Matti Räsänen

LIIKENTEENOHJAUSJÄRJESTELMÄN PÄIVITYSJAKELUN SUUNNITTELU JA PROTOTYYPIN KEHITYS

Opinnäytetyö
Tietojenkäsittelyn koulutus

2017

Kaakkois-Suomen
ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tekijä/Tekijät	Tutkinto	Aika
Matti Räsänen	Tradenomi (AMK)	Kesäkuu 2017
Opinnäytetyön nimi Liikenteenohjausjärjestelmän päivitysjakelun suunnittelu ja prototyypin kehitys		38 sivua
Toimeksiantaja Mipro Oy		
Ohjaaja Jukka Selin		

Tiivistelmä

Tämän opinnäytetyön aiheena on Liikenteenohjausjärjestelmän päivitysjakelun suunnittelu ja prototyypin kehitys. Työn toimeksiantajana toimii Mipro Oy. Mipro on turvallisuuteen ja ympäristötekniikkaan erikoistunut asiantuntijayritys.

Opinnäytetyöni tavoitteena on kehittää prototyyppi päivitystenjakelusta liikenteenohjausjärjestelmään. Opinnäytetyössä perehdytään järjestelmän kannalta oleellisiin ajoympäristöihin ja tekniikkoihin sekä tarkastellaan salausalgoritmeja ja tarkistussummaa.

Työn lopputulos on prototyyppi päivitystenjakelusta liikenteenohjausjärjestelmään Windows-käyttöjärjestelmille. Prototyyppi kehitettiin järjestelmän komponentin yhteyteen, jonka avulla hallitaan järjestelmän käynnistämistä sekä sammuttamista. Toteutuksessa keskitytään käyttöliittymäpuolen komponenttien päivitysten jakeluun. Teknisen toteutuksen lisäksi on kuvailtu käyttötapaus päivittämisprosessista. Opinnäytetyö toteutettiin Java-ohjelmointikielellä.

Päivitys käynnistetään järjestelmästä ylläpitäjän oikeuksilla. Päivityksen käynnistämiseen kehitettiin kaksi eri menetelmää, pakotettu päivitys ja käyttäjälle lähetettävä päivityspyyntö.

Asiasanat

Java, olio-ohjelmointi, algoritmit

Author (authors)	Degree	Time
Matti Räsänen	Bachelor of Business Administration	June 2017
Thesis Title Design and development of an update distribution for a traffic control system		38 pages
Commissioned by Mipro Oy		
Supervisor Jukka Selin		

Abstract

The subject of this thesis was the design and development of an update distribution system for a railway traffic control system. The thesis was assigned by Mipro Oy, specialised in railway and industrial systems.

The requirements for the implementation of the update distribution system was that it should be faster than the current implementation and easy to use. The safety was also one main requirement for the new implementation.

The thesis studied the system execution environment and frameworks, and also cryptography, hashing algorithms and checksum in the thesis. The end result of the thesis was a prototype of the update distribution for a traffic control system on the Windows platform and on the Linux platform. The prototype focused on the update distribution for the client side components of the system.

The prototype was developed with Java technologies. The update process can be started from the system by an administrator. There are two methods for starting the update: a forced update and an update notification sent to the user.

Keywords

Java, object-orientated programming, algorithms

SISÄLLYS

KÄSITTEET

Bundle: Kokoelma Java-luokkia yhtenä komponenttina, joista muodostuu yksi osa ohjelmaa

JAR (Java ARchive): Pakattu tiedosto, joka tyypillisesti sisältää useita Java-luokkia ja niihin liittyvää metadataa sekä lähteitä kuten tekstiä, kuvia. JAR-tiedosto on pakattu ZIP-formaattiin

OSGI (Open Services Gateway Initiative): Java-ohjelmointikielelle kehitetty standardi, jonka avulla voidaan luoda dynaamisia moduuliohjelmia

Päivityspaketti: Sisältää ohjelman version päivittämiseen tarvittavat komponentit. Tässä työssä tarkoitetaan ZIP tai ISO pakettia

Socket: Päätepiste datan lähettämistä tai vastaanottamista varten verkon välityksellä

Tarkistussumma: Summa, jonka laskemalla voidaan varmistaa tietojen eheys sekä oikeellisuus. Tarkistussumma lasketaan ennen siirtoa ja siirron jälkeen

TMS (Traffic Management System): Mipron kehittämä tietokonepohjainen liikenteenohjausjärjestelmä rautatieliikenteeseen

1 JOHDANTO

Opinnäytetyöni toimeksiantaja on Mipro Oy. Mipro on turvallisuuteen ja ympäristötekniikkaan erikoistunut asiantuntijayritys. Opinnäytetyöni tavoitteena on kehittää prototyyppi päivityspakettien jakamisesta liikenteenohjausjärjestelmään. Nykyinen toimintatapa järjestelmän päivittämisessä on päivittää yksi järjestelmän osa kerrallaan. Prosessi on aikaa vievää ja mahdollistaa virheiden tapahtumista. Opinnäytetyössäni kehittämän ratkaisun pitää olla nopeampi ja luotettavampi kuin nykyinen ratkaisu. Tarvittaessa järjestelmä pitää pystyä palauttamaan myös aiempiin versioihin.

Luvussa 2 kerrotaan toimeksiantajasta ja sen tuotteista. Luvussa 3 käydään läpi liikenteenohjausjärjestelmän toiminnan kannalta oleellisia ajoympäristöjä ja tekniikoita. OSGi-kehystä tarkastellaan teoriatasolla ja sen käyttämisestä Apache Felix -ympäristössä. Luvussa 4 käsitellään salaamista yleisesti ja esitellään eri salausalgoritmeja. Kerron myös tarkistussummasta ja miten salausalgoritmit liittyvät sen toimintaan.

Luvussa 5 kerrotaan järjestelmän toiminnasta, mahdollisista toteutustavoista sekä lopullisesta toteutuksesta. Toteutuksesta kerrotaan teknisestä näkökulmasta sekä kuvaillaan päivitysprosessin käyttötapaus. Kehitystyö tehtiin järjestelmässä valmiina olevan komponentin yhteyteen, jonka avulla voidaan hallita järjestelmän käynnistämistä sekä sammuttamista. Päivitys koostuu kuudesta pääprosessista: päivityksen käynnistäminen, päivityspaketin lataus, päivityspaketin tarkistaminen, järjestelmän sammutus, päivityspaketin asennus ja järjestelmän käynnistys.

Päivityksen käynnistäminen suoritetaan etäyhteyden avulla. Käynnistämiseen kehitetään kaksi eri vaihtoehtoa pakotettu käynnistys ja käyttäjälle lähetettävä päivityspyyntö. Päivityspaketti ladataan verkosta tai muusta sijainnista ja tarkistetaan, jonka jälkeen järjestelmä sammutetaan. Päivityspaketti asennetaan ja järjestelmä käynnistetään.

2 MIPRO OY

Tämän luvun tiedot perustuvat Mipron (2016) internetsivustoon. Mipro on turvallisuuteen ja ympäristötekniikkaan erikoistunut asiantuntijayritys. Yrityksen pääkonttori on Mikkelissä, mutta se toimii koko Suomessa sekä Itä-Euroopassa ja Lähi-Idässä. Mipro kuuluu Mipro Group -konserniin ja sen sisaryhtiö on Censeo Oy. Mipro perustettiin vuonna 1980 nimellä Mikkelin Prosessiohjaus Ky. Aluksi yrityksen toimialana oli prosessiteollisuuden sekä vesihuollon automaatiosuunnittelu ja asennukset. Yritys alkoi panostamaan turvallisuuteen liittyviin järjestelmiin ja erityisesti rautatiejärjestelmiin 1990-luvun alussa. Ensimmäinen tasoristeysjärjestelmä toimitettiin vuonna 1995.

Miprolla on omat tuotteensa eri toimialoille. MiSO TCS -asetinlaitejärjestelmä (Traffic Control System), joka huolehtii ratojen ohjauksesta ja muista turvallisuuteen liittyvistä toiminnoista. Mipro TMS (Traffic Management System) -liikenteenohjausjärjestelmä, joka integroituu saumattomasti erityyppisiin asetinlaitejärjestelmiin. Mipro TMS tarjoaa yhtenäisen käyttöliittymän ja joustavan käytettävyyden erilaisiin liikennetilanteisiin.

Järjestelmä sisältää paljon automatiikkaa, joka helpottaa liikenteenohjaajan työtä ja estää virheitä. Järjestelmässä on myös simulaattori, jota käytetään mm. kouluttamisessa ja testaamisessa. Mipro ATS (Automatic Train Supervision) -käyttöohjausjärjestelmä on tietokonepohjainen raideliikenteen ohjaus- ja hallintajärjestelmä.

Mipro REGO on karttapohjainen tilannekuvajärjestelmä, joka kerää tietoa eri järjestelmistä ja muodostaa niistä yhtenäisen kokonaiskuvan. Järjestelmän avulla pystytään tekemään parempia päätöksiä häiriötilanteissa. Mipro REGO käytetään mm. laboratoriojärjestelmissä, kunnossapidon hallintajärjestelmissä sekä automaatio- ja valvomojärjestelmissä.

3 AJOYMPÄRISTÖT JA TEKNIIKAT

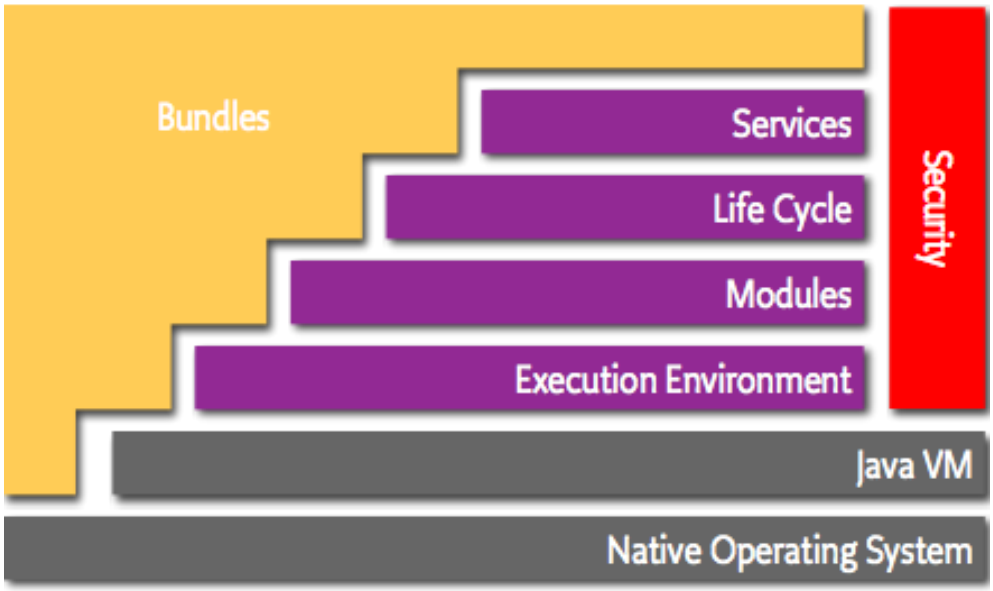
Tässä luvussa käydään läpi ajoympäristöjä, jotka liittyvät liikenneohjausjärjestelmän toimintaan. Ensimmäisessä luvussa käydään läpi OSGi-kehystä teoriatasolla. Toisessa luvussa käsitellään OSGi-kehyksen implementointia Apache Felix -ympäristössä.

3.1 OSGi

Open Services Gateway initiative eli OSGi kehitettiin vuonna 1999. OSGi on avoimen lähdekoodin omaava kehys, jonka alkuperäinen käyttötarkoitus oli Java teknologian mukauttaminen kotiverkoissa. Kehyksen avoinlähdekoodi on luonut sille suuren suosion, jonka myötä se on saanut tuekseen suuria tekijöitä markkinoilta. (Gedeon 2010,8.)

OSGi-kehys on kehitetty Java-ohjelmointikielelle ja sen avulla voidaan luoda dynaamisia moduuliohjelmia. OSGi muodostuu paketeista joita kutsutaan OSGi-bundleiksi. Bundle on JAR-tiedosto, johon on liitetty manifesti. Manifesti sisältää ajoympäristön tarvitsemat otsikkotiedot. Kehyksessä keskitytään funkionaalisuuteen, joka mahdollistaa bundlejen toiminnan itsenäisesti elinkaaren kanssa. Komponentit on ryhmitelty eri kerroksiin niiden toiminnallisuuksien perusteella. (Osgi Alliance 2017.)

Ylimpänä on Services-kerros, jonka avulla bundlet keskustelevat keskenään. Services-kerros kirjaa ylös rekisteröidyt palvelut. Life Cycle -kerros pitää yllä bundlejen elinkaarta sekä mahdollistaa bundlejen asentamisen ja sammuttamisen. Modules-kerros säilyttää bundleja, jotka ovat asennettu. Alimpana kerroksena on Execution Enviroment eli ajoympäristö. (Gedeon 2010, 8–10.) Kuva 1 havainnollistaa toiminnalliset kerrokset.



KUVA 1. OSGi toiminnalliset kerrokset (OSGi Alliance 2017)

Kaksi yleisintä ajoympäristöä ovat CDC Foundation ja Java SE. Security-kerros laajentaa Java 2 -turvallisuusarkkitehtuurin, mutta se ei ole pakollinen. Security-kerroksen ollessa aktiivinen, se vahvistaa bundlejen kirjautumiset sekä hallinoi komponenttin pääsyoikeuksia. (Gedeon 2010, 8–10.)

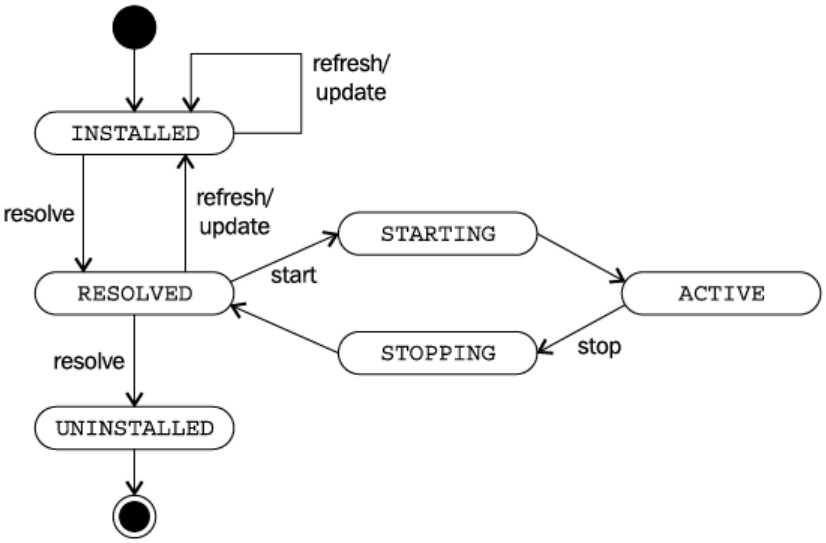
Services

Pelkällä Javan luokkajakamisella olisi hankalaa kehittää yhteistä mallia. Tyypillinen ratkaisu Java-ohjelmoinnissa olisi suorittaa luokkalataus dynaamisesti sekä käyttää staattisia muuttujia. Services-kerroksen avulla bundle voi rekisteröidä minkä tahansa objektin palvelukseen. Objekti rekisteröidään OSGi-kehyksen palvelurekisteriin. Rekisteri sallii muiden bundlejen etsiä sieltä objekteja, jotka ovat rekisteröity sinne. (Osgi Alliance 2017.)

Palvelut ovat dynaamisia, joten bundleja voidaan asentaa tai poistaa lennosta ilman, että se haittaisi muiden bundlejen käyttämistä. Service Tracker huolehtii siitä, että bundlet jotka eivät käytä palveluita, eivät myöskään käytä mitään resursseja. Useat ongelmat ovat helppoja tunnistaa dynaamisten palveluiden ja staattisten muuttujien avulla, koska oikea maailma on myös dynaaminen. (Osgi Alliance 2017.)

Bundlen elinkaari

Elinkaari alkaa bundlen asennusvaiheesta ja elinkaarella on kuusi eri tilaa. Elinkaaren tiloja ovat asennettu, ratkaistu, käynnistymässä, aktiivinen, sammumassa ja poistettu. Kuvassa 2 on esitelty elinkaaren tilat. (Gedeon 2010, 12.)



KUVA 2. Elinkaaren tilat (Gedeon 2010)

Asennettu (installed) bundle on asennettu onnistuneesti. OSGi-kehys tietää riittävästi bundlesta, jotta se voidaan ladata. Ratkaistu (resolved) tarvittavat resurssit ovat ladattu ja bundle on valmis käynnistykseen. Bundle menee myös em. tilaan onnistuneen sammuttamisen jälkeen. Käynnistymässä (starting) bundle on käynnistymässä. Aktiivinen (active) bundlen käynnistys on onnistunut ja se on valmis käytettäväksi. Sammumassa (stopping) bundlen sammuminen on käynnissä. Poistettu (uninstalled) bundle on poistettu, jonka jälkeen moduulilla ei voida tehdä enää mitään. (Atlassian Developers 2017.)

3.2 OSGi ja Apache Felix

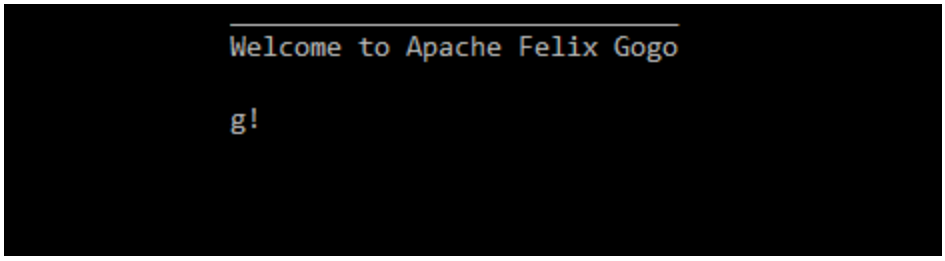
OSGi-kehys on mahdollista implementoida monella eri tavalla. OSGi:n (2017) mukaan suositumpia implementaatioita ovat Apache Felix, Eclipse Equinox ja Knoplerfish. Apache Felix on avoimen lähdekoodin implementaatio OSGi-kehykselle, jonka on kehittänyt Apache Software Foundation (ASF) Oscar-projektin

pohjalta. Eclipsen kehittämä Equinox on todella suosittu, koska se integroituu hyvin Eclipse-kehittimen kanssa. Knoplerfish on myös avoimen lähdekoodin implementaatio OSGi-kehykselle ja sen on kehittänyt Makewawe. Makewawe on ollut osallisena OSGi:n kehityksessä ensimmäisestä päivästä lähtien. (OSGi Alliance 2017.) Apache Felix on kehittänyt alihankkeiden avulla hyödyllisiä liitännäisiä (Gedeon, 27–28).

Liitännäisiä:

- **Log Service:** Lokipalvelu, jota bundlet voivat käyttää virhetilanteissa lokitukseen. Lokitiedot ovat suuressa asemassa ongelmatilanteissa, koska se helpottaa rajaamaan ongelmakohtaa sovelluksessa.
- **Http Service:** Tarjoaa HTTP-rajapinnan bundleille. Bundlet voivat olla yhteydessä verkon käyttäjiin käyttämällä XML- tai HTML-teknologioita.
- **Configuratio Admin Service:** Palvelua käytetään konfirmoitavien komponenttien datan konfirmointiin.
- **Dependency Manager:** Bundlet vaikuttavat yleensä palveluiden kautta ja tämä luo haasteita riippuvuuksien hallinnassa. Dependency Manager tarjoaa API:n, joka mahdollistaa riippuvuuksien muuttamisen dynaamisesti.
- **File Install:** Tarjoaa hakemistopohjaisen bundlen asennusmahdollisuuden.
- **Gogo:** Vaihtoehtoinen komentorivi OSGI-kehyksen käyttämiseen
- **OSGi Bundle Repository Service:** Bundlejen etäsäilytyspaikka, joka pitää listaa käyttöönotetuista ja asennetuista bundleista. Palvelu hallitsee myös bundlejen riippuvuuksien käyttöönoton.
- **Shell Services:** Mahdollistaa bundlejen paikallisesti ja etänä asentamisen yksinkertaisen komentorivin konsolin kautta.
- **Web Console Service:** Tarjoaa graafisen käyttöliittymän sovelluskehyksen hallintaan. Bundlejen asennus tätä kautta on yhtä helppoa kuin liitetiedoston lisääminen sähköpostiin.

OSGi-kehyksen implementointiin Felix:llä tarvitaan Java Development Kit (JDK) ja Felix Framework Distribution. Java Development Kit on kehitysympäristö Java-ohjelmointikielelle ja sen on kehittänyt Oracle. Felix Framework Distribution on ZIP-muotoon pakattu arkisto. Felix voidaan käynnistää esimerkiksi käyttämällä Windows-käyttöjärjestelmän komentotulkkia. Felix käyttää oletuksena Gogo-lisäpalvelua komentorivi ympäristönä. Käynnistäessä Felix komentotulkista se käyttää Gogo-lisäpalveluna oletuksena. (Gedeon 2010, 26–27.) Kuvassa 3 ilmenee Felix Gogo aloitusruutu.



KUVA 3. Apache Felix Gogo aloitusruutu

Java Development Kitin ja Felix Framework Distributionin lisäksi on suositeltavaa asentaa Maven. Maven on Apachen kehittämä työkalu Java-ohjelmistoille, joka yksinkertaistaa riippuvuuksien hallinnan ja toimii samalla kääntöympäristönä. Maven ei ole pakollinen kääntöympäristö OSGi-bundleille, mutta se helpottaa OSGi-bundlejen kääntämistä. Perusperiaate on prosessit, jotka määrittelevät tietyt vaiheet saavuttaakseen lopullisen tuloksen. Maven sisältää paljon automatisoituja toimintoja, joiden avulla vähennetään ihmisen tekemiä virheitä. Automatisoidut toimenpiteet takaavat luotettavat toistettavuuden. Tavoitteella (goal) päätetään, mikä prosessi suoritetaan. (Gedeon 2010, 30–31.)

Mavenin prosesseja:

- **Default:** vie projektin kääntö vaiheiden läpi
- **Clean:** siivoaa projektin eli poistaa väliaikaiset tiedostot ja luodon sisällön
- **Site:** käy läpi dokumentaation ja ilmoittaa vaiheet ja luo projekti puolen

Mavenin vaiheita:

- **Validate:** Vahvistaa, että projekti on hyvin määritelty ja kaikki vaadittu informaatio on saatavilla
- **Compile:** Kääntää projektin lähdekoodin
- **Test:** Testaa käännetyn koodin, käyttäen automatisoitua testi kehystä kuten JUnit.
- **Package:** Pakkaa koodin ja lähteet artefaktiin, joka on projektin lopputulos esim. JAR
- **Integration-test:** Sijoittaa paketin testiympäristöön integraatiotesteihin.
- **Verify:** Varmistaa, että paketti täyttää vaaditut laatu vaatimukset
- **Install:** Asentaa paketin paikallisesti, jotta muut projektit voivat käyttää sitä järjestelmässä
- **Deploy:** Asentaa paketin integraatioon tai vapauttaa sijainnin.

Kaikki vaiheet ovat riippuvaisia aiemmista vaiheista. Käyttämällä Mavenin tai kolmannen osapuolen liitännäisiä (plugins) elinkaaren rakennetta voi muokata haluamakseen. Liitännäisiä voidaan kytkeä tavoitteisiin tai laajentaa niitä tarjoamaan funktionaalisuutta rakenteiden tavoitteilla. Maven mahdollistaa helpon tavan saada halutut liitännäiset. Asennusvaiheessa saadaan minimaalinen määrä kirjastoja, jotka ovat vaadittu funktion toimintaan. Tarvittaessa kirjastoja voidaan ladata lisää verkosta luokittelu tietojen perusteella. Luokittelutietoja ovat liitännäisen nimi, ryhmä- ja artefaktitunniste. Mavenin tärkein informaation lähde on Project Object Model (POM) XML tiedosto. (Gedeon 2010, 32–33.)

POM-tiedosto sisältää tietoa mm. projektin ryhmä- ja artefaktitunnisteet sekä tarvittavat riippuvuudet. POM-tiedostossa määritellään myös projektin pakkaustyyppi, joka määrittelee tavoitteet (goals), jotka sidotaan elinkaaren vaiheisiin. Oletuksena pakkaustyyppi on JAR. Muita sisäänrakennettuja pakkaustyyppejä ovat WAR ja EAR. OSGi-bundlessa käytetään mukautettua pakkaustyyppi, joka on nimeltään bundle. Felix tarjoaa muutaman hyvän liitännäisen auttamaan rakennus- ja pakkausprosessia. Bundle Plugin avustaa bundlen pakkaamisessa. Junit4osgi Plugin integroi jUnit-testi kehyksen bundlen rakennusprosessiin. (Gedeon 2010, 32–33.)

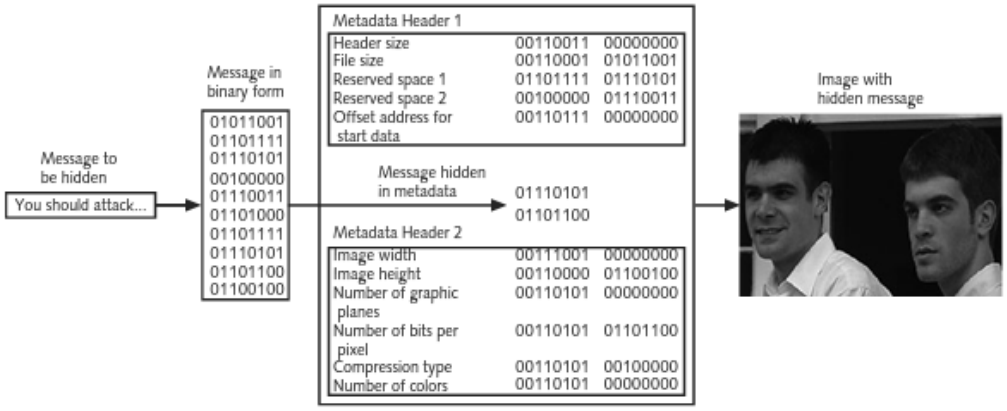
4 SALAUSALGORITMIT JA TARKISTUSSUMMA

Ensimmäisessä luvussa käsitellään salaamista yleisesti ja sen historiaa. Seuraavana käsitellään eri salausalgoritmeja. Viimeisenä käydään läpi, mikä on tarkistussumma ja mihin sitä käytetään.

4.1 Salaaminen

Tämä luku perustuu Ciampan (2008, 283–287) kirjoittamaan kirjaan salaamisesta. Tietojen suojaamisessa tärkeintä on niiden sekoittaminen. Sekoitetut tiedot ovat asiankuulumattomille hyödyttömiä. Tietojen sekoitus prosessia kutsutaan *kryptografiaksi*. Tiedot sekoitetaan käsittämättömään muotoon, kun niitä siirretään tai tallennetaan, jotta luvattomat käyttäjät eivät pääse käsiksi niihin.

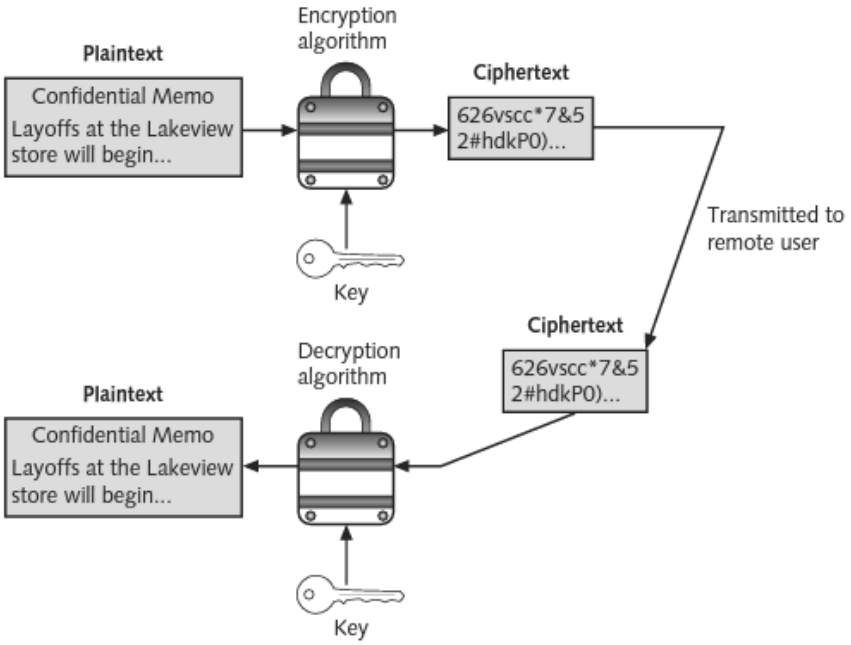
Kryptografiassa sekoitetaan viesti niin, että sitä ei voi katsoa ja *steganografia* piilottaa datan olemassaolon. Steganografia pilkkoo datan pieniin lohkoihin ja piilottaa tiedoston käyttämättömät osuudet. Data voidaan piilottaa tiedoston otsikkokenttiin, jotka kuvailevat tiedostoa metadata osioiden välissä. Steganografiaa voidaan käyttää kuva-, ääni- ja videotiedostoissa sisältämään piilotettua tietoa.



KUVA 4. Steganografialla piilotettua dataa (Ciampa 2008)

Käänteistä prosessia kutsutaan *salauksen puruksi*. Salaamattomassa muodossa olevaa dataa kutsutaan *suojaamattomaksi* (cleartext) dataksi. *Selväkielinen* (plaintext) data tarkoittaa suojaamatonta dataa, joka halutaan salata. Selväkielinen data on salausalgoritmin syöte. Salausalgoritmin prosessit koostuvat matemaattisista kaavoista, joita käytetään datan salaamisessa.

Avain on matemaattinen arvo, joka syötetään algoritmiin luomaan *salakirjoitusta*. Avainta voidaan verrata oikeaan avaimeen. Avain on asetettu lukkoon oven avaamiseen tai turvaamiseen. Kryptografiassa uniikki matemaattinen avain on syöte salausalgoritmin salakirjoittamiseen. Palauttaessa salakirjoitus selväkieliseksi käytetään salauksen purkualgoritmia.



KUVA 5. Salausprosessi (Ciampa 2008)

Perinteisin salausalgoritmi tyyppi on hajautusalgoritmi. Yleisimpiä hajautusalgoritmeja ovat Message Digest, Secure Hash Algorithm ja salasanatiivisteet. Hajautusprosessilla luodaan uniikki allekirjoitus datajoukolle. Allekirjoitusta kutsutaan *hajautukseksi* (hash) tai *tiivisteeksi* (digest).

Vaikka hajautusta pidetään salausalgoritmina, sen tarkoitus ei ole salakirjoittaa viestiä, jonka vastaanottaja purkaa myöhemmin. Hajautusta käytetään myös tiedon eheyden tarkistamiseen. Hajautuksella tarkistetaan, että tieto on alkuperäisessä muodossa eikä sisällä mitään haittaohjelmia.

Datajoukosta luotua hajautusta ei ole mahdollista purkaa. Esimerkiksi luku 12 345 kerrotaan luvulla 143, tulos on 1 765 335. Tulos 1 765 335 annetaan käyttäjälle ja käyttäjä pyytää määrittelemään alkuperäiset luvut. Alkuperäisten lukujen selvittäminen on mahdotonta, koska matemaattisia vaihtoehtoja on liikaa. Hajautus toimii samalla tavalla, koska siinä käytetään luotua arvoa eikä sitä ole mahdollista selvittää, vaikka alkuperäinen datajoukko tiedettäisiin.

Arkipäiväinen esimerkki hajautusalgoritmin käytöstä on pankkiautomaatti. Asiakkaalla on henkilökohtainen tunnistusnumero eli PIN-koodi. PIN-koodi on tallen-

nettu salattuna pankkikortin magneettijuovaan. Asiakkaan saapuessa pankkiautomaatille automaatti pyytää asettamaan pankkikortin lukijaan ja syöttämään PIN-koodin. Syötetty PIN-koodi salataan samalla algoritmilla, jolla se on tallennettu pankkikortin magneettijuovaan, jos nämä kaksi arvoa täsmäävät käyttäjä voi suorittaa haluamansa toimenpiteet pankkiautomaatilla.

4.2 Message Digest

Luvun sisältö perustuu Fisherin (2016) kirjoittamaan verkkoartikkeliin. Message Digest (MD5) on yksi tunnetuimmista hajautusalgoritmeista, jonka kehitti Ronald Rivest. Ensimmäinen Ronaldin kehittämä hajautusalgoritmi oli MD2. MD2 kehitettii vuonna 1989, joka oli rakennettu 8-bitisille tietokoneille. MD2-algoritmissa huomattiin vakavia tietoturvapuutteita, mutta se on silti käytössä vielä. MD2-algoritmia ei suositella käytettäväksi sovelluksissa, jotka vaativat korkean turvallisuustason. MD2 korvattiin vuonna MD4-algoritmilla vuonna 1990.

Ensimmäisenä 32-bittisille koneille kehitetty hajautusalgoritmi MD4 oli nopeampi kuin edeltäjänsä. MD5 julkaistiin vuonna 1992, joka oli tarkoitettu 32-bittisille koneille. Se ei ole niin nopea kuin MD4, mutta sitä pidetään turvallisempana kuin edeltäjänsä. MD5-algoritmissa on puutteita, joten se ei ole käytännöllinen sovellusten salaamisessa, mutta se sopii tiedostojen tarkistamiseen.

MD5 luo tarkistussumman siirrettävälle datajoukolle, jonka avulla sitä voidaan vertailla perillä, täsmääkö tarkistussumma alkuperäiseen. Tarkistussummien täsmätessä dataa ei ole peukaloitu. MD5-hajautukset ovat 128-bittiä pitkiä ja normaalisti ne esitetään 32 numeroa pitkinä heksadesimaaleina. Tämä toteutuu riippumatta siitä, kuinka suuri tai pieni tiedosto tai teksti on.

4.3 Secure Hash Algorithm

Luvun sisältö perustuu Fisherin (2015) kirjoittamaan verkkoartikkeliin. SHA-1 on yksi Secure Hash Algorithm (SHA) perheestä. Useimmat näistä kehitti National Security Agency (NSA) ja julkaisi National Institute of Standards Technology

(NIST). SHA-0 on 160-bittinen *viestitiiviste (message digest)* ja se oli ensimmäinen SHA-algoritmi. SHA-0 hajautus arvot ovat 40 numeroa pitkiä. Alun perin julkaistiin SHA nimellä vuonna 1993, mutta sitä ei ehditty käyttämään monessa sovelluksessa, koska se korvattiin versiolla SHA-1 vuonna 1995.

SHA-1 on myös 160-bittinen viestitiiviste, jota on pyritty vahvistamaan SHA-0 heikkouksien pohjalta. Kyseisessä salauksessa havaittiin puutteita vuonna 2006, jonka pohjalta NIST antoi lausunnon rohkaistakseen liittovaltion virastoja hyväksyvän SHA-2 käyttöönoton 2010 mennessä. SHA-2 on vahvempi kuin SHA-1 ja SHA-2 kohtaan kohdistuvat hyökkäykset ovat epätodennäköisiä nykyisillä laskentatehoilla.

Liittovaltion virastojen lisäksi yrityksiä kuten Google, Mozilla ja Microsoft ovat sanoneet, että he eivät hyväksy SHA-1 SSL-varmenteita vuoden 2017 jälkeen. SHA-2 julkaistiin vuonna 2001 monta vuotta SHA-1 julkaisun jälkeen. SHA-2 sisältää kuusi salaus funktiota vaihtelevilla tiiviste suuruuksilla: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, ja SHA-512/256. NIST julkaisi SHA-3 vuonna 2015, mutta sitä ei ollut kehittänyt NSA:n kehittäjät. SHA-3 ei luoto korvaamaan edeltäjäänsä kuten aiemmat versiot olivat.

SHA-1 käytetään nettisivustojen kirjautumisissa. Kirjautuessa sivustolle, jossa on käytössä SHA-1, salasanasi käännetään tarkistussummaksi. Tarkistussummaa verrataan sivustolle tallennettuun tarkistussummaan, jos ne täsmäävät saat pääsyn käyttäjätilille. SHA-1 käytetään myös tiedoston tarkistamisessa, jotkut nettisivustot tarjoavat SHA-1 tarkistussumman ladattuun tiedostoon.
Ladattuaan tiedoston käyttäjä voi tarkistaa tarkistussuman varmistaakseen, että ladattu tiedosto on sama kuin sivustolla.

4.4 Tarkistussumma

Tarkistussumma on käynnissä olevan algoritmin tulos, jota kutsutaan tiivistefunktioksi. Tarkistussummaa verrataan oman version tuottamasta tarkistussummasta tiedoston lähteen tuottamaan tarkistussummaan. Vertailu tehdään, että varmistutaan tiedoston aitoudesta sekä virheettömyydestä. Tarkistussummaa kutsutaan

myös nimillä tarkistesumma (hash sum), tarkistearvo (hash value), tarkistekoodi (hash code) sekä tarkiste (hash). Tarkistussumman käyttö on erittäin yleistä huoltopäivitysten (Service Pack) yhteydessä. (Fisher 2016)

Kuvitellaan iso päivitys grafiikka ohjelmaan. Päivitys on todennäköisesti iso tiedosto ja sen lataaminen vie useita minuutteja. Latauksen valmistuttua ei kuitenkaan voida olla varmoja tiedoston eheydestä. Tiedosto on voinut korruptoitua levylle kirjoittamisen aikana, silloin se ei ole täsmälleen sama kuin alkuperäinen. Korruptoitunut tai muuten puutteellinen tiedosto voi aiheuttaa paljon ongelmia, jos se asennetaan. (Fisher 2016)

Tarkistussumman avulla voidaan varmistaa, että tiedosto on omalla koneella sama kuin sijainnissa, josta se ladattiin. Internetissä on sivustoja, joista ladatessaan tiedoston saa mukaan tarkistussumman. Ladatun tiedoston tarkistussumman luomiselle on tarkistussummalaskin. Tarkistussummalaskimia on useita ja eri laskimissa on erilaiset tiivistefunktio valikoimat. Windows-käyttöjärjestelmällä toimivia tarkistussummalaskimia ovat mm. Microsoft File Checksum Integrity Verifier ja eXpress CheckSum Calculator (XCSC). Microsoftin ohjelma on komentorivi pohjainen tarkistussummalaskin ja XCSC-ohjelmassa on graafinen käyttöliittymä. JDigest on avoimeen lähdekoodiin perustuva tarkistussummalaskin, joka toimii Windows-, Mac- ja Linuxkäyttöjärjestelmillä. (Fisher 2016)

```
Tämä on testi.
tämä on testi.

Tämä on testi.
MD5:965a10228fb6db4e8fbac898fd8d9103

tämä on testi.
MD5:c51f0320e7447aa13035c148df561bc7
```

KUVA 6. Esimerkki tekstit ja MD5-tarkistussummat.

Kuvassa 6 olevassa teksteissä ylemmässä ei ole MD5-tarkastussummaa. Alemmassa osassa teksteihin on lisätty tarkistussummat. Ensimmäisessä osiossa ainoa ero on pieni alkukirjain toisessa tekstissä. Toisessa osiossa huomataan, että pienikin muutos tekstissä muuttaa tarkistussumman täysin erilaiseksi.

5 TOTEUTUS

Tässä luvussa käydään läpi järjestelmän toimintaa pintapuolisesti sekä kerrotaan, miten käyttöliittymän päivityspakettien jakaminen toteutettiin. Toteutuksesta kerrotaan paljon teknisestä näkökulmasta, mutta yritän tuoda myös käyttötapauksia esille. Toteutuksen läpikäynti alkaa mahdollisten toteutustapojen esittelyllä sekä kerron, päivityspaketin valmistusprosessista. Tämän jälkeen kerrotaan päivityksen lähettämisestä etäyhteydellä. Käyn läpi, miten päivitysprosessi käynnistyy socket –päätepisteeseen saapuneella viestillä. Seuraavana kerrotaan päivityspaketin lataamisesta ja miten sen eheys sekä oikeellisuus tarkistetaan tarkistussuman avulla.

Päivityksen asennus luvussa käsitellään päivityspaketin asennusprosessia. Viimeisenä esitellään, kuinka voidaan toteuttaa lokin kirjoitus Java-ohjelmointikielellä. Toteutus kehitettiin järjestelmässä olevan komponentin yhteyteen, jonka nimi on *käynnistäjä*.

Käynnistäjä on erillinen Java-ohjelma, joka käynnistää Apache Felix -ympäristön. Ympäristön ollessa käynnissä voidaan käynnistää järjestelmän eri komponentit eli OSGi-bundlet. Käynnistäjässä on graafinen käyttöliittymä, mistä järjestelmän käynnistämistä ja sammuttamista voidaan hallita. Järjestelmä voidaan myös käynnistää ilman käynnistäjää esim. Windows-käyttöjärjestelmän komentotulkista. Käynnistäjä on käytössä yleensä kaikissa tietokoneissa, jotka ovat käyttöliittymä roolissa.

5.1 Toteutustapoja ja esivalmistelut

Tutkiessani eri toteutustapoja törmäsin Apachen File Install -liitännäiseen. File Install -liitännäinen vaikutti aluksi hyvältä tavalta toteuttaa käyttöliittymän päivitysten jakelu. Liitännäisen avulla voidaan asentaa bundleja hakemistopohjaisesti. Hakemistopohjainen bundlejen asennus onnistuu luomalla kansio OSGi Felix asennushakemiston juureen. Felixin system properties -tiedostoon lisätään muuttuja *felix.fileinstall.dir* ja luotu kansion nimi. Esimerkiksi kansion nimen ollessa deploy tulisi kokonaisuuden näyttää *felix.fileinstall.dir=deploy*.

Felix lukee system properties -tiedoston käynnistyessään. Tässä vaiheessa deploy kansioon pudotetut bundlet asentuvat OSGi-kehykseen ja käynnistyvät välittömästi. File Install -liitännäisen toiminta perustuu hakemiston kyselyyn, se kyselee 2000ms välein, onko määritettyyn kansioon tullut uusia tiedostoja. Kysely aikaa voidaan muuttaa ja 2000ms on sen oletusarvo.

Kysely aikaa muutetaan samalla tavalla, kuin luotiin hakemisto, josta bundlet asennetaan. Lisätään muuttuja *felix.fileinstall.poll* ja aika millisekunteina. Esimerkiksi *felix.fileinstall.poll=5000*, jolloin kysely tapahtuu 5000 millisekunnin välein. Muuttujia on paljon erilaisia kuten *felix.fileinstall.bundles.startActivationPolicy*, joka määrittää käynnistetäänkö asennettava bundle heti. Aluksi pidin hyvänä vaihtoehtona käyttää File Install-liitännäistä bundlejen asentamisessa. Ongelmana kuitenkin oli, että se ei tarjoa vaadittavaa tietoturvaa liikenteenohjausjärjestelmälle. File Install asentaa kaiken määritetystä kansiosta mitä sinne pudotetaan.

Hylättyäni Apachen File Install -liitännäisen, tutkin muita mahdollisia toteutustapoja. Tarjolla oli paljon liitännäisiä suoraan Felixiin, mutta niissä oli omat rajoitteensa. Päädyin kehittämään oman ratkaisun, joka ei käytä Felix-liitännäisiä, koska sen avulla saavutetaan täsmälleen haluttu lopputulos. Samalla varmistetaan myös riittävä tietoturva järjestelmälle.

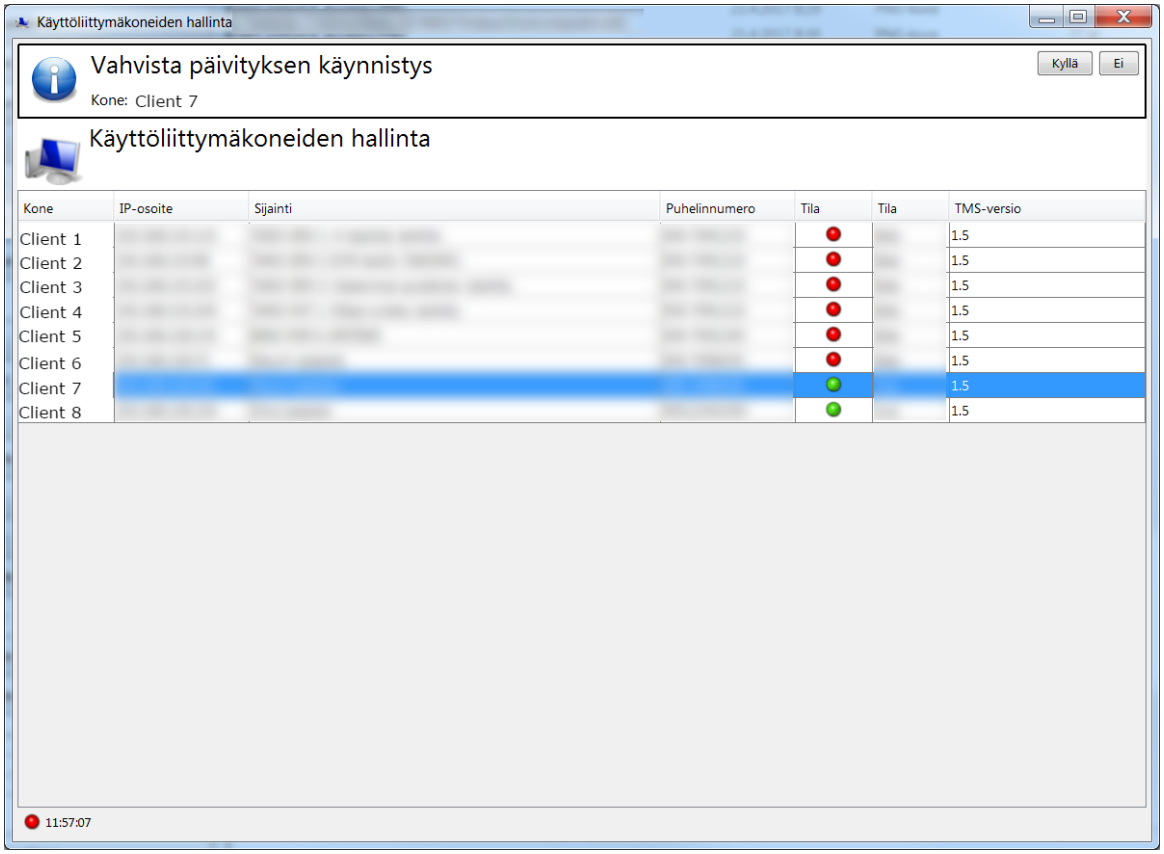
Kehitystyö alkoi tutustamalla järjestelmässä olevan komponentin toimintaan. Tutkin komponentin koodia ja huomasin, että se kuuntelee tiettyä socket –päätepistettä. Siihen voi siis lähettää viestin verkonvälityksellä. Päivitysten käynnistämisen lähettämällä viestin socket –päätepisteeseen. Komponenttia täytyi muokata, jotta se osaa käynnistää päivityksen tietyllä viestillä. Seuraavissa luvuissa kerron toteutuksesta tarkemmin.

5.2 Päivityspaketin valmistumisprosessi ja lähettäminen

Ennen päivittämisen aloittamista tarvitaan uusi versio järjestelmän komponenteista. Komponentteja kehitetään yleensä kahden viikon mittaisen sprintin aikana, jonka jälkeen suoritetaan testit komponenteille, joihin on tehty muutoksia. Komponenttien läpäistyä testit hyväksytysti rakennetaan päivityspaketti tarvittavista komponenteista. Mikäli komponentti ei läpäise testejä hyväksytysti siirretään se uudelleenkehitykseen. Päivityspaketti rakennetaan asentamalla komponentit (OSGi-bundlet) järjestelmään, jonka jälkeen järjestelmä sammutetaan ja kootaan asennushakemiston sisältö samaan pakettiin. Päivityspaketti siirretään haluttuun sijaintiin ja sille lasketaan SHA1-tarkistussumma ennakkoon. Valmis päivityspaketti on ZIP-muotoon pakattuna.

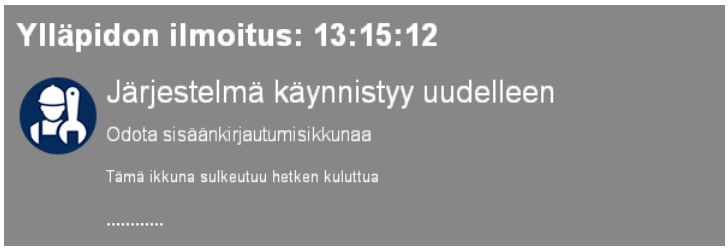
Päivityksen lähettämiseen kehitettiin kaksi mahdollisuutta, käyttäjälle tarjottava päivitys tai ns. pakotettu päivitys. Ensimmäisessä vaihtoehdossa käyttäjä saa ilmoituksen järjestelmään, että päivitys olisi tarjolla, haluatko päivittää. Käyttäjän hyväksyessä päivityspyyntö järjestelmän päivitys käynnistyy. Toinen vaihtoehto on ns. pakotettu päivitys, jonka ylläpitäjä käynnistää itse. Päivitys lähetetään molemmissa tapauksissa järjestelmästä ylläpitäjän toimesta.

Järjestelmässä on valmiina valikko (kuva 7), jossa on taulukko Client-koneista, joilla on etäyhteysjärjestelmään. Taulukko lukee listasta tarvittavat tiedot kuten koneen nimen, IP-osoitteen, sijainnin ja puhelinnumeron. Valikon kautta voidaan käynnistää uudelleen haluttu Client-kone sekä tarkastella sen OSGi-konsolia.



KUVA 7. Päivityksen lähetys

Kehitin samaan yhteyteen päivityksen lähettämisen valitulle Client-koneelle. Kuvailen seuraavaksi käyttötapauksen *pakotettu päivitys*. Päivitysprosessi alkaa ylläpitäjän kirjautumisella järjestelmään ja navigoi Käyttöliittymäkoneiden hallinta –valikkoon. Ylläpitäjä valitsee listasta Client-koneen, johon on yhteys ja lähettää päivityksen. Yhteyden tila on ilmaistu tila sarakkeessa olevalla värillisellä pallolla. Päivityksen lähettämisestä tulee vielä varmistusdialogi. (kuva 7).



Kuva 8. Järjestelmän uudelleenkäynnistys ilmoitus

Ylläpitäjän lähetettyä päivitys onnistuneesti käyttäjälle näytetään ilmoitus järjestelmän uudelleenkäynnistymisestä (kuva 8). Loppukäyttäjältä ei vaadita muita toimenpiteitä, kuin odottaa päivityksen asentumista ja kirjautua sisään asennuksen

valmistuttua. Mikäli päivitys aiheuttaa virhetilan, tulee käyttäjän olla välittömästi yhteydessä ylläpitäjään.

Kuvassa 9 on esimerkki komennon lähettämisestä tiettyyn socket –päätepisteeseen. SocketExample-luokka laajentaa abstraktin luokan Thread (säie), joka mahdollistaa säieturvallisen käytön. Thread-luokkaan kuulu run()-metodi, joka määrittelee mitä säie tekee käynnistyttyään.

Run-metodin sisällä kutsutaan setName(), joka asettaa säikeelle nimen. Socket-luokasta luodaan uusi olio, johon asetetaan parametreilla osoite sekä portti. Tämä jälkeen asetetaan socket –päätepisteen aikakatkaisu setSoTimetout(). Luodaan uusi olio luokasta BufferedReader, jonka parametreiksi laitetaan uusi olio InputStreamReader-luokasta sekä merkistökoodaus.

InputSreamReader olion parametrina kutsutaan getInputStream()-metodia, joka palauttaa syötevirran. BufferedWriter-luokalle tehdään sama sillä erotuksella, että InputStreamReader-luokan tilalle tulee OutputStreamWriter-luokka. OutputStreamWriter-luokan parametrina käytetään getInputStream-metodin sijaan getOutputStream()-metodia.

Otsikoiden (header) avulla voidaan erotella päivitys ja uudelleen käynnistys. Kirjoittamalla otsikoille if-lauseet mahdollistetaan oikean komennon lähetys. Lähetettävä komento kirjoitetaan BufferedWriter-luokan avulla, kun komento on kirjoitettu, kutsutaan flush()-metodia. Flush()-metodi ajaa ulos komennon eli lähettää sen. Oletuksena lähetetään komento EXAMPLE.

```
public class SocketExample extends Thread {
    private static final String LINE_CHANGE = "\r\n";
    private static final int serverPort = 12345;
    private static final String encoding = "ISO8859-1";
    private static final int timetout = 4231;
    private String serverAddr;
    private String header;
    private String command1Str = "Command_1";
    private String command2Str = "Command_2";
    private String command3Str = "Command_3";
    private  boolean success = false;
    private long eventId;

    public void run() {
        setName("The name of this thread");
        String line = "";
        Socket socket = null;
        BufferedReader in = null;
        BufferedWriter out = null;

        try {
            socket = new Socket(serverAddr, serverPort);
            socket.setSoTimeout(timetout);
            in = new BufferedReader(new InputStreamReader(socket.getInputStream(), encoding));
            out = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream(), encoding));
            if(getHeader().equals("Header1")) {
                out.write(command1Str+LINE_CHANGE);
                out.flush();
            }
            if(getHeader().equals("Header2")) {
                out.write(command2Str+LINE_CHANGE);
                out.flush();
            }
            if (getHeader().equals("Header3")) {
                out.write(command3Str+LINE_CHANGE);
                out.flush();
            }

            out.write("EXAMPLE"+LINE_CHANGE);
            out.flush();
```

KUVA 9. Esimerkki komennon lähettämisestä socket –päätepisteeseen

Uuden säikeen käyttöön ottaminen tapahtuu määrittelemällä start()-metodi SocketExample-luokasta luodulle oliolle. Säikeelle voidaan asettaa Thread-luokan sleep()-metodi, jonka aika säie lepää eikä syö resursseja. Sleep()-metodin aika asetetaan millisekunteina. Sleep()-metodin käyttö on erittäin suotavaa, jos säiettä käytetään toistorakenteissa esim. for-silmukassa. Säikeen käyttö toistorakenteessa ilman sleep()-metodia voi heikentää suorituskykyä. Säie sammutetaan kutsumalla interrupt()-metodi määritetylle oliolle.

5.3 Päivityksen käynnistyminen

Launcher kuuntelee tiettyä TCP-sockettia. Järjestelmän uudelleenkäynnistys etänä onnistuu tämän socket –päätepisteen kautta. Kuvassa 10 on esimerkki

siitä, kuinka komentoja voidaan vastaanottaa. Metodissa handleCommunicationExample() tuodaan parametrina Socket-luokasta luotu olio.

Luodaan uusi olio PrintWriter-luokasta ja asetetaan parametreiksi Socket-luokan metodikutsu getOutputStream(), sekä asetetaan boolean autoFlush arvoon true. Luodaan String-muuttuja inputLine, mutta ei aseteta sille mitään arvoa. Tehdään uusi olio BufferedReader-luokasta ja asetetaan parametreiksi uusi InputStreamReader. InputStreamReader-luokan parametrina on Socket-luokan getInputStream()-metodi. While-toistorakenteessa luetaan rivikerrallaan ja asetetaan String-muuttujaan inputLine kyseinen rivi. Inputlinen ollessa Command_3 asetetaan boolean-muuttujan example arvoksi true. Inputlinen ollessa Command_1 tai Command _2 asetetaan boolean -muuttujan start arvoksi true.

```
private void handleCommunicationExample(Socket socket) {
    PrintWriter out = null;
    try {
        out = new PrintWriter(socket.getOutputStream(), true);
        out.println("Something out printing...");
    } catch (IOException e) {
        e.printStackTrace();
    }
    String inputLine;
    try (BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()))) {
        while ((inputLine = in.readLine()) != null) {
            if (inputLine.equals("Command_3")) {
                example = true;
            }
            if(inputLine.equals("Command_1")) {
                System.out.println("Command_1 received");
                start = true;
            }
            if(inputLine.equals("Command_2")) {
                System.out.println("Command_2 received");
                start = true;
            }
            if (example) {
                if (inputLine.equalsIgnoreCase("EXAMPLE")) {
                    out.println("Restarting");
                    try {
                        RestartExamp restartExamp = new RestartExamp();
                        restartExamp.setVisible(true);
                        exampleLaunch.startupExample();
                    } catch (NullPointerException s) {
                        System.out.println("Error");
                    }
                    example = false;
                }
            }

            if (start) {
                if (inputLine.equalsIgnoreCase("EXAMPLE")) {
                    out.println("Update starting");
                    try {
                        RestartExamp restartExamp = new RestartExamp();
                        restartExamp.setVisible(true);
                        startUpdateProcess();
                    } catch (NullPointerException s) {
                        System.out.println("Error");
                    }
                    start = false;
                }
            }
            if (inputLine.equals("Command_0")) {
                example = false;
                start = false;
            }

        }
    }
```

KUVA 10. Esimerkki metodista jonka avulla vastaanotetaan komento

Examplen palauttaessa boolean arvon true tarkistetaan if-lauseella, että onko inputLine example. EqualsIgnoreCase ehdon palauttaessa arvo true luodaan uusi olio RestartExamp-luokasta. Kutsutaan JFrame-luokan setVisible()-metodia ja asetetaan sille boolean arvo true. Kutsutaan ExampleLaunch-luokan startupExample()-metodia. If-lauseen lopuksi asetetaan boolean-muuttuja example arvoksi false.

Startin palauttaessa true tarkistetaan if-lauseella, että onko inputLine example. EqualsIgnoreCase ehdon palauttaessa arvo true luodaan uusi olio RestartExamp-luokasta. Kutsutaan JFrame-luokan setVisible()-metodia ja asetetaan sille boolean arvo true. Kutsutaan startUpdateProcess()-metodia. If-lauseen lopuksi asetetaan boolean-muuttuja start arvoon false. Input linen ollessa Command_0 asetetaan boolean-muuttujat example ja start arvoon false. Finally-blokissa suljetaan PrintWriter sekä Socket.

5.4 Päivityspaketin lataaminen ja tarkistaminen

Päivityskomennon saapuessa tarkistetaan määritetty sijainti, onko sinne ilmestynyt uudempaa versiota järjestelmästä. Uudemman version löytyessä ladataan se koneelle, jonka jälkeen aloitetaan päivityspaketin tarkistus. Tarkistamisen avulla varmistutaan siitä, että päivityspaketin kaikki bitit ovat saapuneet ehjinä perille, eikä päivityspakettia ole peukaloitu matkan varrella. Tarkistaminen suoritetaan laskemalla SHA-1 tarkistussumma päivityspaketista ja verrataan sitä ennalta laskettuun tarkistussummaan. Ennalta laskettu tarkistussuma on muodostettu päivityspaketin luonnin yhteydessä.

Aluksi suunnittelin, että päivityspaketin tarkistaminen suoritettaisiin laskemalla ensin vanhan asennushakemiston tarkistussumma ja sitä verrattaisiin päivityspaketista saatuun tarkistussummaan. Tämä kuitenkin aiheutti useita ongelmia. Yksi ongelmista oli se, että uusi asennushakemisto voi olla tiedostokooltaan suurempi kuin vanha asennushakemisto, jolloin tarkistussummat eivät täsmää. Toinen ongelma oli, että päivityspaketti oli pakattu ZIP-muotoon, joten se on lähtökohtaisesti pienempi kuin asennushakemisto. Tarkistussumman laskemisessa voidaan käyttää eri algoritmeja mm. MD5 tai SHA-1. Laskeminen voidaan suorittaa myös

Javan CRC32, mutta se ei ole kovin vahva. Käytin SHA-1 algoritmia, koska se on vahvin näistä em. vaihtoehdoista.

Kuvassa 11 on esimerkki kuinka SHA-1 tarkistussumma voidaan laskea. Metodi createChecksum() vastaanottaa parametrin avulla tiedostonimen. Luodaan uusi olio FileInputStream-luokasta, jonka parametriksi laitetaan tiedostonimi. Luodaan byte[]-taulukko ja asetetaan sille pituudeksi 1024 tavua. Kutsutaan MessageDigest-luokan getInstance()-metodia ja laitetaan parametriksi algoritmi, joka halutaan.

```
public class ChecksumSHA1Example {

    private static final String fileToCheck = "C:\\Program Files (x86)\\Notepad++\\notepad++.exe";
    private static final String expectedChecksum = "e8e7a42647aee18127e2c5c595f9de5e59051068";

    public static byte[] createChecksum(String filename) throws Exception {
        InputStream fis = new FileInputStream(filename);
        byte[] buffer = new byte[1024];
        MessageDigest complete = MessageDigest.getInstance("SHA1");
        int numRead;
        do {
            numRead = fis.read(buffer);
            if (numRead > 0) {
                complete.update(buffer, 0, numRead);
            }
        } while (numRead != -1);
        fis.close();
        return complete.digest();
    }

    public static String getSHA1Checksum(String filename) throws Exception {

        byte[] b = createChecksum(filename);

        String result = "";
        for (int i = 0; i < b.length; i++) {
            result += Integer.toString((b[i] & 0xff) + 0x100, 16).substring(1);
        }
        return result;
    }

    public static void main(String args[]) {
        try {
            String installedVersionCksum = getSHA1Checksum(fileToCheck);

            if(installedVersionCksum.equals(expectedChecksum)) {
                //Do something
            } else {
                System.err.println("Checksum does not match");
                System.err.println("Expected checksum: "+expectedChecksum);
                System.err.println("Your checksum was: "+installedVersionCksum);
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

KUVA 11. Esimerkki SHA-1 tarkistussumman laskemisesta

Rakenne do-while pyörii niin kauan kuin numRead arvo ei ole -1. Asetetaan int muuttujaan numRead FileInputStream-luokan read()-metodi, jonka parametrina

on byte[]-taulukko. Numread arvon ollessa suurempi kuin 0 määrätään complete oliolle update()-metodi, joka päivittää tiivisteen käyttämällä määrättyä byte[]-taulukkoa. Parametreina on byte[]-taulukko eli puskurin pituus, byte[]-taulukon aloituspiste sekä byte[]-taulukon tavujen määrä. FileInputStream suljetaan kutsumalla close()-metodi. Kutsutaan complete-oliolle digest()-metodi ja palautetaan MessageDigest-objetki.

Metodi getSHA1Checksum() vastaanottaa parametrina tiedostonimen ja palauttaa String-muuttujan result. Luodaan byte[] taulukko ja kutsutaan createChecksume()-metodia, jonka parametrina on tiedostonimi. For-silmukassa käydään läpi niin kauan kuin int-arvo i on suurempi kuin MessageDigest-objektin pituus. Silmukassa muodostetaan tarkitussumma kaksi tavua kerrallaan. Main()-metodissa kutsutaan getSHA1Checksum()-metodia. Tarkistussumman täsmätessä odotettuun tarkistussummaan kutsutaan päivitys rajapintaa, jos tarkistussumma ei täsmää voi päivityspaketti olla korruptoitunut eli kaikki tavut eivät ole siirtyneet levylle. Toinen syy voi olla, että päivityspakettia on peukaloitu.

5.5 Päivityksen asennus

Päivityksen asennus alkaa siitä, että sammutetaan järjestelmä. Kopioidaan vanhasta asennushakemistosta config-tiedosto ja siirrettään se väliaikaiseen sijaintiin. Pakataan vanhan asennushakemiston sisältö ja siirretään määritettyyn varmuuskopiointi sijaintiin. Edellisiä versioita säilytetään aina siten, että kolme edellisintä on varmuuskopioituna.

Onnistuneen varmuuskopioinnin jälkeen aloitetaan päivityspaketin purkaminen asennushakemistoon. Poistetaan uuden asennuksen mukana tullut config-tiedosto ja kopioidaan alkuperäinen config-tiedosto väliaikaisesta sijainnista. Viimeisenä poistetaan väliaikainen config-tiedosto, sekä hakemisto joka sille luotiin. Config-tiedostoon on asetettu muuttujia, joita järjestelmä tarvitsee. Muuttujat ovat laitettu config-tiedostoon, jotta niitä voidaan muuttaa helposti ilman ohjelmointia.

Järjestelmän uuden version asennus perustuu hakemiston ja tiedostojen luomiseen, kopiointiin ja poistamiseen Java-ohjelmointikielen avulla. Varmuuskopiointiin kuului myös hakemiston pakkaaminen ZIP-muotoon (kuva 12). Metodi getAllFiles() listaa kaikki tiedostot. Se vastaanottaa parametrilla hakemiston File-luokasta tehdyn olion avulla ja listan tiedostoista.

Luodaan File[]-taulukko, johon asetetaan parametrilla tuotu kansio ja määrätään File-luokan listFiles()-metodi. Tämä metodi palauttaa kaikki kansion tiedostot ja ne laitetaan File[]-taulukkoon. For-silmukka käy läpi File[]-taulukon kokonaan ja lisää tiedoston kerrallaan listaan. If-lauseen avulla tarkistetaan, että onko tiedosto hakemisto. Tiedoston ollessa hakemisto kutsutaan getAllFiles()-metodia uudestaan.

```
public static void getAllFiles(File dir, List<File> fileList) {
    try {
        File[] files = dir.listFiles();
        for (File file : files) {
            fileList.add(file);
            if (file.isDirectory()) {
                System.out.println("directory:" + file.getCanonicalPath());
                getAllFiles(file, fileList);
            } else {
                System.out.println("     file:" + file.getCanonicalPath());
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void writeZipFile(File directoryToZip, List<File> fileList) {

    try {
        FileOutputStream fos = new FileOutputStream(savingLocation+File.separator+directoryToZip.getName() + ".zip");
        ZipOutputStream zos = new ZipOutputStream(fos);

        for (File file : fileList) {
            if (!file.isDirectory()) {
                addToZip(directoryToZip, file, zos);
            }
        }
        zos.close();
        fos.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void addToZip(File directoryToZip, File file, ZipOutputStream zos) throws FileNotFoundException, IOException {

    FileInputStream fis = new FileInputStream(file);
    String zipFilePath = file.getCanonicalPath().substring(directoryToZip.getCanonicalPath().length() + 1,file.getCanonicalPath().length());
    System.out.println("Writing '" + zipFilePath + "' to zip file");
    ZipEntry zipEntry = new ZipEntry(zipFilePath);
    zos.putNextEntry(zipEntry);

    byte[] bytes = new byte[1024];
    int length;
    while ((length = fis.read(bytes)) >= 0) {
        zos.write(bytes, 0, length);
    }
    zos.closeEntry();
    fis.close();
}
```

KUVA 12. Esimerkki hakemiston pakkaamisesta ZIP-muotoon

Metodi writeZipFile() kirjoittaa ZIP-tiedoston. Se vastaanottaa samat parametrit kuin getAllFiles()-metodi. Luodaan uusi olio FileOutputStream-luokasta ja asetetaan sille parametreina määritetty tallennushakemisto ja tiedostonimi. ZipOutputStream-luokasta luodaan myös uusi olio, jonka parametriksi laitetaan FileOutputStream-luokan olio. For-silmukassa kutsutaan addToZip()-metodia, jos tiedosto ei ole hakemisto. Lopuksi suljetaan ZipOutputStream ja FileOutputStream.

Metodi addToZip() lisää tiedostot ZIP-tiedostoon. Se vastaanottaa parametrina kansion joka halutaan pakata. Luodaan uusi olio FileInputStream-luokasta ja asetetaan sen parametriksi tiedosto. Luodaan uusi olio ZipEntry-luokasta ja asetetaan sen parametriksi String-muuttuja zipFilePath. Kutsutaan ZipOutputStream-luokan putNextEntry()-metodia, jonka parametrina on ZipEntry-luokan olio. Luodaan uusi byte[]-taulukko ja asetetaan pituudeksi 1024 tavua.

```
package fi.mipro.thesis.examples;

import java.io.File;

public class DeleteDirectoryExample {

    private static final String directoryToDeleteStr = "C:\\Users\\mattira\\Documents\\Example";

    public static void main(String[] args) {

        File directoryToDelete = new File(directoryToDeleteStr);
        try {
            FileUtils.deleteDirectory(directoryToDelete);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

}
```

KUVA 13. Hakemiston poistaminen FileUtils-luokalla.

Hakemiston poistamisessa käytin Apache Commons IO FileUtils-luokkaa. Kuvassa 13 on esimerkki kuin FileUtils-luokalla poistetaan hakemisto. Luodaan uusi olio File-luokasta ja asetetaan sille parametriksi poistettavahakemisto. Kutsutaan FileUtils-luokan deleteDirectory()-metodia ja laitetaan sen parametriksi File-luokasta luotu olio.

Kuvassa 14 on esitetty, kuinka pakattu tiedosto voidaan purkaa. Luodaan byte[]-taulukko ja asetetaan pituudeksi 1024 tavua. UnZipIt()-metodi vastaanottaa parametreille purettavan paketin nimen sekä kansion johon se puretaan. File-luokasta

luodaan uusi olio ja asetetaan parametriksi kansio, johon paketti halutaan purkaa. Tarkistetaan if-lauseella, että kansiota ei ole olemassa ja luodaan se File-luokan mkdir()-metodilla.

```
public class UnZip {
    List<String> fileList;
    private static final String INPUT_ZIP_FILE = "C:\\Example.zip";
    private static final String OUTPUT_FOLDER = "C:\\ExampleOutput";

    public static void main(String[] args) {
        UnZip unZip = new UnZip();
        unZip.unZipIt(INPUT_ZIP_FILE, OUTPUT_FOLDER);
    }

    public void unZipIt(String zipFile, String outputFolder) {
        byte[] buffer = new byte[1024];

        try {
            File folder = new File(OUTPUT_FOLDER);

            if (!folder.exists()) {
                folder.mkdir();
            }
            ZipInputStream zis = new ZipInputStream(new FileInputStream(zipFile));
            ZipEntry ze = zis.getNextEntry();
            while (ze != null) {
                String fileName = ze.getName();
                File newFile = new File(outputFolder + File.separator + fileName);
                System.out.println("file unzip : " + newFile.getAbsoluteFile());
                new File(newFile.getParent()).mkdirs();
                FileOutputStream fos = new FileOutputStream(newFile);
                int len;
                while ((len = zis.read(buffer)) > 0) {
                    fos.write(buffer, 0, len);
                }
                fos.close();
                ze = zis.getNextEntry();
            }
            zis.closeEntry();
            zis.close();
            System.out.println("Done");

        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

KUVA 14. Pakatun tiedoston purkaminen

Luodaan uusi olio ZipInputStream-luokasta, jonka parametriksi laitetaan uusi FileInputStream-luokka. FileInputStream-luokan parametrina purettavan paketin sijainti. Olioon on nyt asetettu ZIP-tiedoston sisältö. Kutsutaan ZipInputStream-luokan getNextEntry()-metodia, joka palauttaa list entry:n ja asetetaan ZipEntry muuttujaan. While-toistorakenne pyörii niin kauan, kuin list entry ei ole tyhjä. Asetetaan String-muuttujaan tiedoston nimi kutsumalla ZipEntry-luokan getName()-metodia. Luodaan File-luokasta uusi olio ja asetetaan parametriksi kansio, johon

paketti puretaan sekä tiedosto nimi. Em. parametrit muodostavat polun tiedostoille kun ne erotetaan File-luoka seprator avulla, eli se lisää ”/” hakemiston ja tiedostonimen väliin.

Luodaan uusi olio FileOutputStream-luokasta ja asetetaan parametriksi tiedostosta luotu olio. Kirjoitetaan while-toistorakenteesssa tiedostot paketista hakemistoon FileOutputStream-luokan write()-metodilla. Parametreina on byte[]-taulukko eli buffer, aloituspiste sekä pituus. Toistorakenteen lopussa suljetaan FileOutputStream ja asetetaan ZipEntry-oliolle ZipInputStream-luokan getNextEntry()-metodi. Suljetaan ZipInputStream-luokan entry kutsumalla closeEntry()-metodia. Viimeisenä suljetaan ZipInputStream close()-metodilla. Metodia unZipIt() kutsutaan main metodista ja asetetaan vaadittavat parametrit.

Kuvassa 15 on yksi tapa toteuttaa hakemistojen ja tiedostojen siirtely Java-ohjelmointikielellä. Metodi copyFolder() vastaanottaa parametreilla File-luokan olion src johon on asetettu lähde sekä dest eli kohde. If-lauseella tarkistetaan, onko lähde hakemisto. Seuraavaksi tarkistetaan if-lauseella, onko kohdehakemisto olemassa, jos ei ole, luodaan se File-luokan mkdir()-metodilla.

Listataan koko hakemiston sisältö kutsumalla File-luokan list()-metodia. Asetetaan lista String-muotoiseen files[]-taulukkoon. Käydään taulukko läpi for-silmukalla. Silmukassa selvitetään kaikki alihakemistot ja niiden tiedostot. Luodaan uudet oliot File-luokasta lähteelle ja kohteelle. Parametreina File-luokan olio src sekä String file.

Else-lause tapahtuu, jos lähde ei ole hakemisto vaan se on tiedosto. Luodaan uusi olio FileInputSream-luokasta ja asetetaan parametriksi File-luokan olio src. Tämän jälkeen luodaan uusi olio FileOutputStream-luokasta, jolle asetetaan parametriksi File-luokan olio dest. Tehdään uusi byte[]-taulukko ja asetetaan pituudeksi 1024 tavua. While-toistorakenteessa kirjoitetaan tiedostot uuteen sijaintiin käyttämällä FileOutputStream-luokan write()-metodia. Write()-metodin parametreina byte[]-taulukko buffer, aloituspiste sekä pituus. Lopuksi suljetaan FileInputStream sekä FileOutputStream.

```
    public static void copyFolder(File src, File dest) throws IOException {

        if (src.isDirectory()) {

            if (!dest.exists()) {
                dest.mkdir();
                System.out.println("Directory copied from " + src + "  to " + dest);
            }

            String files[] = src.list();

            for (String file : files) {
                File srcFile = new File(src, file);
                File destFile = new File(dest, file);
                copyFolder(srcFile, destFile);
            }

        } else {
            InputStream in = new FileInputStream(src);
            OutputStream out = new FileOutputStream(dest);
            byte[] buffer = new byte[1024];
            int length;
            while ((length = in.read(buffer)) > 0) {
                out.write(buffer, 0, length);
            }

            in.close();
            out.close();
            System.out.println("File copied from " + src + " to " + dest);
        }
    }
}
```

KUVA 15. Hakemistojen ja tiedostojen siirtäminen

Main()-metodissa asetetaan lähde- ja kohdehakemistot File-luokan olioihin. If-lauseella tarkistetaan, että lähdehakemisto on olemassa, jos sitä ei ole printataan ilmoitus ja kutsutaan System-luokan exit()-metodia. Else-lauseessa kutsutaan copyFolder()-metodia ja asetetaan sille luodut parametrit.

5.6 Lokin kirjoittaminen

Päivityksen asennuksessa voi tapahtua virheitä esim. päivityspaketti ei läpäise tarkistusprosessia, koska tarkistussuma ei täsmää. Virheen sattuessa on syytä kirjoittaa se lokiin. Järjestelmässä on olemassa oma lokipalvelu erityyppisille virheille. Päätin kuitenkin toteuttaa oman lokipalvelun, koska päivitysprosessi on oma osa, joten ei tule sotkea sen virheitä järjestelmän virheiden kanssa.

Internetistä löytyi paljon esimerkkejä, kuinka toteuttaa lokipalvelu Java-ohjelmointikielellä. Kuvassa 16 on yksi tapa toteuttaa lokipalvelu. Luodaan olio Logger-luokasta ja kutsutaan getLogger()-metodia, jonka parametrina kutsutaan class-luo-

kan getName()-metodia. Metodi palautta koko paketin nimen eli *fi.mi-pro.thesis.example.MyLogger*. FileHandler-luokasta luodaan olio ja asetetaan sille null.

```
public class MyLogger {
    public static void main(String[] args) {
        Logger logger = Logger.getLogger(MyLogger.class.getName());
        FileHandler fh = null;

        try {
            String logDirectoryStr = "C:\\MyLogs";
            String logFilePathStr = logDirectoryStr+File.separator+"MyExample.log";
            File logDirectory = new File(logDirectoryStr);

            if(logDirectory.exists()) {
                fh = new FileHandler(logFilePathStr);
            } else {
                logDirectory.mkdirs();
                fh = new FileHandler(logFilePathStr);
            }
            logger.addHandler(fh);
            SimpleFormatter formatter = new SimpleFormatter();
            fh.setFormatter(formatter);
            logger.setUseParentHandlers(false);


            logger.info("This is first option to write Level.INFO log.....");

            logger.log(Level.INFO, "And this is second option to write Level.INFO log");
            logger.log(Level.SEVERE, "This is severe");
            logger.log(Level.WARNING, "This is warning");
        } catch (SecurityException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        logger.info("This is second test.....");
        fh.close();
    }
}
```

KUVA 16. Esimerkki lokipalvelusta

Määritetään String-muuttujiin hakemistonimi sekä tiedostonimi. Luodaan uusi olio File-luokasta ja asetetaan parametriksi tiedostopolku. If-lauseen ehtona on File-luokan exists()-metodi, joka tarkistaa onko hakemisto olemassa. Metodi palauttaa boolean arvon true, jos hakemisto löytyy ja silloin luodaan uusi olio FileHandler-luokasta. Parametriksi laitetaan lokitiedoston polku. Metodin palauttaessa boolean arvon false luodaan kansio, koska sitä ei ole olemassa, jonka jälkeen tehdään samat toimenpiteet kuin if-lauseessa.

Kutsutaan Logger-luokan addHandler()-metodia ja asetetaan parametriksi FileHandler-luokan olio. SimpleFormatter-luokasta luodaan uusi olio. Kutsutaan FileHandler-luokan setFormatter()-metodia ja asetetaan parametriksi SimpleFormatter-luokan olio. Kutsutaan Logger-luokan setUseParentHandler()-metodia ja asetetaan sille boolean arvo false. Metodilla määritellään tulostetaanko lokit kehitysohjelma konsoliin lokin kirjoittamisen lisäksi.

Kutsutaan Logger-luokan info()-metodia ja asetetaan parametriksi String-arvona haluttu lokikirjoitus. Info()-metodilla kirjoitetaan yleensä jotain informatiivista lokiin, yleensä ei käytetä virheiden kirjoittamisessa vaan niille on severe()-metodi. Toinen tyyli lokin kirjoittamiseen on kutsua log()-metodia. Tällöin määritetään parametreilla lokiviestin taso (level) ja kirjoitettava viesti. Lokiviestin tasoja ovat mm. INFO, SEVERE ja WARNING.

Jokaisella tasolla on erilaiset käyttötarkoitukset. INFO-tasolla kirjoitetaan informatiiviset viestit esim. jonkun vaiheen läpäiseminen. SEVERE-tasoa käytetään virheviestin kirjoittamiseen esim. kun jotain tiedostoa ei löydy. WARNING-tasolla kirjoitetaan varoitukset esim. jotain konfiguraatiota ei löydetty ja joudutaan käyttämään sen oletusarvoa.

Tasot auttavat tietynlaisten viestien löytämisessä lokista, koska riville tulee asetettu etuliite esim. *INFO: This is a info*. Nämä ovat erittäin hyödyllisiä siinä vaiheessa, kun lokitiedostot ovat useita kymmeniä tuhansia rivejä. Henkilökohtaisesti tykkään käyttää log()-metodia. Viimeisenä suljetaan FileHandler close()-metodilla.

6 PÄÄTÄNTÖ

Opinnäytetyön lopputuloksena syntyi prototyyppi päivityksen jakelusta liikenteenohjausjärjestelmään Windows- ja Linux-käyttöjärjestelmälle. Toteutuksessa päivityksiä voi lähettää järjestelmän käyttöliittymästä ylläpitäjän oikeuksilla. Toteutus rakennettiin järjestelmän olemassa olevaan komponenttiin, jonka avulla hallitaan järjestelmän käynnistämistä ja sammuttamista. Päivitysprosessi käynnistyy, kun komponentti saa päivitysviestin TCP-protokollan välityksellä.

Opinnäytetyöni oli aiheena mielenkiintoinen ja haastava. Opinnäytetyö sisälsi ohjelmointia sekä ympäristöjen tutkimista. Kehitystyön aikana opin paljon uutta ohjelmointiin liittyen sekä sovelsin aiemmin opittuja tekniikoita. Tarkistussumma oli aiheena minulle aivan uusi. Opin myös paljon toimeksiantajani ympäristöistä sekä kehitettävästä järjestelmästä. Kehitystyössä haastavinta oli tehdä muutoksia olemassa olevan järjestelmän ohjelmaluokkiin.

Opinnäytetyöni jatkokehitystä ajatellen pitää kehittää päivitystenjakelu ratkaisu muille käyttöjärjestelmille sekä oma ratkaisu palvelintason komponenteille. Järjestelmän versionumerointia täytyy miettiä, kuinka sen voisi toteuttaa. Yksi mahdollisuus versionumerointiin olisi tekstitiedosto asennushakemistossa, joka luettaisiin järjestelmän käynnistyessä.

Tekstitiedostoon voisi myös lisätä tiedot, mitä päivityksen mukana on muuttunut, eli release notes -tiedot. Varmuuskopiointia pitää myös kehittää, koska tällä hetkellä järjestelmästä luodaan vain yksi varmuuskopio, joka ylikirjoitetaan aina uuden version asennuksen yhteydessä. Toimeksiantajani toive oli, että kolme edellisintä versiota pitäisi säilyttää.

Mielestäni varmuuskopioinnin kehittäminen vaatii ensin, että versionumerointi ratkaisu kehitetään, jotta varmuuskopioidut asennuspaketit voidaan nimetä versionumeron mukaan dynaamisesti. Opinnäytetyöni lopputulosten pohjalta on hyvä lähteä kehittämään em. asioita.

LÄHTEET

Atlassian Developers 2017. Lifecycle of a Bundle. WWW-dokumentti. Saatavissa: https://developer.atlassian.com/docs/atlassian-platform-common-components/plugin-framework/behind-the-scenes-in-the-plugin-framework/lifecycle-of-a-bundle [viitattu 4.5.2017.]

Ciampa, M. 2008. CompTIA Security+ 2008 In Dept. Boston: Cengage Learning PTR.

Fisher, T. 2016. What is MD5. WWW-dokumentti. Päivitetty 13.10.2016. Saatavissa: https://www.lifewire.com/what-is-md5-2625937 [viitattu 16.4.2017.]

Fisher, T. 2015. What is SHA-1. WWW-dokumentti. Päivitetty 30.10.2015. Saatavissa: https://www.lifewire.com/what-is-sha-1-2626011 [viitattu 16.4.2017.]

Fisher, T. 2016. What is a Checksum. WWW-dokumentti. Päivitetty 16.10.2016. Saatavissa: https://www.lifewire.com/what-does-checksum-mean-2625825 [viitattu 17.4.2017.]

Gedeon, W. 2010. OSGI and Apache Felix 3.0 Beginner's Guide. Birmingham: Packt Publishing.

Keskitetty tietokonepohjainen liikenteenohjausjärjestelmä. 2016. Mipro Oy. WWW-dokumentti. Saatavissa: http://www.mipro.fi/fi/business-lines/transportation/Liikenteenhallintaj%C3%A4rjestelm%C3%A4/ [viitattu 17.3.2017.]

McAffer, J., Vander, P. & Archer S. 2010. OSGi and Equinox: Creating Highly Modular Java Systems. Boston: Addison-Wesley Professional.

OSGi Alliance 2017. Architecture. WWW-dokumentti. Saatavissa: https://www.osgi.org/developer/architecture [viitattu 20.3.2017.]

OSGi Alliance 2017. Where to start. WWW-dokumentti. Saatavissa: https://www.osgi.org/developer/where-to-start/ [viitattu 19.3.2017.]