Tampereen ammattikorkeakoulu

Tiedonkeräyssovelluksen pohja

Simo Bergius

OPINNÄYTETYÖ
Kesäkuu 2020

Tieto- ja viestintätekniikka
Sulautetut järjestelmät ja elektroniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tieto- ja viestintätekniikka
Sulautetut järjestelmät ja elektroniikka

BERGIUS, SIMO:
Tiedonkeräyssovelluksen pohja

Opinnäytetyö 28 sivua
Kesäkuu 2020

Opinnäytetyössä tuotettiin malliesimerkki tiedonkeräyssovelluksesta tiedonkeräys-, tiedonsiirto-, tiedonvarastointi- ja -tiedonnäyttösovelluksia tuottavalle ohjelmistoyritykselle. Sovellus tehtiin C++-ohjelmointikielellä käyttäen Qt-ohjelmistokehystä. Sovelluksesta oli tarkoitus tehdä mahdollisimman yksinkertainen, joka liittää yrityksen omat pohjakomponentit toisiinsa. Sovellusta voitaisiin käyttää pohjana tuleville projekteille, jossa yleisimmät komponentit on otettu valmiiksi käyttöön ja jota on helppo lähteä jatkokehittämään asiakastarpeen mukaan.

Toissijaisena tarkoituksena tuotettiin Jenkins-alustalle käännös- ja testausautomaatio, johon on liitetty koodin staattista analyysiä, käännösvaroitusten keräämisen ja testien koodikattavuutta ja näiden asioiden näyttämistä. Jenkins-automaatio oli tarkoitus tehdä siten, että siitä voidaan ottaa mallia muuallakin. Tätä tehdessä opittiin C++-sovellusten automaattista integrointitestausta ja automaattista koostamista, jota tarvitaan laadun varmistamiseksi.

Testausautomaatio tehtiin Robot Framework -kielellä. Testitapauksia tehtiin testaamaan tiedonkeruusovellusta, joka käyttää pohjakirjastoja, joten epäsuorasti myös pohjakirjastoja testattiin.

Sovelluksesta tehtiin Docker-image, joka voidaan asentaa useanlaiseen ympäristöön, ja jota voidaan käyttää malliesimerkkinä Linux C++ -sovelluksen kontittamisesta.

Työn päätavoite onnistui niin kuin oli määritelty. Tiedonkeruusovelluksen toteutus onnistui juuri niin kuin oli ajateltu. Työssä onnistuttiin toteuttamaan sovellus, joka yhdistää valmiit komponentit toisiinsa yksinkertaisella tavalla. Jenkins-automaatio onnistui myös odotusten mukaisesti. Automaattiset testit tulivat toteutettua hieman suppeampina kuin alun perin oli tarkoitus.

Asiasanat: C++, Qt, Jenkins, Robot Framework, Docker, Testausautomaatio

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in ICT Engineering
Embedded systems and Electronics

BERGIUS, SIMO:
Base for a data collection application

Bachelor's thesis 28 pages
June 2020

The primary goal of this thesis was to create a base datalogging application for a company that creates data collection, transfer, storage, and display applications. The application is made with C++-programming language using Qt-framework. The application is built to be as simple as possible, to connect the base components of the company together. The application serves as a starting point for new projects, where the most common components are already taken into use and can easily developed according to customer needs.

The secondary goal of this thesis was to design build and test automation for Jenkins-environment, which contains static analysis of the code, collection of compiler warnings and test coverage report generation and making these things visible. Jenkins automation was designed to be used as a model in other projects. Experience in C++ automatic integration testing and automatic building were recorded, which is needed for quality assurance.

A Docker-image was made of the application, which could be deployed in different environments, and which could be used as a reference for generating an image from C++ applications.

The primary goal for this thesis was achieved as specified. Secondary goals were also achieved relatively well, and a lot was learned while implementing them. Implementation of the data collection application went as planned. Success was achieved in implementing an application that combines existing components in a simple way. Implementation of Jenkins automation also succeeded as expected. Implemented test automation was more brief than was originally intended.

Key words: C++, Qt, Jenkins, Robot Framework, Docker, test automation

SISÄLLYS

LYHENTEET JA TERMIT

Koostaminen	C++-sovelluksen kääntäminen lähdekoodista ja linkkaaminen kirjastoihin. Englanniksi build.
RSDK	Regatta client Software Development Kit, sisältää toimeksiantajan (Remionin) C++-kirjastoja sovelluskehitystä varten.
RDS	Regatta Device State, Regatta-laitteella oleva tietokanta, mikä sisältää sovelluksen historiatietoja, mm. versio ja konfiguraatio.
RMP	Regatta Messaging Protocol, Remionin luoma tiedonsiirtoprotokolla Regatta Coren ja Regatta-laitteen välistä kommunikointia varten.
Regatta Client	Kommunikaatiosovellus, joka kommunikoi eri palvelimien kanssa, muun muassa Regatta Corelle käyttäen RMP-protokollaa.
Regatta Core	Regatta-palvelinohjelmisto Regatta-laitteiden hallintaa, tarkkailua ja viestintää varten.
CAN	Controller Area Network, muun muassa autoissa ja työkoneissa käytetty väylä.
Docker	Virtualisointiympäristö, jossa voidaan ajaa sovelluksia kontteissa, jotka pystytetään Docker-imageja käyttäen. Docker-image sisältää sovelluksen tarvitsemat tiedostot.
Robot Framework	Automaattisten testien ja prosessiautomaation ohjelmistoviitekehys, jossa tehtävät pyritään kuvaamaan ihmisen helposti ymmärtämällä kielellä (Robot Framework 2020).

1 JOHDANTO

Datan kerääminen erilaisista kohteista, kuten työkoneesta rengaspaineen datan kerääminen, mahdollistaa uudenlaisen liiketoiminnan kehittämistä, kuten ennakoiva huolto, jossa sovellus ilmoittaa, kun renkaan paine laskee liikaa ja se tarvitsee täyttöä. Osa tätä kokonaisuutta on lähellä datan lähdettä oleva tiedonkeräyssovellus.

Toimeksiantajalla (Remion Oy) on useita asiakkaita, joiden tiedonkeräystarpeet ovat toisiinsa nähden osin samankaltaisia. Tätä varten toimeksiantajalla on laaja pohjakomponenttikirjasto, jonka avulla voidaan helposti tehdä sovellus datan keräämistä ja sen pilveen lähettämistä varten. Nykytilanteessa eri asiakkaiden sovellukset ovat jossain määrin toistensa kopioita, joihin on tehty asiakaskohtaisia muutoksia. Toisin sanoen ei ole kokonaista esimerkinomaista pohjasovellusta.

Tämän opinnäytetyön tarkoituksena on tuottaa toimeksiantajan tarpeisiin kokonainen tiedonkeräyssovelluksen pohja, jota voidaan käyttää uusien projektien alustana. Sovellus tehdään käyttäen toimeksiantajan olemassa olevia pohjakirjastoja (RSDK, Regatta client Software Development Kit) ja sovelluksia (mm. Regatta Client), jotka ovat valmiiksi monipuoliset, joten työssä on tarkoituksena liittää ne toisiinsa yleisellä ja yksinkertaisella tavalla. Pohjakirjastoihin kuuluu muiden muassa Measurement-moduuli, jota voidaan käyttää datan keräämiseen erilaisilla tavoilla ja erilaisista lähteistä, kuten CAN-väylältä. Valmiisiin sovelluksiin kuuluu muiden muassa Regatta Client, jota voidaan käyttää tiedostojen lähettämiseen eteenpäin, käyttäen erilaisia tietoliikenneprotokollia, kuten RMP (Regatta Messaging Protocol) tai MQTT (Message Queuing Telemetry Transport) valittuihin kohteisiin.

Nykypäivänä lähes mihin tahansa ohjelmistokehitysprojektiin on trendikästä liittää jatkuva kehitys ja jatkuva integraatio, ja niitä tukemaan automaattinen testaus. Tätä kokonaisuutta kutsutaan löyhästi DevOpsiksi. Siinä on ideana, että ohjelmistoa saadaan tehokkaasti kehitettyä, kun tehdään se pienissä palasissa, jotka ovat määritelty siten, että ne voidaan testauksen avulla todistaa toimiviksi

koko ajan. Lopuksi näiden todistetusti toimivien palasten liitos on myös toimiva. (Mäkelä 2019)

2 TARKOITUS JA TAVOITTEET

Tarkoituksena on luoda itsenäisesti toimiva C++-tiedonkeräyssovellus toimeksiantajan tarpeisiin. Sovelluksen tärkein tehtävä on kerätä dataa ja lähettää sitä palvelimelle. Sovellus käyttää toimeksiantajan valmiita, suhteellisen laajoja kirjastoja ja oheissovelluksia. Yksi tavoite on tehdä sovelluksesta sellainen, jota voi helposti muokata uusien vaatimusten mukaiseksi uusissa käyttökohteissa. Lisäksi yksi tavoitteista on luoda mahdollisimman yksinkertainen sovellus, jonka avulla voidaan testata toimeksiantajan kirjastojen toimintaa automaattisesti.

2.1 Nykyinen tilanne

Toimeksiantajalla on suhteellisen laajat, valmiiksi toteutetut C++-kirjastot, RSDK, ja muutama C++-sovellus, joita käytetään monessa eri asiakasprojektissa. Ne kuitenkin elävät koko ajan hieman, ja niihin tehdään jatkuvasti bugikorjauksia ja uusia ominaisuuksia. Myös uusia ulkoisia C++-kirjastoja lisätään silloin tällöin. Kirjastojen ja sovelluksien koostaminen tapahtuu Jenkins CI- jatkuva integraatio alustalla.

Eri kirjastoja ovat muun muassa Measurement (datan keräys eri lähteistä ja sen kirjoittaminen tiedostoon), RDS (Regatta Device State, sovelluksen tilan, version ja konfiguraatiotiedostojen historiatieto), Location (GPS paikannusfunktioita), GsmModem (GSM modeemin käyttö) ja monia muita tietyn asian tekemiseen liittyviä kirjastoja. Oleellisin tämän työn kannalta on Measurement, koska sen avulla dataa kerätään.

Valmiita C++-sovelluksia ovat mm. RegattaClient (Kommunikaatio Regatta Corelle ja muualle), ParameterMerger (etäpäivityksen yhteydessä tehtävä konfiguraatio- ja parametritiedostojen päivitys), RegattaStarter (hoitaa mm. sovelluksien käynnistyksen ja päivityksen). Tässä työssä käytetään RegattaClientiä, jonka avulla dataa lähetetään palvelimelle, ja RegattaStarteria, jonka avulla tiedonkeräyssovellus ja RegattaClient käynnistetään.

Asiakasprojektit ovat tehty käyttäen pääosin näitä valmiita kirjastoja ja sovelluksia sopivilla konfiguraatioilla. Useisiin projekteihin on lisätty lisäksi tarvittavia lisätoimintoja, jotka ovat koodattu projektisovellukseen. Pääsääntöisesti asiakasprojektit koostetaan Jenkinsissä omissa projekteissaan.

2.2 Nykyisen tilanteen ongelmat

Nykyisellään firmassa sulautettujen järjestelmien projekteissa ei ole laajasti otettu käyttöön koodin laaduntarkistusta, vaan käytössä on siellä täällä tehtyjä kokeiluja. Yksikkötestejä on jokaisessa projektissa jonkin verran, mutta integraatiotestejä on vähän. Myös staattista analyysiä ja testien koodikattavuutta on siellä täällä kokeiltu ja sille on tehty alustavaa tukea, mutta niitä ei kaikkialla ole otettu käyttöön.

2.3 Työn tavoitteet

Tavoitteena on luoda toimeksiantajalle itsenäinen tiedonkeräyssovellus käyttäen toimeksiantajan valmiita kirjastoja ja sovelluksia. Sovelluksen tulee ensisijaisesti olla mahdollisimman yksinkertainen, joka liittää valmiit kirjastot ja sovellukset toisiinsa geneerisellä tavalla. Muita tavoitteita on määritelty toimeksiantajan kanssa ja niitä ovat:

- Jenkins-projektin teko automaattista koostamista ja testausta varten,
- automaattiset testit,
- dockerointi

(Kick-off meeting notes 2019).

Kaikki osat pyritään tekemään siten, että niistä voi ottaa mallia uusiin ja olemassa oleviin toimeksiantajan asiakas- ja sisäisiin projekteihin, siltä osin, kun uusia asioita tehdään ja otetaan käyttöön. Muiden muassa Jenkinsissä tehtävä testausautomaation, koodin staattinen analyysin ja kääntäjän varoitusten tuominen näkyville on tällä hetkellä puutteellista ja sitä on tavoitteena tutkia ja edistää.

2.3.1 Sovellus

Sovellus kirjoitetaan C++-ohjelmointikielellä käyttäen Qt-ohjelmistokehystä. Toimeksiantajan pohjakirjastot ovat myös tehty C++:lla ja Qt:lla. Sovelluksen tehtävä on kertoa Measurement-moduulille, milloin datatiedostoja tulee luoda, ja siirtää tiedostot Outboxiin lähetettäväksi palvelimelle. Kokonaisuuteen kuuluu RegattaClient-sovellus, joka hoitaa tiedostojen lähetyksen Outboxista palvelimelle. Sovellukseen kuuluu myös RegattaStarter-sovellus, joka konfiguroidaan käynnistämään tiedonkeräyssovellus ja kommunikaatiosovellus, eli RegattaClient.

2.3.2 Jenkins-automaatio

Sovellukselle tehdään käännös- ja testausautomaatio toimeksiantajan Jenkins-palvelimelle. Jenkins-projektissa otetaan käyttöön koodin staattinen analyysi, sekä muita projektin ”terveyden” mittaamiseen käytettäviä metriikoita, kuten kääntäjän antamien varoitusten tuominen näkyville. Jenkinsissä myös ajetaan sovelluksen testit ja niiden tulokset tuodaan näkyville. Projektille tuodaan graafit edellä mainituista asioista. Jenkinsin toimintaan ja teoriaan paneudutaan luvussa 3.2.

2.3.3 Testausautomaatio

Sovellukselle tehdään soveltuvat testit, joiden avulla voidaan testata sovelluksen toimivuus. Testit ajetaan aina Jenkins käännöksen yhteydessä, ja niistä tuotetaan koodikattavuusraportti ja läpäisyraportti.

2.3.4 Docker

Sovelluksesta tehdään Docker-image, joka voidaan ottaa helposti käyttöön laitteissa, joissa on Docker-alusta. Docker-image luodaan aina Jenkins-ajon yhteydessä ja se työnnetään Remionin Docker-repositorioon. Dockerista lisää luvussa 3.4.

3 TYÖN TOTEUTUS

Ensimmäinen vaihe kaikessa modernissa ohjelmistokehityksessä on Git-repositorion tekeminen. Git on suomalaisen Linus Torvaldsin luoma versionhallintajärjestelmä, jonka avulla projektin koodit ja muut tiedostot voidaan jakaa muille, ja jonka avulla useampi tekijä voi tehdä lisäyksiä projektiin. Usein myös Jenkins-projekti käyttää käännöksen tekemiseen versionhallinnasta saatuja tiedostoja ja hakemistorakennetta. Git-repositorio tehtiin Remionin Bitbucket-palvelimelle. Sen nimeksi annettiin ”Example Datalogging Client Application”. (Git 2020).

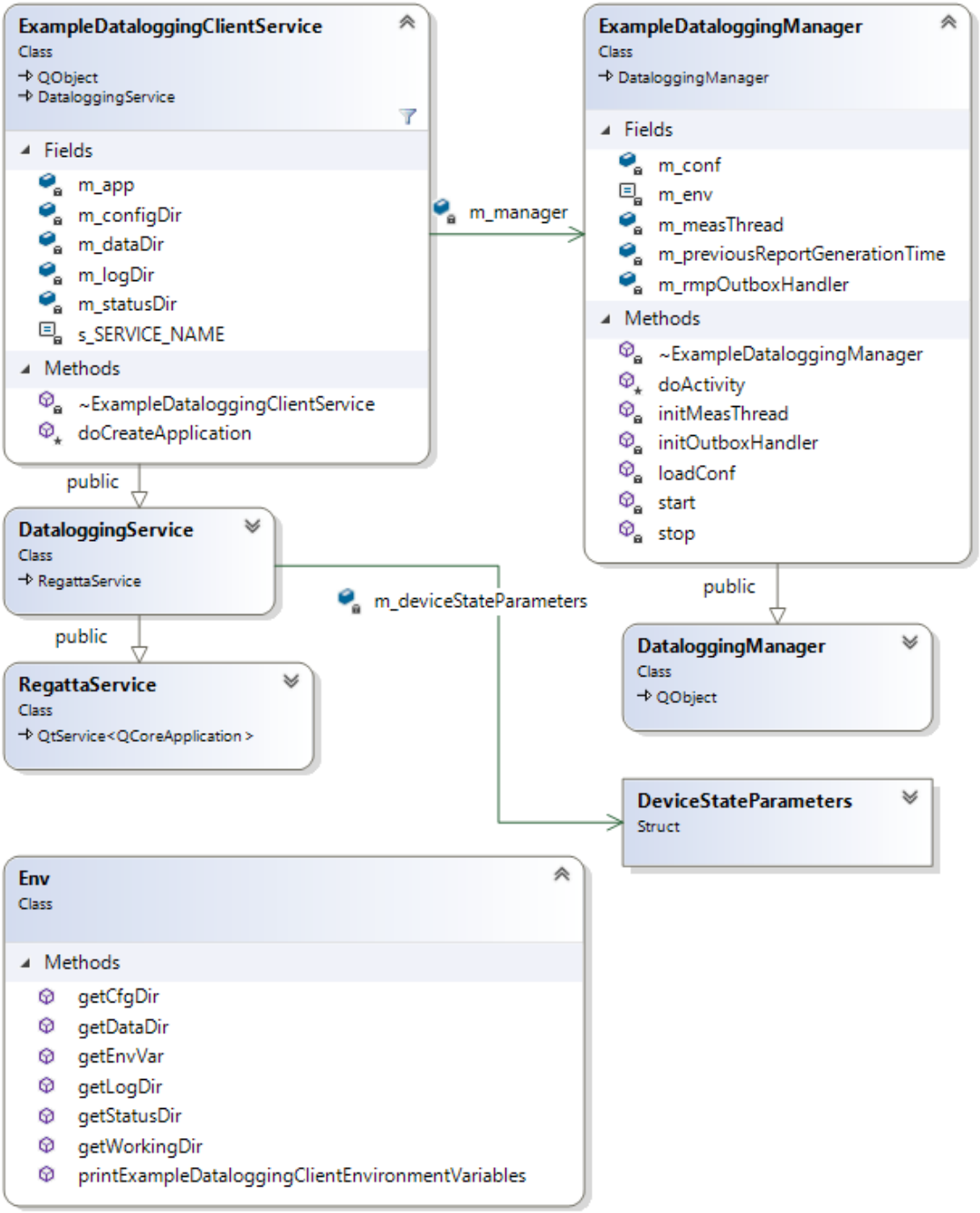
3.1 Tiedonkeräyssovellus

Sovellus tehtiin C++-ohjelmointikielellä. Koostamisessa käytetään CMake-koosto-ohjelmistoa. Sovelluksen lähdekoodit ovat hakemistossa Application/src (cpp-tiedostot) ja Application/include (otsikkotiedostot). Application-hakemistossa on lisäksi cfg-alihakemisto (oletuskonfiguraatiot), demodata (demo/testidatalle) ja CMakeLists.txt-tiedosto. CMakeLists.txt sisältää CMake-skriptin, joka kertoo CMakelle, mitä tässä hakemistossa tehdään. Application-hakemistossa oleva CMakeLists.txt kertoo sovelluksen koostamiseen liittyvät tiedot, eli mitkä ovat lähdekooditiedostot ja otsikkotiedostot ja mitä kirjastoja vastaan niitä linkitetään käännöksen yhteydessä. Siinä kerrotaan myös sovelluksen asennussijainti ja mitä kirjastoja asennussijaintiin asennetaan. CMakeLists.txt:ssä on käytetty paljon makroja ja muuttujia, jotka ovat määritelty RSDK:n koostotyökaluissa. Koostotyökalut ovat buildtools-nimisena moduulina (CMake 2020).

Sovellus koostuu main-funktiosta ja kolmesta luokasta, ExampleDataloggingClientService (palveluluokka), ExampleDataloggingManager (hallintaluokka) ja Env-luokka. Palveluluokka ja hallintaluokka ovat periytetty RSDK:n DataloggingAppCommonissa olevista kantaluokista. Palveluluokka on periytettyDataloggingAppCommonin DataloggingService-luokasta, joka puolestaan on periytetty RegattaService-luokasta, joka on periytetty Qt:n QtService-luokasta. Hallintaluokka on periytetty DataloggingAppCommonin DataloggingManager-luokasta,

joka on periytetty Qt:n QObject-luokasta. Env-luokka tehtiin lukemaan ympäristömuuttujia, joissa saattaa olla eri hakemistojen osoitteita ja jos ei ole, niin oletusarvo annetaan.

Hallintaluokka on toiminnan kannalta oleellisin luokka, koska siellä on ohjelman päätoimintasilmukka, jossa määritellään mitä ohjelma tekee määräajoin. Kuvssa 1 on näytetty sovellukseen tehdyt luokat jäsenmuuttujineen ja jäsenfunktioineen.



KUVIO 1. Luokkakaavio

Main-funktiossa alustetaan lokitus, sammutusfunktio ja parametrit palveluluokkaa (Service) varten. Alustuksien jälkeen käynnistetään palveluluokka. Lokituksen aloitukseen ja sammutusfunktion rekisteröintiin on DataloggingAppCommonissa valmiit apufunktiot. Palveluluokan tarvitsemat parametrit ovat:

- hakemistot konfiguraatioille, sovelluksen tilalle, datalle ja lokeille,
- RegattaDeviceState (RDS)-parametrit

Hakemistot haetaan Env-luokalta. RDS-parametreihin kuuluu jakelun (distribution) nimi ja versio, datankeräyssovelluksen nimi ja versio sekä datakeräyssovelluksen konfiguraatiotiedostot ja niiden tiedot.

Palveluluokan kantaluokka kutsuu aloittaessaan doCreateApplication-funktiota, joka on toteutettu periytetyssä luokassa. Funktiossa luokka alustaa parametrit hallintaluokalle ja käynnistää sen. Kantaluokassa lisätään RDS-tietokantaan sovelluksen tila, eli versionumerot ja konfiguraatiotiedostot.

Hallintaluokka lukee käynnistyessään sovelluksen konfiguraatiotiedoston, nimeltään datalogging_conf.xml. Konfiguraatiotiedostossa on kaksi parametria, jotka määrittävät, kuinka usein tiedostoja lähetetään ja pakataanko tiedostoja ennen lähetystä vai ei. Tämän jälkeen alustetaan Measurement-moduulin MeasThread lukemalla signals.xml-tiedosto, missä on määritetty signaalit, inputit ja outputit. MeasThread on luokka, joka tekee mittauksia eri inputeista ja kerää niiden tuloksia signaaleihin ja outputteihin. Tämä tapahtuu omassa säikeessään. Hallintaluokka tekee myös OutboxHandlerin, joka hallinnoi RegattaClientin outboxia, eli lähtevien tiedostojen hakemistoa.

Hallintaluokassa on doActivity-funktio, jota kutsutaan tietyin välein, mikä on määritelty konfiguraatiotiedostossa lähetysväliajaksi. Funktio ottaa MeasThreadista outputit ja käskee niitä tekemään uuden tiedoston. Tämän jälkeen tiedosto siirretään outboxiin, josta RegattaClient alkaa lähettämään sitä tiedostonimen perusteella etäpalvelimelle. Etäpalvelin on konfiguroitu RegattaClientin konfiguraatiotiedostossa.

RegattaClient konfiguroidaan regattaclient.xml-tiedostolla, jossa määritetään mm. gateway-lisäosia. Gatewayt ovat eri protokollalla tehtävää lähetystä ja vastaanottoa varten tehtyjä luokkia. Tällä hetkellä on olemassa MQTTGateway ja RMPGateway. Gatewaylla on inbox konfiguraatioita ja outbox konfiguraatioita. Jos haluttaisiin vastaanottaa dataa, esimerkiksi päivityspaketteja, ne voitaisiin vastaanottaa määrittämällä inboxeja. Outboxin avulla voidaan määrittää hakemisto, johon laitettavat tiedostot lähetetään gatewayn kautta palvelimelle.

RegattaStarter konfiguroidaan regattastarter.xml-tiedostolla. Sen tehtävänä on käynnistää konfiguraatiossa olevat sovellukset. RegattaStarter konfiguroitiin käynnistämään RegattaClientin ja tiedonkeruusovelluksen.

Sovellusta testattiin ajamalla sitä Windowsilla. Oletuskonfiguraatiossa signals.xml:ään laitettiin CANInput, joka konfiguroitiin lukemaan CAN-lokitiedostosta. Tarkistettiin, ilmestyykö outboxiin mittaustiedostoja ja niiden sisältöä tarkisteltiin, että niissä on oikeanlaisia signaaliarvoja vertaamalla niitä CAN-lokiin. Testattiin RegattaClientiä myös käynnistämällä se ja katsomalla Regatta Core-palvelimelta, että uusi laite rekisteröityy ja alkaa lähettämään RMP-viestejä. Varmistettiin myös, että sovellus luo RDS-tietokannan, jossa on tieto sovelluksen nimestä ja versiosta kuten kuuluukin.

3.2 Jenkins-automaatio

Jenkins on alusta, jolla voidaan toteuttaa automaattisesti koostamiseen, testaamiseen, toimittamiseen ja pystyttämiseen liittyviä tehtäviä. (Jenkins 2020).

Jenkins-projektin perusteena on projektin Git-repositorion juuressa oleva Jenkinsfile. Jenkinsfile-tiedostossa on putken (pipeline) määritys, eli mitä tehdään missäkin järjestyksessä ja millä suorittajalla (node). Pipelinessä määritellään vaiheita (stage), jotka voivat sisältää askelia (step), toisia vaiheita, rinnakkaisvaiheita (parallel) ja määrittelyn siitä millä suorittajalla vaihe suoritetaan (agent).

Toimeksiantajan Jenkins-ympäristössä on käytössä muutama fyysinen tietokone. Yksi näistä on nimeltään ”vs2012”, jolla voidaan tehdä Windows-käännöksiä, ja

”linux_docker”, joka on Linux-tietokone ja jolla voidaan tehdä käännöksiä käyttäen jotain Docker-imagea.

Toimeksiantajalla on valmiita Docker käännös-imageja eri alustoille kääntämistä varten. Yksi asiakkaan käyttämä kohdelaite on belgialaisen laitevalmistajan Optionin valmistama Linux-pohjainen Cloudgate (Option 2020). Cloudgate-käännöksien tekemiseen on docker-image, johon on asennettu Cloudgaten tarvitsema työkaluketju (toolchain). Tämän imagen nimi on ”cloudgate-docker-build-env”. Tämän imagen pohjana on käytetty työpöytä-linuxin käännös-imagea, mutta sinne on lisäksi asennettu Cloudgaten työkalut, eli kääntäjät, kirjastot ja otsikkotiedostot sekä muuta laitteeseen liittyviä tiedostoja.

Työpöytä linux käännöksien tekemiseen on docker-image ”rsdk-linux-docker-base-build-env” ja tätä pohjana käyttäen tehty ”rsdk-linux-docker-robotfw-build-env”, johon on lisätty RobotFramework-testausympäristö.

Projektin Jenkinsfilessä määriteltiin tekemään Cloudgate-käännös, Linux_x86-käännös ja Windows-käännös. Koska nämä käännökset ovat toisistaan riippumattomia, ne voidaan tehdä rinnakkain. Linux_x86 käännöstä käytetään docker-imagen luontiin ja testien suorittamiseen. Cloudgate- ja Windows-käännöksistä voisi tehdä asennuspaketin, jos haluttaisiin, jatkuvan pystyttämisen hengessä. Kuviossa 2 on esimerkki mikä toteuttaa tämän.

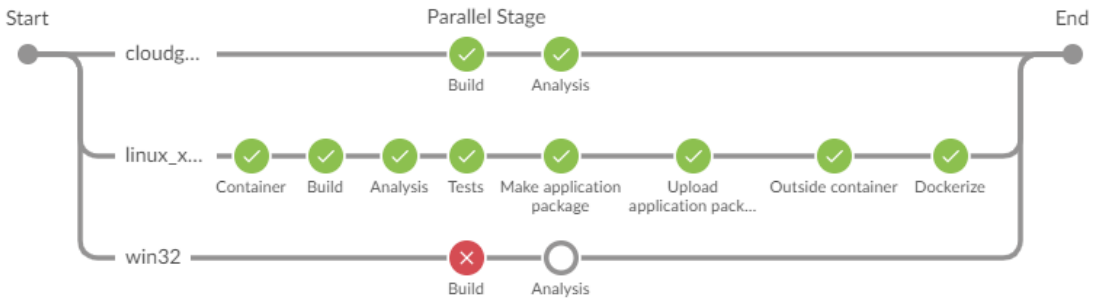
```
stages {
    stage('Parallel Stage'){
        parallel {
            stage('cloudgate') {
                agent {
                    docker {
                        image 'docker-private.remion.com/remion/cloudgate-docker-build-env'
                        args '-v /opt/RSDK:/opt/RSDK'
                        // /opt/RSDK contains installed RSDK libraries
                    }
                }
                stages {
                    stage('Build'){
                        // Build steps
                    } // cloudgate build

                    stage('Analysis') {
                        // Analysis steps
                    }
                }
            } // cloudgate
            stage('linux_x86') {
                stages {
                    // Linux build, testing, analysis, application package and docker image stages
                }
            } // linux_x86
            stage ('win32') {
                agent { node { label 'vs2012' } }
                stages {
                    stage ('Build') {
                        // Build steps
                    } // win32 build

                    stage('Analysis') {
                        // Analysis steps
                    } // win32 analysis
                }
            } // win32
        } // parallel
    } // Parallel Stage
}
```

KUVIO 2. Jenkins-pipelinen rakenne

Jenkins tarjoaa BlueOcean-lisäosan, josta voi hyvin visualisoida vaiheet. Kuvissa 3 on visualisoitu kaikki vaiheet.



KUVIO 3. Jenkins-pipeline visualisoitu Blue Oceanissa

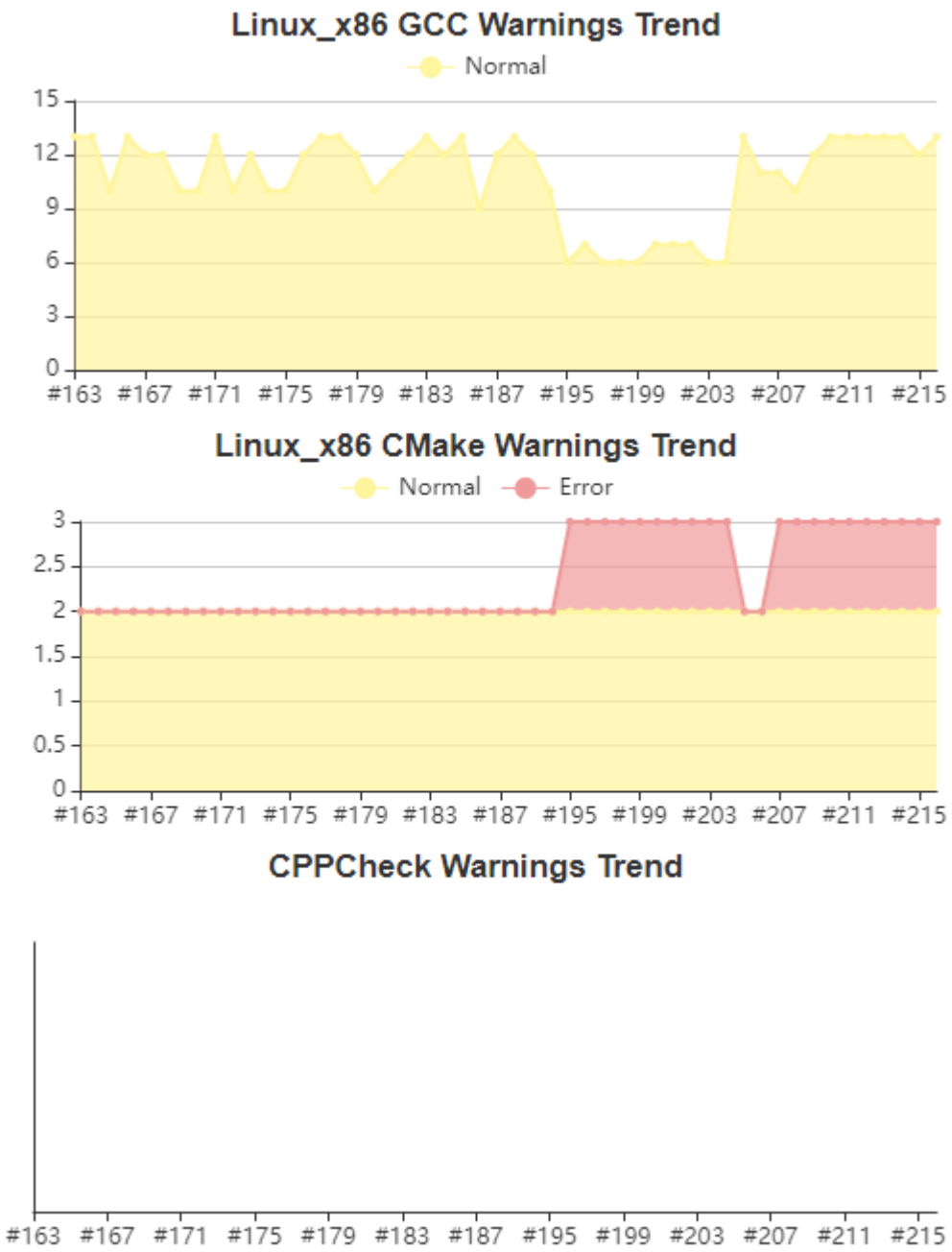
Kuvassa näkyy punaisella pohjalla rasti win32-haarassa Build-vaiheessa. Tämä tarkoittaa sitä, että Windows-käännös on epäonnistunut. Tässä tapauksessa se on epäonnistunut, koska pohjakirjastojen Windows-käännös ei ollut mennyt läpi.

3.2.1 Linux-käännös

Linux_x86-käännös tehdään docker-imagella, johon on lisätty RobotFramework-ohjelma. Käännöksen vaiheet ovat käännöksen tekeminen (build), koodin staattisen analyysin tekeminen, testien ajaminen, asennuspaketin luominen ja docker-imagen luonti.

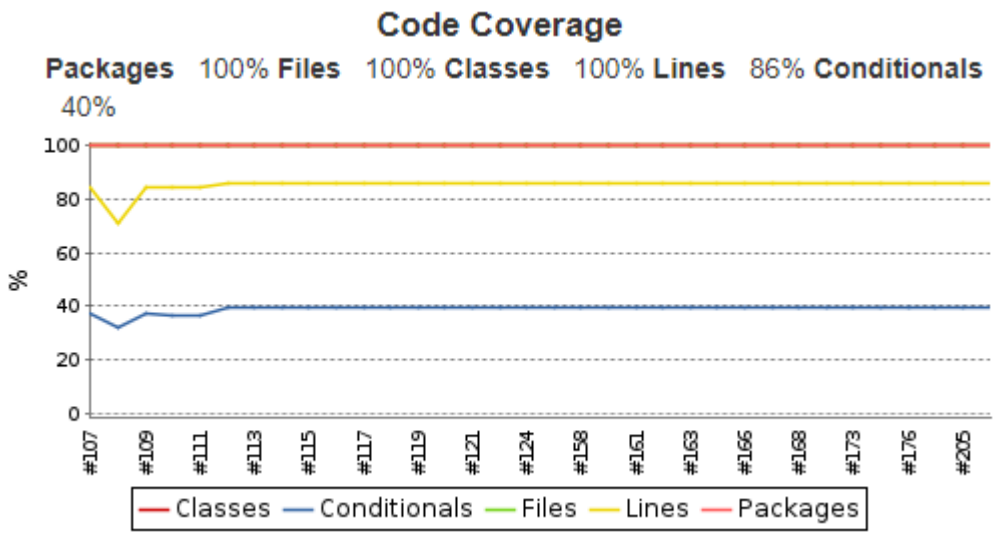
Käännöksen tekeminen tehdään kutsumalla koostoskriptiä sopivat ympäristömuuttujat asetettuina. Ympäristömuuttujissa määritellään RSDK-kohdelaitteen tyyppi, lokitustyyppi, kirjastotyyppi (jaettu vai staattinen) ja julkaisu- vai debug-käännös. Määriteltiin tehtävän sekä julkaisu- ja debug-käännös käyttäen log4cpp-lokitusta jaetuilla kirjastoilla kohdelaitteelle linux_x86. Log4cpp on C++-kirjasto, jonka avulla voidaan tuottaa lokirivejä, joissa on aikatieto, moduuli tai kooditiedosto- ja rivitieto sekä lokituksen vakavuuden taso (Log4cpp 2017).

Staattinen analyysi tehdään kutsumalla sitä varten tehtyä skriptiä, ja sen tulokset tuodaan näkyville kutsumalla recordIssues-funktiota sopivilla työkaluparametreilla. Käännöstyökaluista, eli CMake ja GCC-kääntäjä, otetaan varoitukset ja virheet. Skriptissä käytetään CPPCheck-työkalua, ja sen tuottamat varoitukset otetaan talteen recordIssues-funktiossa. CppCheck on C++-koodin staattinen analyysityökalu (Cppcheck 2020). Kuviossa 4 näkyy millaisen kuvaajan Jenkins tuottaa projektille. Työssä huomattiin, että pohjakirjastojen otsikkotiedostoissa olevat varoitukset tulevat myös tälle projektille. Ei tutkittu syytä, miksi käyrä heittelehtii paljon käännöksien välillä, vaikka muutoksia ei olisi.



KUVIO 4. Staattisen analyysien tulokset Jenkinsissä. X-akselilla on buildin numero ja Y-akselilla virheiden ja varoitusten määrä kustakin työkalusta

Testit ajetaan run-tests-skriptillä. Jos olisi tehty yksikkötestejä, ne ajettaisiin käyttäen valgrind-työkalua, joka koittaa huomata muistivuotoja ja muita virhetiloja ohjelman ajotilanteessa. Skripti ajaa myös RobotFrameworkilla tehdyt testit. Lopuksi käytetään gcovr-työkalua, joka tuottaa kattavuusraportin sen perusteella, mitä ohjelman ja testien ajon aikana on tapahtunut. Gcovr huomaa mitä koodirivejä on käyty läpi ajon aikana ja sen perusteella voidaan tehdä kattavuusraportti moduuleittain, luokittain, riveittäin ja ehdoittain (Gcovr 2019). Kuviossa 5 näytetään Jenkinsin näyttämä koodikattavuuskuvio.



KUVIO 5. Koodikattavuusraportti. X-akselilla on Jenkins-buildin numero

Työssä kokeiltiin, pystyykö Docker-kontissa tekemään muita docker-imageja. Sitä ei saatu toimimaan, joten docker-imagen tekeminen päädyttiin tekemään kontin ulkopuolella. Tätä varten asennuspaketti täytyy ottaa talteen ja tuoda kontin ulkopuolelle erikseen, jotta se voidaan asentaa imagen sisälle. Tämän vuoksi vaihe piti vielä jakaa kahteen eri suorittajilla tehtävään vaiheeseen, joista ensimmäinen on kontissa ja jälkimmäinen kontin ulkopuolella. Jenkinsfilessä tämä tehdään siten, että stage-osion sisällä on vielä ylimääräinen stages-osio, jossa ovat vaihe ”Container” kontissa tehtävälle vaiheelle ja ”Outside container” kontin ulkopuolella tehtävälle vaiheelle. Kuviossa 6 näkyy vaiherakenteen toteutus Jenkinsfilessä. Määritellään vaihe nimeltä ”linux_x86”, joka on yksi rinnakkaisvaiheista kuten kuvassa 2 näkyy. Tässä vaiheessa on alivaiheita, joista ensimmäinen tehdään docker-suorittajalla käyttäen imagea ”docker-private.remion.com/remion/rsdk-linux-docker-robotfw-build-env”, jolle on argumenteiksi annettu kaksi hakemistoa, mitkä liitetään konttiin sisälle isäntäjärjestelmästä. Hakemisto ”/opt/RSDK” sisältää RSDK:n kaikki käännetyt versiot. Käännösskriptit lukevat tiedostosta ”rsdk_branch.txt”, joka on Git-repositorion juuressa, RSDK:n version, jota vasten sovellus linkitetään. Kontin sisällä tehdään käännös (Build), staattinen analyysi (Analysis) testaus (Tests), asennuspaketin luonti (Make application package) ja asennuspaketin lataaminen Artifactory-paketinhallintajärjestelmään (Upload application package to Artifactory). Kontin ulkopuolella tehdään Docker-imagen generointi (Dockerize)-vaihe suorittajalla nimeltään ”linux-docker”, joka on linux-kone, jolle on asennettu Docker, itseasiassa sama kone jolla Jenkinsin

käyttämät docker-kontit pyörivät, mutta sillä ei ole merkitystä Jenkinsin toimivuuden kannalta.

```
stage('linux_x86') {
    stages {
        stage('Container') {
            agent {
                docker {
                    image 'docker-private.remion.com/remion/rsdk-linux-docker-robotfw-build-env'
                    args '-v /opt/RSDK:/opt/RSDK -v /mnt/build-output:/mnt/build-output'
                    // /opt/RSDK contains installed RSDK libraries
                }
            }
            stages {

                stage('Build') {…
                } // linux_x86 build
                stage('Analysis') {…
                } // linux_x86 analysis
                stage('Tests') {…
                } // linux_x86 Tests
                stage('Make application package') {…
                }
                stage('Upload application package to Artifactory') {…
                }
            } // linux_x86 docker stages
        } // Container
        stage('Outside container') {
            agent { node { label 'linux_docker' } }
            stages {
                stage('Dockerize') {…
                } // linux_x86 Dockerize
            } // linux_x86 cube stages
        } // Non-container
    }
} // linux_x86
```

KUVIO 6. Jenkins-pipeline Linux vaihe

3.2.2 Cloudgate-vaihe

Cloudgate-vaihe on muuten samanlainen, kuin linux_x86-käännös, mutta se tehdään docker-imagella ”docker-private.remion.com/remion/cloudgate-docker-build-env”, eikä siinä tehdä testejä, asennuspakettia tai docker-imagea. Tähän voisi lisätä vaiheita asennuspaketin tekemiseen ja lataamiseen, mutta sitä varten täytyisi tarkemmin suunnitella CloudGateen liittyviä muitakin asioita, kuten käynnistysskriptiä, CloudGaten firmware-pakettia ja asennusta. Kokonaisuudessaan sille ei nähty tarvetta, ja toteutettiin vain CloudGate-käännös sovelluksesta, jotta tulisi testattua kääntämistä useammalla alustalla.

3.2.3 Windows-vaihe

Windows-vaihe tehdään suorittajalla ”vs2012”, joka on Windows-kone, jolle on asennettu Visual Studio 2012 käännösympäristö. Kääntäminen tapahtuu generoimalla käännösskriptit, ja kutsumalla niitä. Vaiheessa tehdään myös staattinen analyysi, jossa otetaan kääntäjän varoitukset ja virheet talteen. Visual Studion kääntäjä saattaa tuottaa hieman eri varoituksia ja virheitä kuin linuxin gcc-kääntäjä, joten niistäkin saadaan tietoa koodin tasosta. Kuviossa 7 on vaiheen toteutus kokonaisuudessaan.

```
stage ('win32') {
    agent { node { label 'vs2012' } }
    stages {
        stage ('Build') {
            steps {
                bat 'generate-windows-buildscripts.bat'
                bat 'build_win32_debug_shared_log4cpp.bat'
                bat 'build_win32_release_shared_log4cpp.bat'
            }
        } // win32 build

        stage('Analysis') {
            steps{
                recordIssues aggregatingResults: false, tools:
                    [cmake(id:'Windows CMake',name:'Windows CMake'),
                    msBuild()]
            }
        } // win32 analysis
    }
} // win32
```

KUVIO 7. Windows-vaihe Jenkinsfile-tiedostossa

Windows-käännöksestäkin voitaisiin tehdä asennuspaketti, mutta sillekään ei nähty tarvetta.

3.3 Testausautomaatio

Sovelluksen testausautomaatioon käytetään RobotFramework-testausautomaatiotyökalua. RobotFramework on tarkoitettu testitapauksien esittämiseen ihmisen ymmärtämällä kielellä ja avainsanoilla, sen sijaan että pitäisi kryptistä koodia tulkita.

RobotFrameworkissa testit ovat jaettu testisarjoihin (test suite), jotka ovat omissa tiedostoissaan. Tiedostot voivat olla hakemistoissa, jolloin testisarja katsotaan kuuluvan toiseen testisarjaan, joka on hakemiston nimi. Tiedostoissa on testitapauksia, eli test case, osiossa Test cases. Siinä suoritetaan avainsanoja yksi kerrallaan, kunnes joku epäonnistuu tai aavainsanat loppuvat kesken. Jos yksikin avainsana epäonnistuu, testitapaus epäonnistuu. Jos mikään avainsana ei epäonnistu, testitapaus on silloin onnistunut.

Työssä toteutettiin tiedonkeräyssovellukselle käynnistystestejä ja tiedonkeräystestejä. RegattaClientille toteutettiin myös käynnistystestejä. Käynnistystesteissä testattiin, että sovellus käynnistyy ja sulkeutuu kun sille annetaan tiettyjä signaaleja. Myös sovellusten itsetestaus-ominaisuus testattiin. Kuviossa 8 on testitapauksen toteutus ja kuviossa 9 on testitapauksen loki, kun se on Jenkinsissä suoritettu.

```
Data Is Put Into Outbox From CAN Log
    # Check that outbox is empty
    Directory Should Be Empty    ${RMP_OUTBOX_DIR}
    Directory Should Be Empty    ${INTERMEDIATE_JSON_DIR}
    Directory Should Be Empty    ${INTERMEDIATE_RXBS_DIR}
    Start Application
    Sleep    15s
    # Check that file(s) has appeared in outbox
    Directory Should Not Be Empty    ${RMP_OUTBOX_DIR}
    # check that first file is not empty
    ${files}  =  List Files In Directory    ${RMP_OUTBOX_DIR}
    ${file_size}  =  Get File Size    ${RMP_OUTBOX_DIR}/${files}[0]
    Should Be True  ${file_size} > 0
```

KUVIO 8. Robot Framework-testitapaus

TEST Data Is Put Into Outbox From CAN Log

Full Name:	Suites.Datalogging.Datalogging.Datalogging.Data Is Put Into Outbox From CAN Log
Start / End / Elapsed:	20200229 05:02:01.322 / 20200229 05:02:16.545 / 00:00:15.223
Status:	PASS (critical)

SETUP Test Case Setup
KEYWORD OperatingSystem.Directory Should Be Empty ${RMP_OUTBOX_DIR}
KEYWORD OperatingSystem.Directory Should Be Empty ${INTERMEDIATE_JSON_DIR}
KEYWORD OperatingSystem.Directory Should Be Empty ${INTERMEDIATE_RXBS_DIR}
KEYWORD Start Application
KEYWORD BuiltIn.Sleep 15s
KEYWORD OperatingSystem.Directory Should Not Be Empty ${RMP_OUTBOX_DIR}
KEYWORD ${files} = OperatingSystem.List Files In Directory ${RMP_OUTBOX_DIR}
KEYWORD ${file_size} = OperatingSystem.Get File Size ${RMP_OUTBOX_DIR}/${files}[0]
KEYWORD BuiltIn.Should Be True ${file_size} > 0
TEARDOWN Test Case Teardown

KUVIO 9. Robot Framework testitapauksen loki Jenkinsissä suorituksen jälkeen

RegattaClientin itsetestaus-ominaisuuteen oli työn aikana tullut bugi, joka aiheutti sen, että etäpäivitys ei onnistunut, koska etäpäivityksen yhteydessä tehdään päivitetylle ohjelmistolla itsetesti ja jos se epäonnistuu, niin päivitys perutaan ja otetaan vanha ohjelmisto takaisin käyttöön. Tämän vuoksi RegattaClientille tehtiin itsetestaus-testi.

Tiedonkeräystestissä katsottiin, että sovellus laittaa outbox-kansioon tiedostoja, kun sille annetaan konfiguraatio, joka käynnistyessään tuottaa dataa. Testissä käytettiin CAN-väylän lukemista, mutta sellaisella CAN-väylän konfiguraatiolla, jossa väylänä toimii lokitiedostosta lukeminen, koska suorittajalla ei ole oikeaa CAN-väylää. Testiä voisi parantaa vielä siten, että verifioitaisiin, että tieto on oikeanlaista. Ajankäytöllisistä syistä niin monimutkaista testiä ei toteutettu.

3.4 Docker

Docker on virtualisaation mahdollistava teknologia, jossa sovelluksista tehdään imageja, eli levykuvia, ja niitä ajetaan konteissa. Kontit ovat eristetty toisistaan samaan tapaan kuin fyysiset tai virtuaaliset koneet olisivat toisistaan. Kontteja voidaan kuitenkin parametroida käyttämään jotain isäntäjärjestelmän hakemistoja liittämällä niitä kontin luontivaiheessa. Myös verkkoportteja voidaan ohjata konttiin. (Docker 2020).

Docker-imaget ovat paikallisessa rekisterissä ja niitä voi työntää ja noutaa etärepositoriosta.

Docker-imageja luodaan niin sanotuilla Dockerfileillä, jotka ovat lista käskyjä, jotka ajetaan kontin sisällä ja tallennetaan lopputulos uudeksi imageksi. Dockerfilessä aluksi otetaan yleensä pohjalle joku toinen Docker-image.

Tässä työssä Docker-image tehdään viimeisenä osana Jenkins pipelineä. Pohjana käytetään kevyttä Alpine Linux distribuutiota sen keveyden vuoksi. Kuitenkin sovelluksen koostamisvaiheessa käytetään Ubuntun imagea, koska sovellus on linkitetty Ubuntussa käytettyjä kirjastoja vasten. Tämän vuoksi aluksi Dockerfilessä on otettu Ubuntun image, josta kopioidaan kirjastoja, joita vasten sovellus on linkitetty. Imageen laitetaan käynnistysskripti ja RegattaStarterin konfiguraatiotiedosto. Käynnistysskriptissä asetetaan ympäristömuuttujat sovelluksien toimintaa varten. Koska sovellus on dynaamisesti linkitetty jaettuja kirjastoja vasten, kirjastot täytyy löytyä jostain LD_LIBRARY_PATH -ympäristömuuttujan osoittamista hakemistoista. Myös Qt:n pluginit ladataan QT_PLUGIN_PATH -ympäristömuuttujan osoittamista hakemistoista. RegattaStarterin konfiguraatiotiedosto laitetaan imageen sisälle, jotta se olisi aina mukana samanlaisena. Siinä kerrotaan RegattaStarterille, mitkä sovellukset sen tulee käynnistää. Koska sovellukset ovat aina RegattaClient ja tiedonkeruusovellus, voi RegattaStarterin konfiguraatio olla staattisena imagen sisällä. Kuviossa 10 näkyy Dockerfilen toteutus.

```
FROM library/ubuntu:16.04 AS ubuntu

RUN apt-get update -y && apt-get -y install vim iputils-ping dbus libproxy-dev libicu55 libpcre16-3 \
    libglib2.0-0 libsqlite3-0 libqt5sql5 && rm -rf /var/lib/apt/lists/*

FROM alpine:latest

LABEL maintainer="Remion Oy <support@remion.com>"
# Copy dependencies from ubuntu
COPY --from=ubuntu /lib64/ld-linux-x86-64.so.2 /lib64/ld-linux-x86-64.so.2
COPY --from=ubuntu /lib/x86_64-linux-gnu/libgcc_s.so.1 \
    /lib/x86_64-linux-gnu/libc.so.6 \
    /lib/x86_64-linux-gnu/libm.so.6 \
    /lib/x86_64-linux-gnu/libpthread.so.0 \
    /lib/x86_64-linux-gnu/librt.so.1 \
    /lib/x86_64-linux-gnu/libdl.so.2 \
    /lib/x86_64-linux-gnu/libz.so.1 \
    /lib/x86_64-linux-gnu/libglib-2.0.so.0 \
    /lib/x86_64-linux-gnu/libcrypto.so.1.0.0 \
    /lib/x86_64-linux-gnu/libssl.so.1.0.0 \
    /lib/x86_64-linux-gnu/libpcre.so.3 \
    /lib/x86_64-linux-gnu/libnss_dns.so.2 \
    /lib/x86_64-linux-gnu/libnss_files.so.2 \
    /lib/x86_64-linux-gnu/libresolv.so.2 \
    /usr/lib/x86_64-linux-gnu/libsqlite3.so.0.8.6 \
    /usr/lib/x86_64-linux-gnu/libsqlite3.so.0 \
    /usr/lib/x86_64-linux-gnu/libstdc++.so.6 \
    /usr/lib/x86_64-linux-gnu/libicui18n.so.55 \
    /usr/lib/x86_64-linux-gnu/libicuuc.so.55 \
    /usr/lib/x86_64-linux-gnu/libpcre16.so.3 \
    /usr/lib/x86_64-linux-gnu/libproxy.so.1 \
    /usr/lib/x86_64-linux-gnu/libicudata.so.55 /usr/local/data-logging-service/lib/x86_64-linux-gnu/

ADD build/example-datalogging-service.tar.gz /usr/local/data-logging-service

WORKDIR /usr/local/data-logging-service/bin
CMD [ "./RegattaStarter.sh" ]
```

KUVIO 10. Dockerfilen toteutus

Docker-imagea ajettaessa on tarkoitus liittää isäntäjärjestelmän tiedostojärjestelmässä oleva konfiguraatiohakemisto imageen sisälle, ja tätä kautta konfiguroida tiedonkeräys- ja kommunikointisovellus.

Imagea testattiin lataamalla se Linux-pöytäkoneelle repositoriosta, jolla on Docker toimintakunnossa, ja ajamalla sitä sopivalla konfiguraatiolla.

4 POHDINTA

Työn päätavoite onnistui niin kuin oli määritelty. Muut tavoitteet onnistuivat myös suhteellisen hyvin ja niitä tehdessä opittiin uusia asioita. Mielenkiintosimmaksi tekemiseksi osoittautui yllättävästi Jenkinsin käyttöönotto ja sinne staattisen analyysin ja testausautomaation tekeminen.

Tiedonkeruusovelluksen toteutus onnistui juuri niin kuin oli ajateltu. Onnistuttiin tekemään sovellus, joka yhdistää valmiit komponentit toisiinsa yksinkertaisella tavalla, mikä kuitenkin valmiiden komponenttien laajuuden takia, on monipuolisesti konfiguroitavissa. Pohjakomponenteista otettiin käyttöön Measurement-moduuli, DataloggingAppCommon-kirjasto, RegattaDeviceState (RDS)-kirjasto ja RegattaClient.

Jenkins-automaatio onnistui myös odotusten mukaisesti. Sitä tehdessä opittiin enemmän Jenkinsin käyttöä. Jenkinsiin tuli tehtyä sovelluksen koostaminen, koodin staattinen analyysi, testausautomaatio ja docker-imagen tuottaminen. Koodin staattinen analyysi on varmasti sellainen mitä muissakin projekteissa on hyvä käyttää, joten tästä voi katsoa mallia sen tekemiseen.

Automaattiset testit tulivat tehtyä hieman suppeampia kuin alun perin oli tarkoitus. Tiedonkeruusovelluksesta tuli tehtyä käynnistystestit ja tiedonkeruutestit. Alun perin oli tarkoituksena tehdä myös RegattaClientille enemmän testejä, joissa olisi esimerkiksi testattu tiedostojen lähettämistä palvelimelle. Ajankäytöllisistä syistä RegattaClientille tuli tehtyä vain yksi, minkä aiempi puute aiheutti työn teon aikana toisissa projekteissa ongelman, joka olisi tämänkaltaisen testitapauksen avulla voitu huomata. Testauksessa yleisesti on myös sellainen dilemma, että testitapauksia ei ikinä voi olla liikaa, mikä johtaa jatkuvaan tunteeseen siitä, että ne ovat puutteellisia. Toisin sanoen erikoistapauksia on missä vaan ohjelmassa niin paljon, että olisi epäkäytännöllistä testata niitä kaikkia. Kuitenkin jokaiseen erikoistapaukseen liittyy mahdollinen bugi, joka voitaisiin testaamalla huomata ja korjata. Jossain vaiheessa on tehtävä päätös, että nyt on tarpeeksi hyvin testattu.

Docker-imagen tekeminen onnistui myös. Siinä tuli myös uudenlainen tilanne liittyen sovelluspaketin ajoympäristöön, eli konfiguraatiotiedostojen sijainti. Huomattiin, että kun imageen sisältyy aloitussovellus, tässä tapauksessa RegattaStarter, niin tämän aloitussovelluksen konfiguraatio on oltava staattisena imagen sisällä, koska sitä ei ole tarkoitus muuttaa.

5 LÄHTEET

CMake. About CMake. Luettu 9.6.2020. https://cmake.org/overview/.

Cppcheck. 2020. Luettu 9.6.2020. http://cppcheck.sourceforge.net/.

Docker. What is a Container? Luettu 8.6.2020. https://www.docker.com/resources/what-container.

Gcovr. 2019. Luettu 9.6.2020. https://gcovr.com/en/stable/.

Git. Etusivu. Luettu 8.6.2020. https://git-scm.com/.

Jenkins. Jenkins User Documentation. About Jenkins. Luettu 8.6.2020. https://www.jenkins.io/doc/.

Log4cpp. 2017. What is log4cpp? Luettu 9.6.2020. http://log4cpp.sourceforge.net/.

Mäkelä, K. Lead DevOps Architect. 2019. Focusing your software process with Acceptance Test Driven Development and DevOps practices. Luento. Tietotekniikan tulevaisuusseminaari 3.9.2019. Tampereen ammattikorkeakoulu.

Option. Etusivu. Luettu 9.6.2020. https://www.option.com/.

Remion Confluence. Simo’s Thesis. Kick-off meeting notes. 22.10.2019.

Robot Framework. Etusivu. Luettu 9.6.2020. https://robotframework.org/.