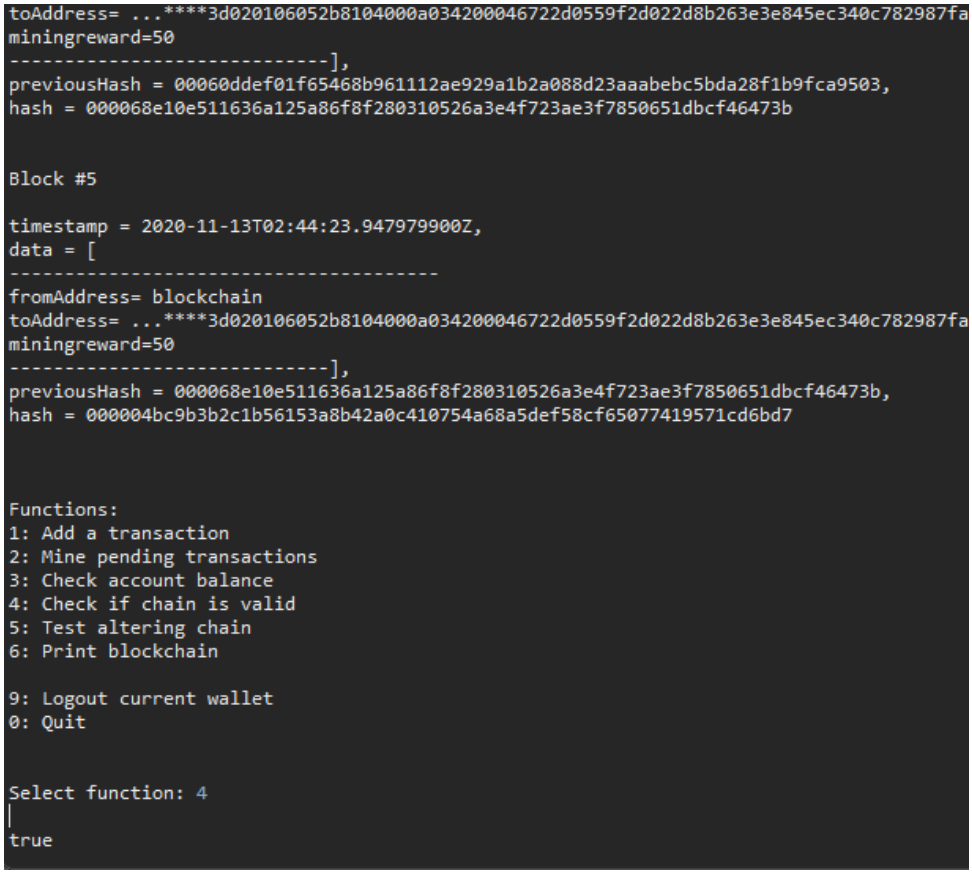
Liite 16. Testitulos, validin transaktion lisääminen järjestelmään.



Liite 17. Testitulos, rahan lähettäminen ilman tarvittavaa varallisuutta



Liite 18. Testitulos louhinnan jälkeisestä lohkojen oikeellisuudesta



Liite 19. Testitulos lohkoketjun manipuloinnin jälkeen

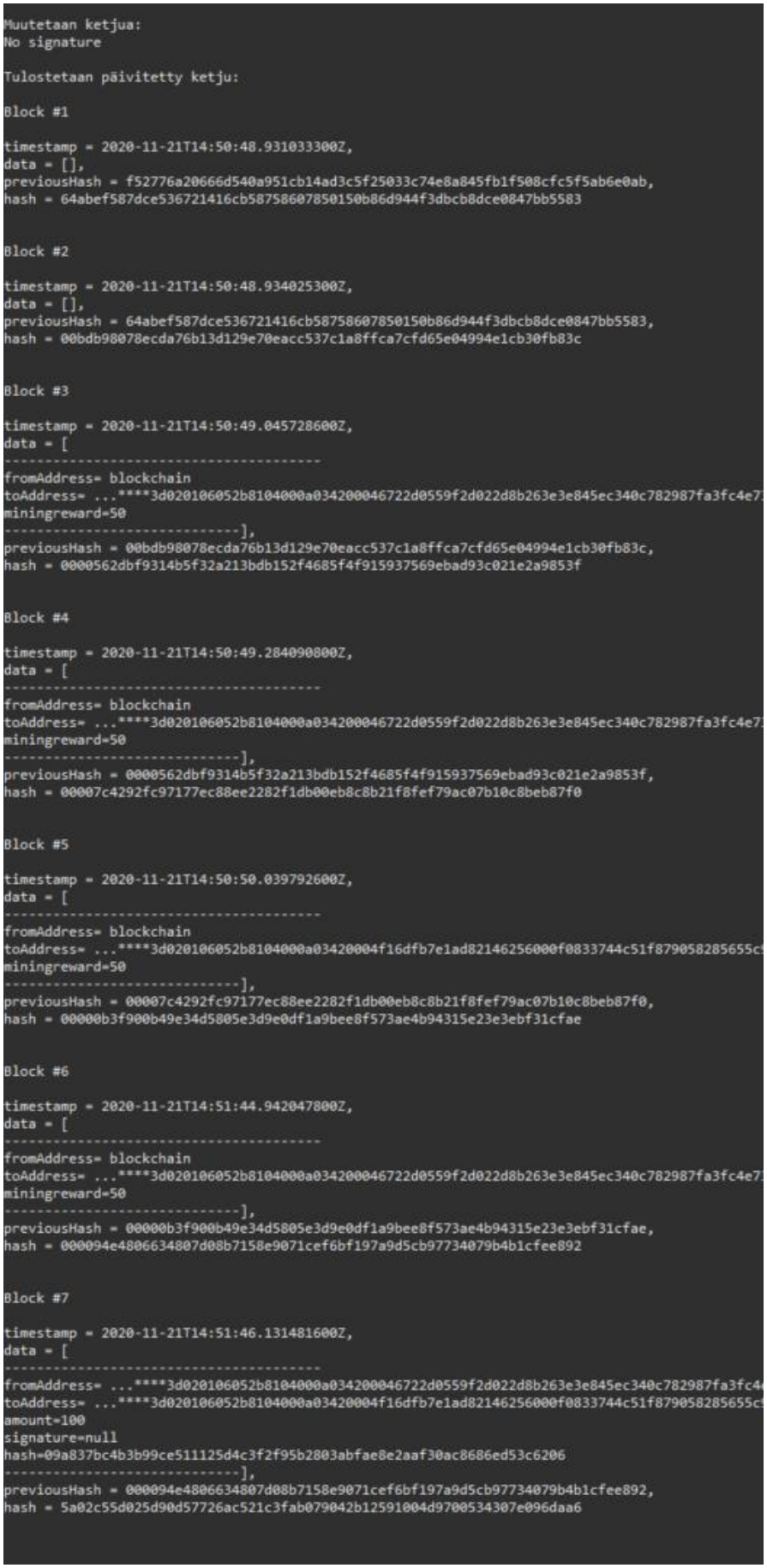
Liite 20. Testitulos, epärehellisen noden lähettämä väärennetty transaktio

```
Mining block with difficulty 2
Block 007f366b9716ceedf88f4b85cb535674b1d0f10251b3d1f388c8d40731fb2221 succesfully mined.
Block added to chain.
Took 0 seconds.
Difficulty increased to 3
Mining block with difficulty 3
Block 000ae5d7cb208091469e05eb87bec6b2038a0ef4def0e7ed0470a42bf660221b succesfully mined.
Block added to chain.
Took 0 seconds.
Difficulty increased to 4
Mining block with difficulty 4
Block 0000c1fe78764b89ad7f9bb3c4454590a3eb4e1c375ba06df5d60f9cada6b0e9 succesfully mined.
Block added to chain.
Took 2 seconds.
Mining block with difficulty 4
Block 0000ee96709e149c4d981fe4fca6795b1e34813ff268063380a4196d052ab229 succesfully mined.
Block added to chain.
Took 0 seconds.
Difficulty increased to 5
No signature
Mining block with difficulty 5
Block 00000d69d9eb991e77302cc1a594bbe5c0495fdd68430b8f571f59c2f3e4d1c8 succesfully mined.
Block added to chain.
Took 17 seconds.
Difficulty decreased to 4
```

Liite 21. Testitulos epärehellisen transaktion lähetyksestä

```
timestamp = 2020-11-13T03:14:22.411028800Z,
data = [
---------------------------------------
fromAddress= blockchain
toAddress= ...****3d020106052b8104000a03420004f16dfb7e1ad8214625600f0833744c51f879058285655c9
miningreward=50
----------------------------],
previousHash = 0000c1fe78764b89ad7f9bb3c4454590a3eb4e1c375ba06df5d60f9cada6b0e9,
hash = 0000ee96709e149c4d981fe4fca6795b1e34813ff268063380a4196d052ab229


Block #6

timestamp = 2020-11-13T03:14:22.846617800Z,
data = [
---------------------------------------
fromAddress= blockchain
toAddress= ...****3d020106052b8104000a034200046722d0559f2d022d8b263e3e845ec340c782987fa3fc4e73
miningreward=50
----------------------------],
previousHash = 0000ee96709e149c4d981fe4fca6795b1e34813ff268063380a4196d052ab229,
hash = 00000d69d9eb991e77302cc1a594bbe5c0495fdd68430b8f571f59c2f3e4d1c8
```

Liite 22. Testitulos epärehellisen lohkon lisäämisestä



Liite 23. Transaktio-luokka, osa 1

```
package demoCoin;

import demoCoin.Blockchain;
import java.security.KeyPair;
import java.security.PublicKey;
import java.security.Signature;
import java.time.Instant;

public class Transaction {

    private String fromAddress;
    private String toAddress;
    private int amount;
    private String signature;
    private String hash;

    public Transaction(String fromAddress, String toAddress, int amount) {
        super();
        this.fromAddress = fromAddress;
        this.toAddress = toAddress;
        this.amount = amount;
        this.hash = this.calculateHash();
    }


    private String calculateHash() {
        Instant now = Instant.now();
        return Cryptography.generateHash(this.fromAddress + this.toAddress + this.amount + now.toString());
    }

    public void signTransaction(KeyPair keypair) throws Exception {
        String publicKeyHex = Cryptography.toHexString(keypair.getPublic().getEncoded());

        if (!publicKeyHex.equals(this.fromAddress)) {
            throw new Exception("Cannot sign transaction for someone else's wallet.");
        }

        Signature ecdaSign = Signature.getInstance("SHA256withECDSA");
        ecdaSign.initSign(keypair.getPrivate());
        ecdaSign.update(this.hash.getBytes());
        String signature = Cryptography.toHexString(ecdaSign.sign());
        this.signature = signature;
    }


    public boolean isValid(Blockchain chain) throws Exception {

        if (this.fromAddress == null) {
            return true;

        } else if (this.signature == null || this.signature.length() == 0) {
            throw new Exception("No signature");

        } else if (Node.getBalanceOf(this.fromAddress, chain) < this.amount) {
            throw new Exception("Insufficient balance");
        } else {

        Signature ecdsaVerify = Signature.getInstance("Sha256withECDSA");
        PublicKey publicKey = Cryptography.generateFromPublic(Cryptography.hexStringToByteArray(this.fromAddress));
        ecdsaVerify.initVerify(publicKey);
        ecdsaVerify.update(this.hash.getBytes());
        boolean result = ecdsaVerify.verify(Cryptography.hexStringToByteArray(this.signature));


        return result;

        }
    }
```

Liite 24. Transaktio-luokka, osa 2

```
    public String getFromAddress() {
        if (this.fromAddress != null) {
            return fromAddress;
        }
        return null;
    }

    public void setFromAddress(String fromAddress) {
        this.fromAddress = fromAddress;
    }

    public String getToAddress() {
        if (this.toAddress != null) {
            return toAddress;
        }
        return null;
    }

    public void setToAddress(String toAddress) {
        this.toAddress = toAddress;
    }

    public int getAmount() {
        return amount;
    }

    public void setAmount(int amount) {
        this.amount = amount;
    }


    @Override
    public String toString() {
        if (this.fromAddress != null && this.toAddress != null) {
            return "\n-------------------------------------"
                    + "\nfromAddress= ...****" + fromAddress.substring(20)
                    + "\ntoAddress= ...****" + toAddress.substring(20)
                    + "\namount=" + amount
                    + "\nsignature=" + signature
                    + "\nhash=" + hash + "\n----------------------------";
        } else if (this.toAddress != null) {
            return "\n-------------------------------------"
                    + "\nfromAddress= blockchain"
                    + "\ntoAddress= ...****" + toAddress.substring(20)
                    + "\nminingreward=" + amount
                    + "\n----------------------------";
        }
        return "\n-------------------------------------"
        + "\ngenesis"
        + "\n----------------------------";
    }


}
```

Liite 25. Lohko-luokka, osa 1

```
public class Block {

    private String timestamp;
    private ArrayList<Transaction> transactions;
    private String previousHash;
    private String hash;
    private int nonce;

    public Block(ArrayList<Transaction> transactions, String previousHash) {
        super();
        this.timestamp = timestamp();
        this.transactions = transactions;
        this.previousHash = previousHash;
        this.nonce = 0;

        String hash = calculateHash();
        if (hash != "") {
            this.hash = hash;
        }
    }


    public String timestamp() {
        Instant now = Instant.now();
        return(now.toString());
    }


    public String calculateHash() {
        String hash = "";
        hash = Cryptography.generateHash(this.previousHash
                + this.timestamp
                + this.transactions
                + this.nonce);
        return hash;
    }


    public void mineBlock(int difficulty) {

        String difficultyGoal = "";
        for (int i = 0; i < difficulty; i++) {
            difficultyGoal += "0";
        }

        System.out.println("Mining block with difficulty " + difficulty);

        while (!this.hash.substring(0, difficulty).equals(difficultyGoal)) {
            this.nonce++;
            this.hash = this.calculateHash();
        }
        System.out.println("Block " + this.hash + " succesfully mined.");
    }


    public boolean hasValidTransactions(Blockchain chain) throws Exception {
        for (Transaction tx : this.transactions) {
            if (!tx.isValid(chain)) {
                return false;
            }
        }

        return true;
    }
```

Liite 26. Lohko-luokka, osa 2

```
    public String getTimestamp() {
        return timestamp;
    }



    public void setTimestamp(String timestamp) {
        this.timestamp = timestamp;
    }


    public ArrayList<Transaction> getTransactions() {
        return transactions;
    }


    public void setTransactions(ArrayList<Transaction> transactions) {
        this.transactions = transactions;
    }


    public int getNonce() {
        return nonce;
    }


    public void setNonce(int nonce) {
        this.nonce = nonce;
    }


    public String getPreviousHash() {
        return previousHash;
    }



    public void setPreviousHash(String previousHash) {
        this.previousHash = previousHash;
    }



    public String getHash() {
        return hash;
    }



    public void setHash(String hash) {
        this.hash = hash;
    }



    @Override
    public String toString() {
        return "\ntimestamp = " + timestamp
                + ", \ndata = " + transactions
                + ", \npreviousHash = " + previousHash
                + ", \nhash = " + calculateHash() + "\n";
    }


}
```

Liite 27. Lohkoketju-luokka

```
public class Blockchain {

    private ArrayList<Block> blockchain;
    private ArrayList<Transaction> pendingTransactions;
    private int difficulty;
    private int miningReward;
    private long miningMilliseconds;

    public Blockchain() {
        super();
        this.difficulty = 2;
        this.miningReward = 50;
        this.blockchain = new ArrayList<>();
        this.pendingTransactions = new ArrayList<>();
        this.blockchain.add(this.createGenesisBlock());
        this.miningMilliseconds = 10000;
    }

    public Block createGenesisBlock() {
        this.pendingTransactions.add(new Transaction(null, null, this.miningReward));
        return new Block(
                new ArrayList<Transaction>()
                , Cryptography.toHexString(Cryptography.getSHA("Ei ole edellistä")));
    }

    public ArrayList<Block> getBlockchain() {
        return blockchain;
    }

    public void setBlockchain(ArrayList<Block> blockchain) {
        this.blockchain = blockchain;
    }

    public ArrayList<Transaction> getPendingTransactions() {
        return pendingTransactions;
    }

    public void setPendingTransactions(ArrayList<Transaction> pendingTransactions) {
        this.pendingTransactions = pendingTransactions;
    }

    public int getDifficulty() {
        return difficulty;
    }

    public void increaseDifficulty() {
        this.difficulty++;
    }

    public void decreaseDifficulty() {
        this.difficulty--;
    }

    public int getMiningReward() {
        return miningReward;
    }

    public void setMiningReward(int miningReward) {
        this.miningReward = miningReward;
    }

    public long getMiningMilliseconds() {
        return miningMilliseconds;
    }

}
```

Liite 28. Node-luokka, osa 1

```
public class Node {

    private Blockchain blockchain;

    public Node(Blockchain blockchain) {
        super();
        this.blockchain = blockchain;
    }

    public boolean isChainValid() throws Exception {
        for (int i = 1; i < this.blockchain.getBlockchain().size(); i++) {

            Block current = this.blockchain.getBlockchain().get(i);
            Block previous = this.blockchain.getBlockchain().get(i - 1);

            if (!current.getHash().equals(current.calculateHash())) {
                System.out.println("Invalid block hash : " + i);
                System.out.println(current.getHash());
                System.out.println(current.calculateHash());
                System.out.println();
                return false;
            }
            if (!current.getPreviousHash().equals(previous.getHash())) {
                return false;
            }
            if (!current.hasValidTransactions(this.blockchain)) {
                return false;
            }
        }
        return true;
    }

    public void refreshChain(Blockchain chain) {
        String previousHash = "f52776a20666d540a951cb14ad3c5f25033c74e8a845fb1f508cfc5f5ab6e0ab";
        for (Block b : chain.getBlockchain()) {
            b.setPreviousHash(previousHash);
            b.setHash(b.calculateHash());
            previousHash = b.calculateHash();
        }
    }


    public Block findLastValidBlock() throws Exception {
        for (int i = 1; i < this.blockchain.getBlockchain().size(); i++) {

            Block current = this.blockchain.getBlockchain().get(i);
            Block previous = this.blockchain.getBlockchain().get(i - 1);

            if (!current.getHash().equals(current.calculateHash())) {          // Tarkistetaan onko lohkon
                System.out.println("Invalid block hash at chain index: " + i);  // tiiviste muodostettu oikein
                System.out.println(current.getHash());
                System.out.println(current.calculateHash());
                System.out.println();
                return previous;
            }

            if (!current.getPreviousHash().equals(previous.getHash())) {    // Tarkistetaan täsmääkö lohkon
                return previous;                                            // lohkon "edellinen tiiviste"
            }                                                               // edellisen lohkon tiivisteeseen

            if (!current.hasValidTransactions(this.blockchain)) {       // Tarkistetaan lohkon transaktiot
                return previous;
            }
        }
        return this.blockchain.getBlockchain().get(this.blockchain.getBlockchain().size() - 1);
    }
```

Liite 29. Node-luokka, osa 2

```
public static int getBalanceOf(String address, Blockchain chain) {

    int balance = 0;

    for (Block b : chain.getBlockchain()) {            // Lasketaan saldo lohkoketjun transaktioista
        for (Transaction t : b.getTransactions()) {
            if (t.getFromAddress() != null) {
                if (t.getFromAddress().equals(address)) {
                    balance -= t.getAmount();
                }
                if (t.getToAddress().equals(address)) {
                    balance += t.getAmount();
                }
            } else if (t.getToAddress() != null){
                if (t.getToAddress().equals(address)) {
                    balance += t.getAmount();
                }
            }
        }
    }

    for (Transaction t : chain.getPendingTransactions()) {  // Lasketaan saldo myös
        if (t.getFromAddress() != null) {                   // odottavista transaktioista
            if (t.getFromAddress().equals(address)) {
                balance -= t.getAmount();
            }
            if (t.getToAddress().equals(address)) {
                balance += t.getAmount();
            }
        } else if (t.getToAddress() != null){
            if (t.getToAddress().equals(address)) {
                balance += t.getAmount();
            }
        }
    }
    return balance;
}

public void addTransaction(Transaction tx) throws Exception {

    try {
        if (tx.getFromAddress() == null                 // Varmistetaan, että lisättävään transaktioon
            || tx.getFromAddress().length() == 0        // on merkitty lähettäjä ja vastaanottaja
            || tx.getToAddress() == null
            || tx.getToAddress().length() == 0 ) {

            throw new Exception("Must include from and to address");

        } else if (!tx.isValid(this.getBlockchain())) {     // Käytetään transaktio-luokan
            throw new Exception("Invalid transaction");     // isValid() -metodia
                                                            // allekirjoituksen tarkistamiseen.

        } else if (Node.getBalanceOf(tx.getFromAddress(), this.blockchain) < tx.getAmount()) {
                                                            // Varmistetaan, että transaktion
            throw new Exception("Insufficient wealth");     // lähettäjällä on tarvittava varallisuus

        } else {
            this.blockchain.getPendingTransactions().add(tx);   // Jos kaikki on ok,
                                                                // lisätään transaktio
                                                                // lohkoketjuun odottavien listalle.
        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

Liite 30. Node-luokka, osa 3

```
public void minePendingTransactions(String rewardAddress) throws Exception {

    ArrayList<Transaction> transactions = new ArrayList<>();

    for (Transaction t : this.blockchain.getPendingTransactions()) {    // Tarkastaa transaktiot ja käyttäjien varallisuuden

        if (t.getFromAddress() != null) {
            if (Node.getBalanceOf(t.getFromAddress(), this.blockchain) > t.getAmount()) {
                transactions.add(t);
            }
        } else if (t.getFromAddress() == null && t.getToAddress() != null) {
            transactions.add(t);
        }
    }

    Block block = new Block(transactions, this.findLastValidBlock().getHash()); // Etsii viimeisimmän validin lohkon

    long start = System.currentTimeMillis();                                    // Ajanlaskutoiminnallisuus

    block.mineBlock(this.blockchain.getDifficulty());                           // Aloittaa lohkon louhinnan

    long end = System.currentTimeMillis();
    long elapsedTime = end - start;

    this.blockchain.getBlockchain().add(block);                                 // Louhinnan valmistuttua lisää lohkon ketjuun
    System.out.println("Block added to chain.");

    this.blockchain.setPendingTransactions(new ArrayList<Transaction>());       // Tyhjentää listan
    this.blockchain.getPendingTransactions().add(                               // Lisää uudelle listalle louhintapalkkion
            (new Transaction(null, rewardAddress, this.blockchain.getMiningReward())));

    if (elapsedTime < (this.blockchain.getMiningMilliseconds() * 0.2)) {                    // Toiminnallisuus louhinnan
        this.blockchain.increaseDifficulty();                                               // ajan säätämisen
        System.out.println("Took " + (elapsedTime / 1000)                                   // simuloimiseksi
                + " seconds. \nDifficulty increased to " + this.blockchain.getDifficulty());
    } else if (elapsedTime > (this.blockchain.getMiningMilliseconds() * 1.2)) {
        this.blockchain.decreaseDifficulty();
        System.out.println("Took " + (elapsedTime / 1000)
                + " seconds. \nDifficulty decreased to " + this.blockchain.getDifficulty());
    } else {
        System.out.println("Took " + (elapsedTime / 1000) + " seconds.");
    }
}


public void printBlockchain() {

    int index = 1;

    for (Block b : this.blockchain.getBlockchain()) {

        System.out.println("Block #" + index);
        System.out.println(b);
        System.out.println();
        index++;

    }
}
```

Liite 31. Node-luokka, osa 4

```
public void printUpdatedBlockchain() {

    Blockchain updated = new Blockchain();
    updated.setBlockchain(new ArrayList<>(this.getBlockchain().getBlockchain()));
    refreshChain(updated);

        int index = 1;

        for (Block b : updated.getBlockchain()) {

            System.out.println("Block #" + index);
            System.out.println(b);
            System.out.println();
            index++;

        }
    }

    public Blockchain getBlockchain() {
        return blockchain;
    }

    public void setBlockchain(Blockchain blockchain) {
        this.blockchain = blockchain;
    }



}
```

Liite 32. Kryptografia-luokka, osa 1

```
public class Cryptography {


    public static KeyPair generateKeypair() {

        try {
            ECGenParameterSpec ecSpec = new ECGenParameterSpec("secp256k1");
            KeyPairGenerator g = KeyPairGenerator.getInstance("EC");
            g.initialize(ecSpec, new SecureRandom());
            KeyPair keypair = g.generateKeyPair();
            return keypair;
        } catch (Exception e) {
            System.out.println(e);
        }

        return null;

    }

    public static PrivateKey generateFromPrivate(byte[] privateKey) {

        try {

        KeyFactory f = KeyFactory.getInstance("EC");
        EncodedKeySpec spec = new PKCS8EncodedKeySpec(privateKey);
        PrivateKey key = f.generatePrivate(spec);

        return key;

        } catch (Exception e) {
            System.out.println(e);
        }

        return null;

    }

    public static PublicKey generateFromPublic(byte[] publicKey) {

        try {

        KeyFactory f = KeyFactory.getInstance("EC");
        EncodedKeySpec spec = new X509EncodedKeySpec(publicKey);
        PublicKey key = f.generatePublic(spec);

        return key;

        } catch (Exception e) {
            System.out.println(e);
        }

        return null;

    }


    public static KeyPair generatePairFromHex(String publicHex, String privateHex) throws Exception {

        PublicKey pu = generateFromPublic(hexStringToByteArray(publicHex));
        PrivateKey pr = generateFromPrivate(hexStringToByteArray(privateHex));

        return new KeyPair(pu, pr);

    }
```

Liite 33. Kryptografia-luokka, osa 2

```
    public static String generateHash(String input) {
        return toHexString(getSHA(input));
    }


    public static byte[] getSHA(String input)
    {
        MessageDigest md;
        try {
            md = MessageDigest.getInstance("SHA-256");

            return md.digest(input.getBytes(StandardCharsets.UTF_8));

        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }

        return null;
    }



    public static String toHexString(byte[] hash)
    {

        StringBuilder hexString = new StringBuilder();

        for (byte b : hash) {
            hexString.append(String.format("%02x", b));
        }



        return hexString.toString();
    }



    public static byte[] hexStringToByteArray(String s) {
        int len = s.length();
        byte[] data = new byte[len / 2];
        for (int i = 0; i < len; i += 2) {
            data[i / 2] = (byte) ((Character.digit(s.charAt(i), 16) << 4)
                                 + Character.digit(s.charAt(i+1), 16));
        }
        return data;
    }


}
```

Liite 34. Main-luokka, osa 1

```
public class Main {

    public static void main(String[] args) throws Exception {

        // Tämä on yksinkertainen käyttöliittymä testaamista varten.

        Blockchain ketju = new Blockchain();


        Node node = new Node(ketju);
        String rewardAddress1 = "305630100607 2a8648ce3d020106052b8104000a034200046722d0559f2d022d8b263e3e845ec


        Scanner s = new Scanner(System.in);
        boolean kaynnissa = true;


        String walletAddress = "";

        while (kaynnissa) {

            if (walletAddress == "") {
                System.out.println("Enter wallet address: ");
                walletAddress = s.next();
            }

            System.out.println();
            System.out.println("Functions:\n"
                    + "1: Add a transaction\n"
                    + "2: Mine pending transactions\n"
                    + "3: Check account balance\n"
                    + "4: Check if chain is valid\n"
                    + "5: Test altering chain\n"
                    + "6: Print blockchain\n"
                    + "\n"
                    + "9: Logout current wallet\n"
                    + "0: Quit\n");
            System.out.print("\nSelect function: ");


            int input = s.nextInt();

            System.out.println();

            switch (input) {

                case 0:
                    kaynnissa = false;
                    break;

                case 1:
                    try {

                        System.out.println("\nEnter target address: ");
                        String to = s.next();
                        int balance = Node.getBalanceOf(walletAddress, node.getBlockchain());
                        System.out.println("\nYour balance: " + balance);
                        System.out.println("Enter the amount: ");
                        int amount = s.nextInt();
                        System.out.println("Enter your private key: ");
                        String pvtKey = s.next();
                        System.out.println("\nAre you sure? y / n");
```

Liite 35. Main-luokka, osa 2

```
                    if (s.next().equals("y")) {
                        KeyPair kp = Cryptography.generatePairFromHex(walletAddress, pvtKey);
                        Transaction tx = new Transaction(walletAddress, to, amount);
                        tx.signTransaction(kp);
                        node.addTransaction(tx);
                        break;
                    }

                } catch (Exception e) {
                    System.out.println(e.getMessage());
                    break;
                }
                break;


            case 2:
                String rewardTo = walletAddress;
                node.minePendingTransactions(rewardTo);
                break;

            case 3:
                System.out.println("\nYour balance: " + Node.getBalanceOf(walletAddress,
                        node.getBlockchain()));
                break;

            case 4:
                System.out.println(node.isChainValid());
                break;

            case 5:
                System.out.println("Enter block index: ");
                int index = s.nextInt();
                System.out.println("Enter number: ");
                int newNonce = s.nextInt();
                node.getBlockchain().getBlockchain().get(index).setNonce(newNonce);
                break;

            case 6:
                node.printBlockchain();
                break;

            case 7:
                node.printUpdatedBlockchain();
                break;

            case 9:
                walletAddress = "";
                break;


            default:
                System.out.println("unknown command");
                continue;

        }

    }
```

Liite 36. Main-luokka, osa 3

```
    /*  Testikoodia ilman käyttöliittymää
     *
    Node node2 = new Node(ketju);
    String rewardAddress2 = "305630100607 2a8648ce3d020106052b8104000a03420004f16dfb7e1ad8214625600f0833744c51f879058285655c9d2a69ab9ad4533835417a61280a22d22a8832463520fb516a9fb30a20f82317cac131195cfac78db";

    Node node3 = new Node(ketju);

    node.minePendingTransactions(rewardAddress1);
    node.minePendingTransactions(rewardAddress1);
    node2.minePendingTransactions(rewardAddress2);
    node.minePendingTransactions(rewardAddress1);
    Node.getBalanceOf(rewardAddress1, ketju);
    Node.getBalanceOf(rewardAddress2, ketju);

    node3.addTransaction(new Transaction(rewardAddress1, rewardAddress2, 100));

    node2.minePendingTransactions(rewardAddress2);

    node.printBlockchain();

    System.out.println(node.isChainValid());

    System.out.println();
    System.out.println("Muutetaan ketjua:");

    node3.getBlockchain().getBlockchain().add(
            new Block(
                    new ArrayList<Transaction>(Arrays.asList(
                            new Transaction(rewardAddress1, rewardAddress2, 100)))
                    , node3.findLastValidBlock().getHash()));

    try {
        System.out.println(node.isChainValid());
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }


    System.out.println("\nTulostetaan päivitetty ketju:\n");
    node.printUpdatedBlockchain();
    */

}
}
```