TAMPEREEN
AMMATTIKORKEAKOULU

PALVELIMIEN MONITOROINNIN JATKO-KEHITTÄMINEN

Antti Leppänen

Opinnäytetyö
Marraskuu 2017
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka

LEPPÄNEN ANTTI:
Palvelimien monitoroinnin jatkokehittäminen

Opinnäytetyö 28 sivua, joista liitteitä 0 sivua
Marraskuu 2017

Opinnäytetyössä jatko kehitettiin yrityksen monitorointijärjestelmää .Net Framework kehysympäristöllä.

Alkutilanteena oli, että järjestelmän tavallisimpiin valvontakohteisiin oli jo toteutettu monitorointia kuten levytilan-, verkon- ja muistin käytön, sekä muun numeerisen datan seuranta. Työssä toteutettiin uusia vastaavanlaisia seuranta metodeja, sekä kehitettiin mahdollisuus monipuolisempaan ja hallittavampaan monitorointiin.

Lopputuloksena oli monitorointijärjestelmä vanhan rinnalle, jonka pohjalta uusien kohteiden seuranta mahdollistui. Tulevaisuudessa järjestelmä tulee helpottamaan valvonnan laajentamista ja hallintaa.

Asiasanat: monitorointi, valvonta, seuranta

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree program of Information Technology
Software Engineering

ANTTI LEPPÄNEN:
Development of server monitoring

Bachelor's thesis 28 pages, appendices 0 pages
November 2017

In this thesis, there was developed servers monitoring system for a company with .Net Framework.

The Initial situation was that systems most common monitoring targets like disc space, net usage and memory usage were already developed and in use. In the project, there was created new similar monitoring methods, and also program with possibility for versatile and manageable monitoring was developed.

In the end, there was monitoring system alongside the old one, based on which new monitoring targets were possible to be controlled. In the future, the monitoring system will make extending and managing a whole monitoring process much easier.

Key words: monitoring

SISÄLLYS

1 JOHDANTO 7
2 KÄYTETYT TEKNIIKAT 9
2.1 Kolmannen osapuolen osat 9
2.1.1 CollectD 9
2.1.2 StatsD 10
2.1.3 Graphite 10
2.1.4 Grafana 10
2.1.5 Cabot 11
2.2 Ohjelmoinnissa käytetyt tekniikat ja kirjastot 12
2.2.1 C# 13
2.2.2 .Net Framework 14
2.2.3 Tyyppi-introspektio ja Reflektio 14
2.2.4 Protobuf-net 15
2.2.5 Topshelf 15
3 ONGELMAN KUVAUS 16
3.1 Vanhan järjestelmän kuvaus 16
3.2 Vanhan järjestelmän puutteet ja viat 16
3.3 Vanhan järjestelmän vahvuudet 18
4 PROJEKTI 19
4.1 Monitoroinnin laajennukset vanhan järjestelmän puitteissa 19
4.2 Monitorointijärjestelmän osien uusiminen 20
5 TOTEUTETTU OHJELMISTO 21
5.1 Collector 22
5.1.1 Collector.Jobs 23
5.2 MainMonitor 23
5.2.1 MainMonitorin asetustiedosto 24
5.2.2 MainMonitor.Jobs 25
5.3 Client 25
5.4 Muut moduulit 26
5.4.1 MainMonitor.Alerting 26
5.4.2 MainMonitor.Common 26
5.4.3 Scheduler3 27
5.4.4 SimpleLogger 29
5.4.5 TCPConnectionHandler 29
6 YHTEENVETO 33
6.1 Kehitettävää ja jatkotoimenpiteet 33

LÄHTEET 35

ERITYISSANASTO tai LYHENTEET JA TERMIT (valitse jompikumpi)

UDP	(User Datagram Protocol) Yhteydetön tiedonsiirtoprotokolla
LINQ	(Language Integrated Query) Datakysely tekniikka
CLR	(Common Language Runtime) Virtuaalikone .Net ohjelmien ajoon
CLI	(Common Language Infrastructure) Tekninen standardi
TCP	(Transmission Control Protocol) Yhteydellinen tiedonsiirto-protokolla

1 JOHDANTO

Tänä päivänä tietojärjestelmät näyttelevät merkittävää roolia lähestulkoon jokaisessa suomalaisessa yrityksessä riippumatta siitä, onko kyseessä tietotekniikkayritys vai ei. Luonnollisesti tietoteknilliset yritykset ovat täysin riippuvaisia tietojärjestelmänsä toimivuudesta sekä laadusta, ja täten tietojärjestelmien tarkkailu ja laadunvalvonta ovat lähes välttämätön osa tällaisen yrityksen toimintaa. Tietojärjestelmän pettäessä voi yrityksen tuotekehitys pysähtyä kokonaan tai pahimmillaan estää tuotteen tai palvelun loppukäyttäjää käyttämästä ostamaansa tuotetta tai palvelua, ja siksi yritykset usein haluavat tarkkailla järjestelmänsä tilaa.

Tietojärjestelmien seuranta ja tarkkailu eli monitorointi on yleensä ainakin osittain automatisoitua. Tarkkailtavaa tietoa vähintäänkin kerätään ja talletetaan yhteen paikkaan, josta se saadaan koostettua helpommin tulkittavaan muotoon. Monesti monitorointijärjestelmä myös hälyttää automaattisesti jonkin valvotun kohteen ollessa ei-halutussa tilassa. Tällöin voidaan puhua jo automaattisesta valvonnasta, jolloin monitorointijärjestelmän käyttäjän ei tarvitse kuin reagoida hälytyksen edellyttämällä tavalla. Joskus valvontajärjestelmä saattaa jopa itse korjata joitain löytyneitä ongelmia, mutta tällöin voidaan pohtia, että onko kyseessä enää vain seuranta- ja tarkkailutyökalu?

Toimeksiantajayrityksenä opinnäytetyöhön toimi Entteri Professional Software Oy, joka on vuonna 1994 perustettu ohjelmistokehitysyritys. Entterin päätuote AssisDent on suun terveydenhuollon potilashallintajärjestelmä, joka on Suomessa markkinoiden johtavassa asemassa. (Entteri Oy, Viralliset verkkosivut).

Opinnäytetyön tavoitteena oli saada aikaiseksi toimeksiantajayritykselle valvontajärjestelmä, joka kattaa yrityksen nykyiset valvontatarpeet, jotta asiakkaiden palvelut ja yrityksen tuottavuus pystyttäisiin turvaamaan mahdollisimman pitkälle. Tavoitteena oli myös vähentää ja helpottaa tietohallinnon työmäärää tekemällä järjestelmän tarkkailusta mahdollisimman yksinkertaista, ja täten myös vapauttaa tietohallinnon työaika resursseja muuhun työhön. Projektin alkaessa oli tiedossa, että tavoitteeseen ei päästäisi kokonaisuudessaan opinnäytetyön aikana, mutta sitä edistettäisiin niin pitkälle kuin mahdollista.

Opinnäytetyön tarkoituksena oli korjata puutteita tai mahdollisesti myös vikoja, joita toimeksiantajayrityksen entisessä monitorointijärjestelmässä oli. Ongelmalliset osat olisi tarkoitus vaihtaa uusiin, toimivampiin ja monipuolisempiin osiin, tai uudistaa koko valvontajärjestelmä, mikäli sille olisi tarvetta. Tarkoitus oli saada valvontajärjestelmästä mahdollisimman adaptiivinen, jotta se palvelisi myös tulevaisuuden muuttuvia monitorointitarpeita.

2 KÄYTETYT TEKNIIKAT

Tässä kappaleessa esitellään sekä jo-olemassa olleessa monitorointi järjestelmässä osat että monitoroinnin uudelleentoteutuksessa käytetyt tekniikat.

Koska opinnäytetyö käsittelee monitoroinnin koko kaarta tiedon keräyksestä tietojen tulkintaan ja hälytysten tekemiseen, on tekniikat esitelty kevyesti sillä tarkkuudella, että lukijan olisi tarkoitus ymmärtää, mikä on minkäkin tekniikan, palvelun ja taustaprosessin tehtävä. Tarkoitus tässä kappaleessa ei ole esitellä tekniikoiden käyttöä, palveluiden asentamista tai tekniikoiden paremmuutta muihin vaihtoehtoisiin tekniikkoihin verrattuna.

2.1 Kolmannen osapuolen osat

Monitorointiin on jo ennestään tehty useita ohjelmia, jotka kattavat koko monitorointi prosessin tai vain pienen osan sitä ja kaikkea siltä väliltä. Osa palveluista on maksullisia, mutta tarjolla on myös useita ilmaisia ja täysin käyttökelpoisia ohjelmia. Seuraavaksi on esitelty opinnäytetyön monitorointi projektissa jo-käytössä olleita ohjelmia siinä järjestyksessä, kuin niitä käytettäisiin oikeassa monitorointi prosessissa.

2.1.1 CollectD

CollectD on C:llä kirjoitettu metriikkatietojen keräykseen kehitetty avoimen lähdekoodin ilmainen taustaprosessi. Tietoja voidaan kerätä paikallisesta käyttöjärjestelmästä, ohjelmista, lokeista, tai ulkoisista laitteista, ja se voidaan varastoida joko paikallisesti tai lähettää toiseen varastoon verkon yli. CollectD:n mukana saa yli 100 liitännäistä, joiden avulla voidaan laajentaa sen yhteensopivuutta muihin järjestelmiin, tai laajentaa kerättäviä kohteita. Koska CollectD on kirjoitettu C-kielellä, se on kevyt ja käytettävissä helposti myös sulautetuissa järjestelmissä.

2.1.2 StatsD

StatsD taustaprosessi metriikan siirtoon. Eri ohjelmointikielille on toteutettu StatsD kirjastoja, joiden avulla omasta ohjelmakoodista voidaan lähettää metriikkatietoa omalle StatsD palvelimelle UDP:n avulla. Palvelimella oleva taustaprosessi kuuntelee kaikkien käytössä olevien kirjastojen lähettämää liikennettä, jonka palvelin lähettää sitten haluttuun backendiin, kuten Graphiteen. (Datadog, 7.8.2013)

2.1.3 Graphite

Graphite on vuonna 2008 Apache 2.0 avoimen lähdekoodin lisenssillä julkaistu yritystason monitorointi työkal, jonka tehtävänä on pitää sisällään aika-sarjoitetun numeerisen tiedon tallentaminen, ja esittää kuvaajia tallennetun tiedon pohjalta (Graphiten viralliset dokumentaatio verkkosivut). Se on tehty toimimaan Linux-pohjaisille laitteille. Graphite ei ole tiedonkerääjä, mutta se tukee useita eri tiedonkeruu tekniikoita; esimerkkinä CollectD. Lisäksi Graphite on yhteensopiva useiden tiedonvälitys-, visualisointi- ja monitorointitekniikoiden kanssa, joista esimerkkeinä opinnäytetyössä käytetyt tekniikat: StatsD (tiedonvälitys), Grafana (Visualisointi) ja Cabot (Monitorointi) (Graphiten viralliset dokumentaatio verkkosivut).

2.1.4 Grafana

Grafana on numeraalisen tiedon visualisointiin tarkoitettu palvelu, ja on kykenevä piirtämään tallennettujen metriikoiden pohjalta useita erilaisia kuvaajia, joista muutamia on esitetty kuvassa 1. Grafana ei itse säilö tietoa, vaan on ainoastaan tietoa tulkitsevassa roolissa, ja on kykenevä tulkitsemaan tietoa useista tietolähteistä. Myöhemmin Grafanaan on toteutettu myös hälytyksien aiheuttaminen tulkitun tiedon perusteella.

KUVA 1. Esimerkkikuva Grafanan visualisoinnista (https://grafana.com/grafana)

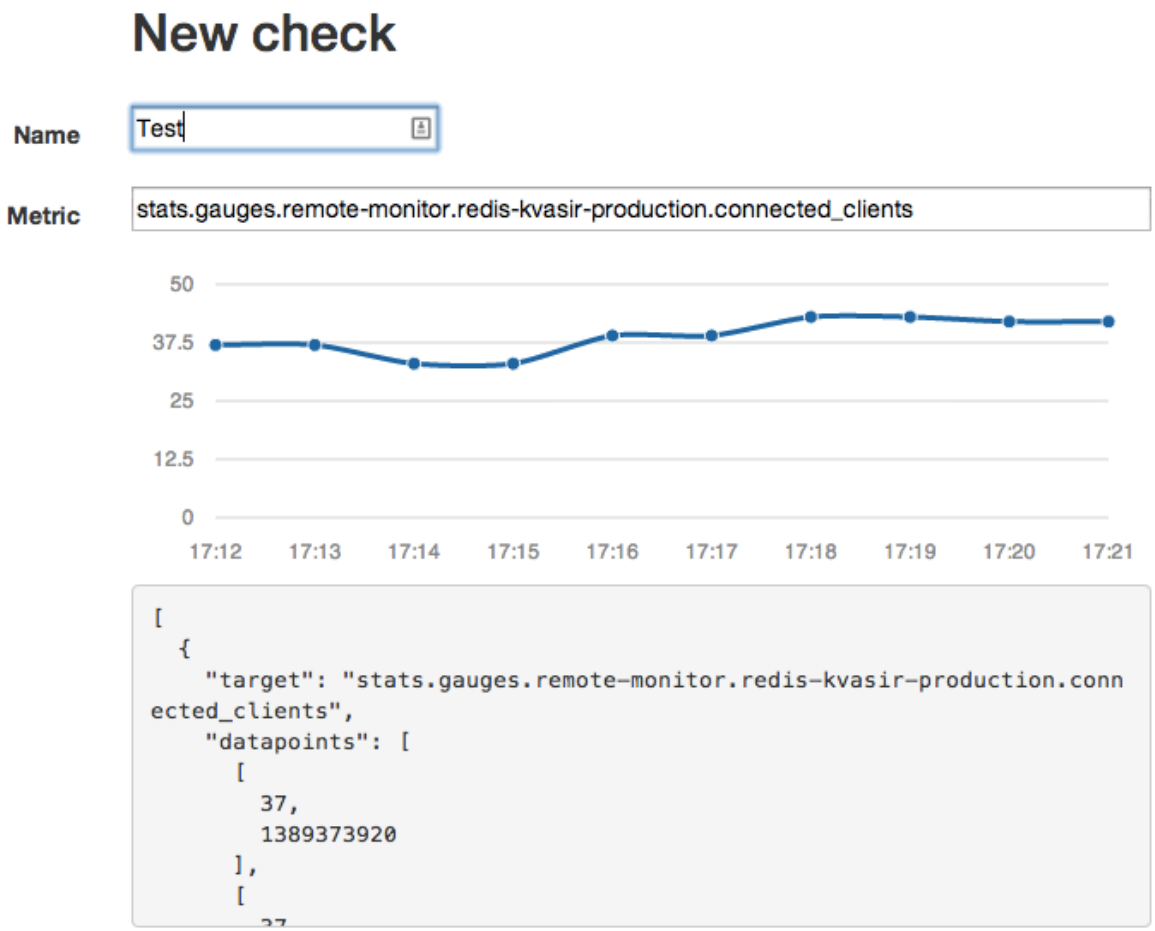
2.1.5 Cabot

Cabot on Pythonilla ja Djangolla kirjoitettu ilmainen avoimen lähdekoodin monitorointi- ja hälytys järjestelmä, joka kykenee tekemään http-, Jenkins, ja Graphite tarkastuksia, joista tässä opinnäytetyössä oleellisena Graphite tarkastukset. Cabot voi tehdä hälytyksen käyttäen sähköpostia, puhelinta tai Hipchatia.
(HackerEarth Engineering, 2017)

Graphite tarkastus toimii antamalla Cabotille halutut Graphiten metriikka tiedot. Esimerkkinä Graphite voisi sisältää tiedot yksittäisen laitteen kiintolevyjen käytöstä, jolloin Graphiteen oltaisiin tallennettu kyseiset tiedot esimerkiksi metriikoiden ”system.disk.C.usage”, ”system.disk.D.usage”, ”system.disk.E.usage” jne. taakse. Tällöin Cabottiin syötetty metriikka ”system.disk.*.usage” valvoisi kaikkia levyasemia.

Kuvassa 2 on esimerkkikuva Cabotin Graphite tarkastelun metriikka kentästä, sekä tarkastelun tuloksista kuvaajassa. Kuva on Cabotin sivuilla oleva esimerkkikuva, jonka metriikasta voidaan päätellä, että tulostaulukossa on todennäköisesti kuvattu Redis-ohjelman asiakaspäätteiden lukumäärää ajan funktiona. Kuva ei kerro, mutta hälytys

voisi tapahtua vaikkapa silloin kun asiakaspäätteiden lukumäärä putoaa alle 35:n. Tällöin Hälytys tapahtuisi ajanhetkellä 17.14.



KUVA 2. Esimerkkikuva Cabotin Graphite tarkastuksen metriikasta ja tuloksesta (http://cabotapp.com/use/graphite-checks.html)

2.2 Ohjelmoinnissa käytetyt tekniikat ja kirjastot

Opinnäytetyö tehtiin pitkälti ohjelmoimalla itse tarvittavat monitoroinnin osat. Ohjelmointi suoritettiin käyttämällä Visual Studio 2015 C# kielellä .Net ympäristössä.

Visual Studion Solution Explorerin ikonit on opinnäytetyön kuvien tulkitsemisen kannalta tarpeellista. Ikonit ovat esitettynä taulukoissa 1 ja 2, joista kaikkia ei ole käytetty opinnäytetyössä.

TAULUKKO 1. Visual Studion Solution Explorerin ikonit (https://msdn.microsoft.com/en-us/library/y47ychfe.aspx)

Icon	Description	Icon	Description
	Namespace		Method or Function
	Class		Operator
	Interface		Property
	Structure		Field or Variable
	Union		Event
	Enum		Constant
	TypeDef		Enum Item
	Module		Map Item
	Extension Method		External Declaration
	Delegate		Error
	Exception		Template
	Map		Unknown
	Type Forwarding		

TAULUKKO 2. Visual Studion Solution Explorerin Signaali ikonit (https://msdn.microsoft.com/en-us/library/y47ychfe.aspx)

Icon	Description
<No Signal Icon>	Public. Accessible from anywhere in this component and from any component that references it.
	Protected. Accessible from the containing class or type, or those derived from the containing class or type.
	Private. Accessible only in the containing class or type.
	Sealed.
	Friend/Internal. Accessible only from the project.
	Shortcut. A shortcut to the object.

2.2.1 C#

C# on syntaksiltaan paljolti Javaa muistuttava korkean tason tyyppiturvallinen olio-ohjelmointikieli Microsoftilta, joka julkaistiin vuonna 2000. C#:n tarkoitus on olla yksinkertainen, yleiskäyttöinen ja kustannustehokas kieli sisältäen muun muassa seuraavia

ominaisuuksia: nullable value types, enumerations, delegates, lambda expressions, Properties, Attributes ja LINQ. (Microsoft, 2015).
Ominaisuudet on mainittu tässä englanniksi, koska monillekaan niistä ei ole kunnollisia suomenkielistä vastinetta (tai niitä ei juuri käytetä).

2.2.2 .Net Framework

.Net kehys on Windowsin sisäinen osa, johon kuuluu joukko ohjelmoinnissa käytettäviä luokka kirjastoja sekä C# ohjelmien ajamiseen käytettävää CLR (common language runtime) virtuaalikonetta, joka on Microsoftin toteutus kansainvälisestä CLI standardista (common language infrastructure). CLI:n tarkoitus on määrittää suoritettavaa koodia ja ajoympäristöä, jotta useita korkea tasoisia kieliä voitaisiin ajaa eri ympäristöissä ilman koodin uudelleenkirjoittamista. CLR:n tehtävä on kääntää C# ohjelman esikäännettyä koodia (Common Intermediate Language) ajettavan laitteen ymmärtämään muotoon ajonaikaisesti.

2.2.3 Tyyppi-introspektio ja Reflektio

Tyyppi-introspektio on tekniikka, jolla ohjelma kykenee tutkimaan objektin metadataa ajonaikaisesti. Tyypillinen esimerkki on, että kysytään olion tyyppiä ajonaikaisesti ja suoritetaan koodia sen mukaan.

Reflektiolla voidaan muokata ja manipuloida koodia ajonaikaisesti objektin metadataan perustuen. Kuvassa 3 on esimerkki, jossa reflektion avulla luodaan muuttujaan ”foo” uusi instanssi oliosta, ja ajetaan kyseisen luokan metodi niin, että olion viite ”foo” on koko prosessin ajan Object tyyppiä, ja täten metodia ei voida suoraan kutsua sen kautta.

```
Object foo = Activator.CreateInstance("Luotavan luokan luokkapolku", "luokan nimi");
System.Reflection.MethodInfo method = foo.GetType().GetMethod("Metodin nimi");
method.Invoke(foo, new object[] { "parametri" });
```

KUVA 3. Esimerkki Reflektion käytöstä

2.2.4 Protobuf-net

Protobuf-net on datan serialisointiin tarkoitettu .Net kirjasto, joka pohjautuu Googlen Protobuf kirjastolle. Protobuf-net ei hyödynnä .Netin serialisointi ominaisuuksia kuten Serializable attribuuttia tai ISerializable rajapintaa. (CodeProject, 2013)

2.2.5 Topshelf

Topshelf on .Net:lle kehitetty viitekehys, jonka avulla konsolisovelluksesta voidaan helposti tehdä Windowsin palvelu, joka helpottaa ohjelman testausta, eikä kehittäjän tarvitse luoda erikseen palvelupohjaista projektia, vaan voi kehittää palvelua samaan tapaan kuin tavallista konsoli projektia. Topshelf on kooltaan hyvin pieni moduuli, kooltaan noin 200kt ilman riippuvuuksia muihin kirjastoihin tai viitekehyksiin.
(Topshelfin virallinen dokumentaatio)

Palvelun luominen onnistuu luomalla Topshelf palveluluokan, jolle määritellään, mitkä julkiset metodit ajetaan palvelun käynnistyessä ja pysähtyessä. Kuvassa 4 on esitetty kooditasolla, kuinka konsolisovelluksesta tehdään palvelu Topshelfin avulla.

```
HostFactory.Run(x =>
{
    x.Service<MainMonitor>(s =>
    {
        s.ConstructUsing(settings => new MainMonitor());
        s.WhenStarted(m => m.Start());
        s.WhenStopped(m => m.Stop());
    });
    x.RunAsLocalSystem();

    x.SetServiceName("MainMonitor");
    x.SetDisplayName("MainMonitor");
    x.SetDescription("Monitoring service for AD");
});
```

KUVA 4. Konsoliprojektin muuntaminen Windows palveluksi Topshelfin avulla

Kuvassa näkyy, miten kirjastolle annetaan palveluna toimiva luokka, palvelun aloitus ja lopetus käskyjen yhteydessä ajettavan metodit sekä palvelun tyyppi. Lopussa vielä asetetaan palvelun nimi ja kuvaus.

3 ONGELMAN KUVAUS

Kappaleessa käydään suurpiirteisesti läpi vanha monitorointijärjestelmä, sekä sen vahvuudet ja heikkoudet, joiden perusteella saatiin määritettyä, mitä osia järjestelmästä pitäisi uusia.

3.1 Vanhan järjestelmän kuvaus

Vanha järjestelmä oltiin toteutettu kappaleessa 2.1 esitettyjen tekniikoiden avulla, sekä kollegan .Net ympäristössä tehdyllä tiedonkeruu ohjelmalla. Järjestelmä koostui kahdesta tiedon kerääjästä, tietosäilöstä, kuvaajien piirto ohjelmasta sekä hälytyspalvelusta.

Tieto kerättiin Windows pohjaisissa laitteissa yrityksessä työskennelleen ohjelmistokehittäjän tuottamalla ohjelmalla (Monitoring.Collector), jonka lähdekoodi oli käytettävissä. Linux pohjaisissa laitteissa tiedon keruu työkaluna oli käytössä CollectD, johon projektin aikana ei tarvinnut tehdä muutoksia. Tietojen kerääjät keräsivät tietoa palvelimien prosessorien ja muistinkäytöstä, verkon lähetys- ja vastaanottoaktiivisuudesta, kiintolevyjen luku ja kirjoitusnopeuksista, sekä muista vastaavista tiedoista.

Tiedon kerääjät lähettivät tietoja StatsD:n avulla Graphiteen, josta tiedot saatiin visualisoitua Grafanan avulla, ja tietojen oikeellisuuden tulkitseminen ja niistä hälyttäminen suoritettiin Cabot palvelulla.

3.2 Vanhan järjestelmän puutteet ja viat

Projektin ensimmäinen haaste alkoi jo vaatimusmäärittelyjen osalta, sillä kukaan projektin osallisista ei tarkalleen tiennyt projektin alkaessa, että mihin kaikkeen monitoroinnin tulisi taipua. Tämä tulkittiin vaatimusmäärittelyn osalta niin, että uudelleentoteutus tulisi olla mahdollisimman hyvin laajennettavissa, koska ei tiedetty tarkalleen mitä kaikkea haluttiin lopulta monitoroida järjestelmässä. Tarkemmat määrittelyt ilmenisivät itsellään projektin, yrityksen päätuotteen valmistumisen ja ajan kuluessa.

Vanhan järjestelmän puutteita ja uusia monitoroitavia kohteita, joita vanha järjestelmä ei kyennyt hallitsemaan, ilmeni aina toisinaan ajan kanssa ja ne kirjattiin ylös sitä mukaan, kun niitä ilmaantui.

Tekstimuotoisen tiedon lähettämisen ja tallentamisen mahdottomuus nousi yhdeksi isommista ongelmatekijäksi siinä vaiheessa, kun oltaisiin haluttu lähettää yrityksen pääohjelmasta aiheutuneet Windowsin tapahtuma lokit talletettavaksi ja hälytettäväksi. Tekstin siirto ja tallennus tulisi todennäköisesti olemaan myös muissakin tulevissa tarkastuksissa hyödyllinen.

Cabotin metriikat oltiin sisällytetty funktion keepLastValue sisään, jolloin Cabot tarkastelee edellistä Graphitesta saatua arvoa. Muutoin mikäli uutta arvoa ei ole Graphiteen tullut, niin Cabot ei suorita tarkastusta. keepLastValue ei kuitenkaan pystynyt pitämään vanhaa arvoa noin kahden tunnin jälkeen, vaan Cabot tulkitsi, ettei uutta arvoa ollut tullut, joka johti virheelliseen tarkastustulokseen. Cabot aiheutti ongelmia paitsi aiemmin mainitussa kahden tunnin ongelmassa, niin myös aiheuttamalla turhia hälytyksiä sen sisään rakennetulla ping-testillä. Jostain edelleen tuntemattomasta syystä Cabot alkoi lähetellä muutaman minuutin välein hälytyksiä epäonnistuneesta ping-tarkastuksesta kaikista testattavista laitteista, ja heti perään ilmoittamalla, että kaikki onkin kunnossa. Hälytykset olivat aina aiheettomia, sillä käsin komentoriviltä suoritettu ping-testi onnistui aina ongelmitta, ja väärät hälytykset saatiin loppumaan Cabot palvelun, tai palvelun sisältämän laitteen uudelleenkäynnistyksellä.

Vanha järjestelmä oli luotu pyörimään Linux pohjaisille laitteille, joka oli siinä mielessä ongelma, ettei kellään monitoroinnin vastuuhenkilöistä ollut isompaa kokemusta Linux laitteista, joten yrityksen kannalta oli parempi, että monitorointi järjestelmä olisi siirretty Windows pohjaisille laitteille toimivaksi. Yrityksen kehitysympäristön ollessa muutenkin .Net olisi myös luontevaa toteuttaa monitorointijärjestelmä vastaavassa ympäristössä. Haasteena oli kuitenkin, että tietoa piti saada kerättyä myös Linux laitteista.

3.3 Vanhan järjestelmän vahvuudet

Vanhan järjestelmän vahvuutena oli sen metriikkatietojen keräys ja analysointi. StatsD, CollectD, Grafana, ja Graphite suoriutuivat tehtävästään moitteettomasti, ja tietoa saatiin helposti kerättyä, säilöttyä ja tulkittua.

Graphite haluttiin pitää ainakin toistaiseksi skaalautuvuutensa ja keveytensä takia, vaikka se ei ollut tarpeeksi laaja kaikkiin vaadittaviin käyttötarkoituksiin, koska monitorointia varten oli saatava tallennettua myös tekstimuotoista dataa. Lisäksi Graphiten tietoja oli helppo tulkita Grafanan avulla, jolla saatiin hyvin monipuolisesti visualisoitua tallennetut metriikkatiedot.

4 PROJEKTI

Tässä kappaleessa kerrotaan projektin etenemisen vaiheista.

Projektin aikana päätehtävänäni oli yrityksen päätuotteen kehittäminen, ja siksi projektia tehtiin aluksi yhtenä päivänä viikossa, ja myöhemmin projektiin käytettävä aika nostettiin kahteen päivään viikossa.

Itse projekti ajateltiin koostuvan kahdesta osasta; Ensimmäisessä vaiheessa tavoitteena oli vanhan monitorointijärjestelmän parantaminen siten, että sen puitteissa olevilla tekniikoilla laajennettiin monitoroitavia kohteita niin paljon kuin oli mahdollista. Samalla tarkasteltiin mitä muita kohteita tarvitsisi monitoroida, mutta joita ei nykyisellä välineistöllä pystytty toteuttamaan. Toisessa vaiheessa, joka otti selvästi suurimman osan ajasta projektissa, oli tarkoitus uudistaa monitorointijärjestelmän osia niin, että uusi järjestelmä kykenisi suoriutumaan myös niistä tarpeellisiksi määritellyistä tehtävistä, joihin vanha järjestelmä ei enää kyennyt, tai jotka eivät toimineet kunnolla vanhassa järjestelmässä. Uusi järjestelmä otettaisiin vanhan rinnalle ja sen kehittyessä voitaisiin sulkea vanhan järjestelmän osia, ja lopulta korvata vanha järjestelmä kokonaan niiltä osin, missä se ei kyennyt suoriutumaan vaadittavalla tasolla.

4.1 Monitoroinnin laajennukset vanhan järjestelmän puitteissa

Ensimmäinen monitoroinnin muutos oli tarkasteltavien Windows koneiden tietyn aseman levytilan seurannan lopettaminen, koska koneiden kyseiselle asemalle oli tyypillistä täyttyä aika-ajoin. Korjaus tehtiin Cabot järjestelmään, johon muutettiin levyjen tilan valvonta metriikka poissulkemaan yksi levy. Tämä onnistui Graphiten virallisen dokumentaation mukaisesti Graphiten funktiolla ”exclude”.
(Graphiten virallinen dokumentaatio)

Toinen laajennus oli sertifikaattien voimassaolon tarkastelu. Toteutus tehtiin Monitoring.Collectoriin ohjelmoimalla kerääjälle uusi tehtävä sertifikaattien voimassaoloajan tarkastelua varten ajettavaksi kerran päivässä. Kaikkien sertifikaattien voimassaoloajat lähetettiin Graphiteen. Tässä ilmeni luvussa 3.2 esitetty ongelma Cabotin kanssa, joka ei

kyennyt tekemään tarkastuksia harvemmin kuin kahden tunnin välein. Tästä syystä korjauksessa päädyttiin tekemään tarkastus useammin kuin olisi ollut tarpeen.

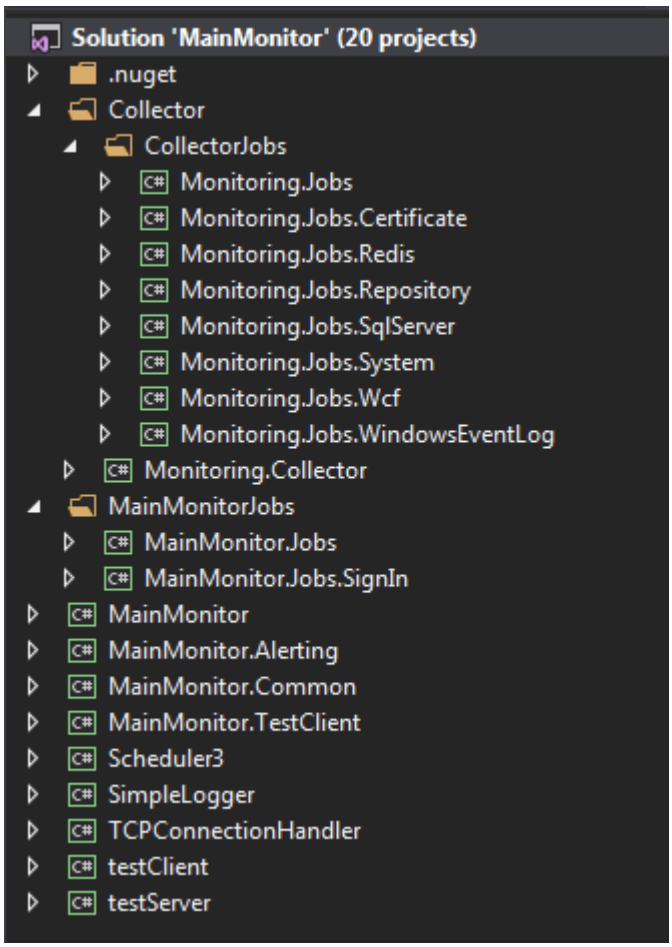
4.2 Monitorointijärjestelmän osien uusiminen

Monitorointijärjestelmän puutteelliset osat oltaisiin voitu paikata korvaamalla riittämättömät osat uusilla valmisohjelmilla, mutta koska vanha järjestelmä koettiin hajanaiseksi ja sitä myötä hankalasti hallittavissa olevaksi useiden eri ohjelmien takia, niin ratkaisuksi haluttiin mieluummin yhtenäisempi ohjelmisto. Tämä tarkoitti sitä, että joko vaihdettaisiin koko järjestelmä toiseen yhtenäiseen ohjelmistoon, tai sitten ohjelmoitaisiin oma omiin tarpeisiin parhaiten soveltuva ohjelmisto. Näistä vaihtoehdoista päädyttiin aloittamaan toteutus oman räätälöidyn järjestelmän ohjelmoimiseen.

5 TOTEUTETTU OHJELMISTO

Tässä kappaleessa selostetaan projektissa valmistuneen ohjelmiston rakennetta, ja toimintaa. Ohjelmisto tuli tavoiteaikaan mennessä käyttökelpoiseksi, mutta sitä ei pystytty ottamaan vielä käyttöön yhteen sopimattoman ympäristön takia.

Lopullinen tuote ei enää ollut vain yksittäinen sovellus tai ohjelma vaan se oli enemmänkin ohjelmisto, koska lopputuloksessa käytössä oli kolme eri sovellusta vähintään kolmelle eri laitteelle. Nämä ohjelmat ovat tiedon kerääjä (Collector), pääohjelma (MainMonitor) ja asiakassovellus (Client). Näiden lisäksi sekä Collectoriin että MainMonitoriin liittyi useita tehtäviä (Jobs), joita kyseiset sovellukset suorittivat aika-ajoin. Ohjelmiston rakenne moduuleittain on esitetty kuvassa 5, jossa ”Monitoring” on vanha nimitys Collectorista, eli ”Monitoring”-alkuiset moduulit viittaavat siis Collectoriin, ja ne olisi tarkoitus tulevaisuudessa uudelleen nimetä ”Collector”-alkuisiksi. Muutenkin moduulien nimet yhä hakevat oikeaa muotoaan, mutta tässä esitellään moduulit niillä nimillä, jotka ne projektin tässä vaiheessa ovat.



KUVA 5. Valmistuneen ohjelmiston rakenne moduuleittain

5.1 Collector

Ohjelmistossa Collectorin tehtävä oli toimia tiedon kerääjänä. Collector luotiin kollegan aiemmin luodun ohjelman Monitoring.Collector pohjalta, johon lisättiin mahdollisuus ottaa yhteys ja siirtää tietoa MainMonitoriin, jossa tietoa voitiin jatko käsitellä halutulla tavalla. Lisättiin myös mahdollisuus ohjata Collectorin suorittamia keräystoimintoja ja muuta toiminnallisuutta MainMonitorista käsin.

Laite, johon Collector asennetaan sisältää moduulin Monitoring.Collector, sekä kaikki halutut moduulit kuvan 5 kansiosta CollectorJobs, sekä apumoduulit Scheduler3, SimpleLogger ja TCPConnectionHandler.

5.1.1 Collector.Jobs

Monitoring.Jobs sisältää kaikille jobeille yhteisiä tietoja; Tärkeimpänä jobien kantaluokka. Muut jobit oltiin toteutettu erillisiksi moduuleiksi, jotta uusia jobeja voitaisiin liittää osaksi monitorointia ajonaikaisesti. Ajatus oli kiehtova, joten se jätettiin ohjelmaan, vaikka se toikin lisää haasteita Collectorin ja MainMonitorin välisen kommunikoinnin kanssa.

Kuvan 5 Collectorin jobeista Certificate ja WindowsEventLog toteutettiin projektin aikana. Muut olivat jo valmiiksi toteutettuja, joten niitä ei käsitellä tässä opinnäytetyössä.

Certificate yksinkertaisesti käy läpi kaikki laitteelle tallennetut sertifikaatit, ja lähettää tiedon sertifikaattien jäljellä olevista voimassaolopäivien lukumäärästä. Jobi käyttää hyväkseen .Net Frameworkin X509Certificate2-luokkaa.

WindowsEventLog kuuntelee Windowsin tapahtumalokin muutoksia ja lähettää halutut lokit eteenpäin, joka oli vanhassa monitorointi järjestelmässä ongelmallista, sillä se ei kyennyt lähettämään, eikä tulkitsemaan tekstimuotoista tietoa. WindowsEventLog voidaan asettaa kuuntelemaan lokeja tyypin, lähteen ja viestin sisällön perusteella. Toimeksiantajayrityksen tapauksessa siis haluttiin asettaa kuuntelu lokeihin, joiden tyyppi on virhe tai varoitus, lähde sisältää merkkijonon ”.net” ja viesti pitää sisällään yrityksen päätuotteen nimen.

5.2 MainMonitor

MainMonitor on ohjelmiston palvelinpään ohjelma, joka on yhteydessä sekä kaikkiin Collectoreihin että Clienteihin, ja täten toimii myös välittäjänä niiden välissä, koska Client ja Collector eivät ole suorassa yhteydessä toisiinsa. MainMonitor ohjaa Collectoria, Collectorin jobeja sekä omia jobejaan, ja monesti tekee sen Clientiltä tulevien komentojen perusteella, mutta ohjauksen voi tehdä myös manuaalisesti MainMonitorin asetustiedostosta App.xaml (tai ajonaikana tiedostosta MainMonitor.vshost.exe.config).

MainMonitor on suunniteltu pyörimään Windows pohjaisissa laitteissa palveluna. Vaikka Visual Studiossa onkin mahdollisuus luoda erillinen palvelu projekti, niin lopulta päädyttiin toteuttamaan palvelu Topshelf kirjaston avulla, eli MainMonitor tehtiin tavallisen konsoliprojektin pohjalta. Topshelfiin päädyttiin siksi että sen käytöstä oli aiempia positiivisia kokemuksia, ja Visual Studion palvelu-projekti oli koettu huomattavasti hankalammaksi.

5.2.1 MainMonitorin asetustiedosto

Asetustiedostoa voidaan monilta osin ohjata suoraan Clientiltä, tai kokonaisuudessaan muuttaa suoraan tiedostosta. Kuvassa 6 on esitetty MainMonitorin asetustiedoston sisältö. Kuvasta nähdään, että tiedostoon on määritelty kolme osiota: test, jobsection ja confParameterSection. Osio test ei nimensä mukaisesti ole kuin testausta varten, eikä täten jää lopulliseen toteutukseen.

```
<configuration>
    <configSections>
        <section name="test" type="MainMonitor.testClass, MainMonitor" />
        <section name="jobsection" type="MainMonitor.ConfigurationClasses.JobSection, MainMonitor" />
        <section name="confParameterSection" type="MainMonitor.ConfigurationClasses.ConfigurationParam
    </configSections>

    <test some="Hahaa!" />
    <jobsection>
        <jobs>
            <job name="SignIn"  run="false" parameter="" interval="*/5 * * * * ?" type="MainMonitor.Jo
            <job name="TestJob" run="false" parameter="" interval="*/1 * * * * ?" intervalIfFailing="
            <job name="PingJob" run="true"  parameter="" interval="*/5 04 * * * ?" intervalIfFailing="
        </jobs>

        <!-- Job specific parameters -->
        <parameters>
            <jobParameterCollection jobName="PingJob">
                <jobParameter parameterName="timeout" value="2000" />
            </jobParameterCollection>
            <jobparameterCollection jobName="TestJob">
                <jobParameter parameterName="successRate" value="1,5" />
            </jobparameterCollection>
        </parameters>
    </jobsection>

    <!-- These are also forwarded to jobs as parameter -->
    <confParameterSection>
        <collectorIps>
            <ip hostname="self" address="127.0.0.1" />
            <ip hostname="emptyIp" address="" />
        </collectorIps>
    </confParameterSection>
```

KUVA 6. MainMonitorin asetustiedoston sisältö

Osio jobsection sisältää kaksi aliosiota jobs ja parameters, joista jobs määrittelee, ajettavien omien jobien tiedot, eli ajetaanko sitä ylipäätään (run), kuinka usein (interval), vaihtoehtoinen suoritus taajuus (intervalIfFailing), jota käytetään jobien epäonnistuessa, ja C# luokan tyyppi, jonka mukaan luokka luodaan käyttäen apuna reflektio tekniikkaa. Kuvassa 6 jokaisella jobilla ”run” tiedon jälkeen myös tieto ”parameter”, joka on osa vanhaa toteutusta ja poissa käytössä. Aliosio parameters sisältää listan nimi-arvo parillisia parametreja, joita suoritettavat jobit käyttävät. Esimerkiksi kuvassa 6 PingJob saa parametrin, jonka nimi on ”timeout” ja arvo 2000.

Osio confParameterSection sisältää MainMonitorin yleisiä tietoja, kuten Collectorien IP osoitteet kuvassa 6. Kaikki tässä osiossa olevat tiedot välitetään myöskin parametrina kaikille MainMonitorin Jobeille. Esimerkiksi PingJob käyttää omaa parametrinsa lisäksi kuvan 6 collectorIps parametrilistaa yhteyskokeilupaketin vastaanottaja listana.

5.2.2 MainMonitor.Jobs

MainMonitoriin on tällä hetkellä toteutettu kaksi käytössä olevaa jobia, jotka ovat PingJob ja SignIn. Jobit kirjoittavat tuloksen lokiin ja palauttavat saadun tuloksen MainMonitorille, josta voidaan esimerkiksi tarvittaessa tehdä hälytys.

PingJob lähettää yhteyskokeilupaketin kaikkiin parametrinaan saamiin osoitteisiin käyttäen apunaan .Netin Ping- PingOptions- ja PingReply luokkia. SignIn suorittaa sisään- ja uloskirjautumisen toimeksiantajayrityksen pääohjelmaan.

5.3 Client

Clientin toteutus jätettiin tämän projektin puitteissa konsolitasolle, ja sen moduuli onkin toistaiseksi nimeltään MainMonitor.TestClient. Tulevaisuudessa tarkoitus olisi toteuttaa graafinen käyttöliittymä, josta pystyttäisiin näkemään yhdellä silmäyksellä, mikäli monitoroitavassa järjestelmässä jokin osa hälyttäisi, tai jos jokin olisi muuten vialla.

Clientin tehtävänä on antaa käyttäjälle palautetta monitoroitavista kohteista ja niiden tilasta, sekä hallita monitorointia antamalla komentoja MainMonitorille. Palaute koh-

teista tarkoittaisi erilaisia hälytyksiä Collectorin ja MainMonitorin tehtäviltä. Komennot ovat tiettyjen tehtävien lopettamis- tai aloittamiskomentoja, tai niiden suoritus tiheyden muuttamiskomento. Tulevaisuudessa komentoja olisi muitakin; Esimerkiksi suoria ongelmien korjaus toimintoja, sekä erilaisten raporttien tulostamista, kuten koontiraportti hälytyksistä, tai tietyn tehtävän hälytysaikaraportti.

5.4 Muut moduulit

Kuvan 5 moduulit MainMonitor.Alerting ja MainMonitor.Common liittyvät monitorointiin, kun taas Scheduler3, SimpleLogger ja TCPConnectionHandler ovat itsenäisiä kirjastoja, jotka luotiin projektin tarpeellisina kylkiäisinä. testClient ja testServer ovat testi moduuleja, eivätkä ole jäämässä lopulliseen tuotteeseen.

5.4.1 MainMonitor.Alerting

Alerting moduuli pitää sisällään kaiken hälyttämiseen tarvittavan koodin. Tällä hetkellä se sisältää ainoastaan staattisen luokan AlertService, jonka ainoa staattinen funktion on SendEmail, jolla voidaan lähettää hälytys asetettuun sähköpostiin. Lähetys on tehty käyttämällä .Net luokkia MailMessage ja SmtpClient.

Alerting on tehty omaksi moduulikseen, koska ei oltu varmoja, että haluttiinko hälyttäminen suorittaa vain MainMonitorista, vai suoraan Collectorista tai jobeista, tai mahdollisesti myös clientistä, mikäli se esimerkiksi ei saa yhteyttä MainMonitoriin.

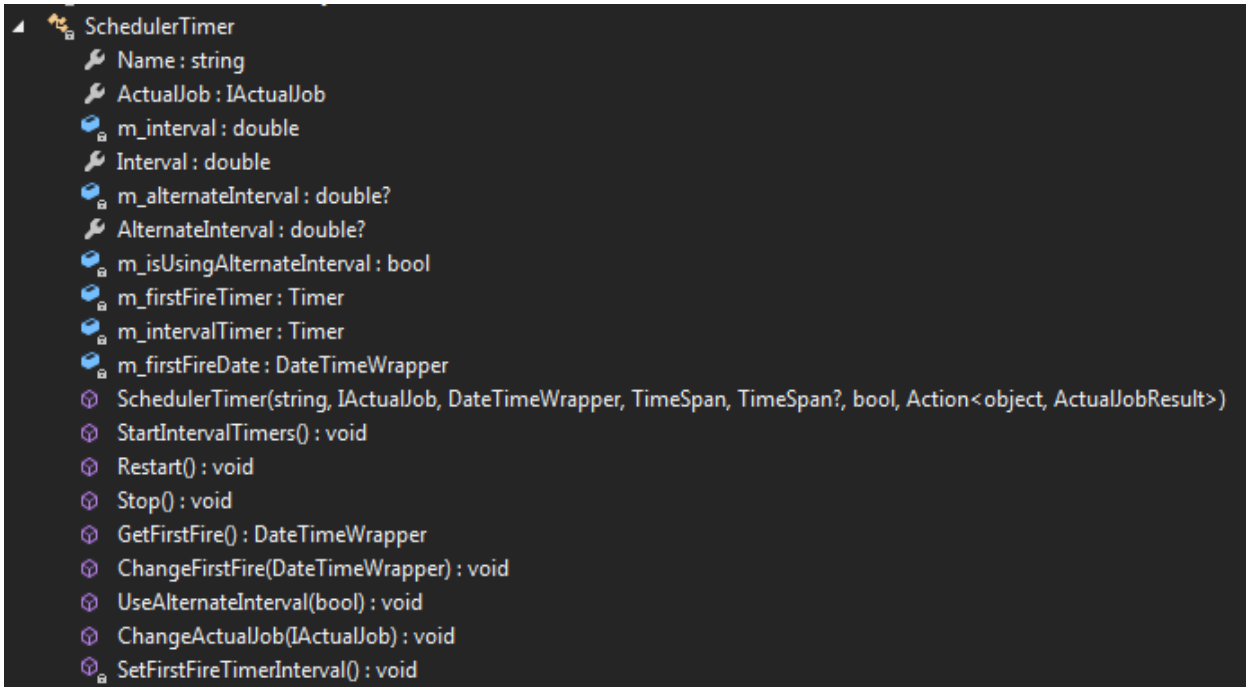
5.4.2 MainMonitor.Common

Common moduuli pitää sisällään tietoja, joita tarvitaan kaikissa päämoduuleissa: Clientissä, MainMonitorissa ja Collectorissa, tai vähintään kahdessa edellisessä. Sisältää tällä hetkellä rajapintaluokat päämoduulien väliseen kommunikointiin. Tulevaisuudessa sisältää todennäköisesti myös joitain utility-tyylisiä metodeja.

5.4.3 Scheduler3

Scheduler3 on monitoroinnista erillinen moduuli, joka nimensä mukaisesti aikatauluttaa sille annettuja tehtäviä, ja sitä käytetään sekä Collectorissa että MainMonitorissa Jobien aikatauluttamiseen ja suorittamiseen. Näitä tehtäviä kutsutaan moduulin sisällä nimellä ActualJob.

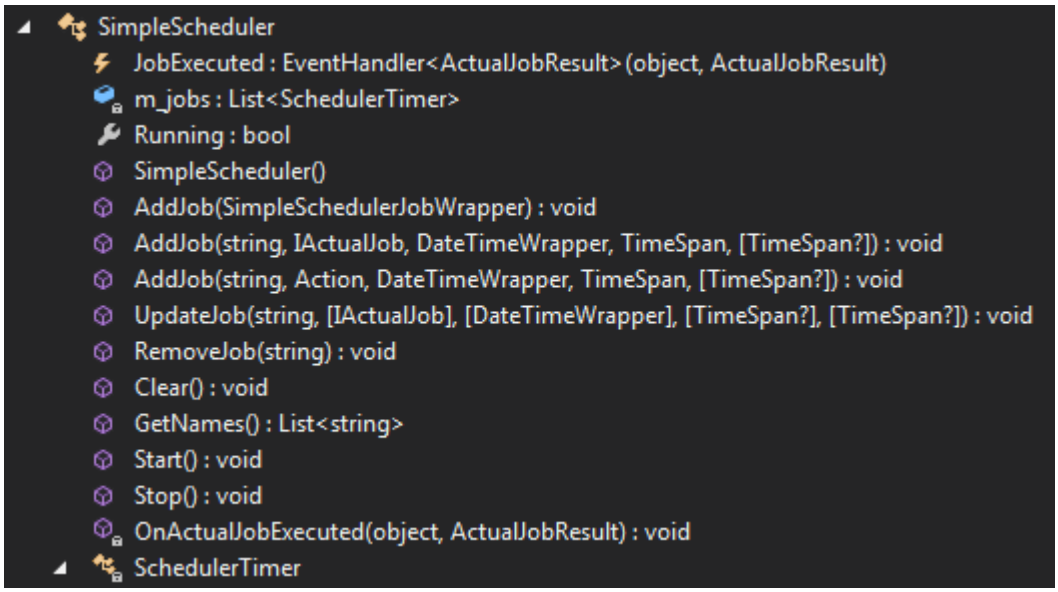
Scheduler3 sisältää luokan SimpleScheduler, joka sisältää listan SchedulerTimer ajastimia. Yksittäinen SchedulerTimer taas pitää sisällään tiedon suoritettavasta tehtävästä, ensimmäisestä suoritusajasta, suoritustaajuudesta ja vaihtoehtoisesta suoritustaajuudesta kuvan 7 mukaisesti. Täten jokaista tehtävää voidaan käsitellä yksitellen SimpleSchedulerin listalla ajastimen nimen perusteella, kuten esimerkiksi vaihtaa suoritustaajuutta tai pysäyttää suorittaminen kokonaan.



KUVA 7. SchedulerTimer luokka

Useiden ajastimien pitäminen listassa tuntui aluksi resurssien hukalta, koska intuitiivisesti ajattelisi, että jokaiselle ajastimelle olisi oma säikeensä. Tästä syystä tehtiin muutama muisti, prosessori ja säie lukumäärä testi sekä tälle toteutukselle että yhtä ajastinta päivittävälle toteutukselle, josta selvisi, että C# koodin kääntäjä (tai mahdollisesti CLR) oli osannut tehdä tarvittavat optimoinnit, eikä toteutuksissa ollut lähes lainkaan eroa testeissä.

SimpleScheduler on oikeastaan lähinnä SchedulerTimer-listan Wrapper-luokka, jonka toiminta mahdollisuudet ovat listaan lisääminen, listan päivittäminen, listalta poistaminen, listan tyhjentäminen ja listassa olevien tehtävien suorittamisen aloittaminen ja lopettaminen; SimpleScheduler luokan sisältö on esitetty kuvassa 8, josta voidaan toiminnollisuuksien lisäksi havaita SimpleScheduler luokan sisältävän yhden eventin, joka laukaistaan aina kun mikä tahansa tehtävä on suoritettu. Täten SimpleSchedulerin instanssin omaava luokka voi tehdä tarvittavat jälkikäsittelyt jokaisen ajon jälkeen, kuten esimerkiksi kirjoittaa lokitietoja tai vaikkapa laskea tehtävien suoritusten lukumäärää.



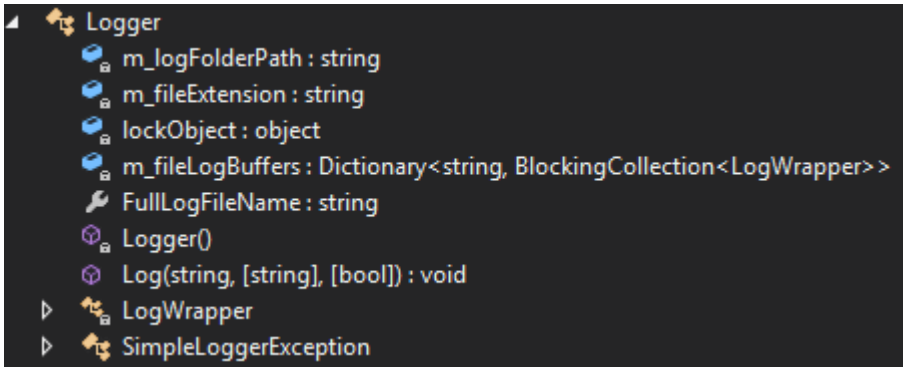
KUVA 8. SimpleScheduler luokka

Eventin mukana kulkee tieto suoritetun tehtävän nimestä, joka on merkkijono SchedulerTimer.Name, sekä tehtävän lopputuloksesta ActualJobResult, joka tehtäväkohtaista tietoa.

SimpleSchedulerille voidaan antaa tehtäväksi yksinkertaisimmillaan mikä tahansa Action tai vaihtoehtoisesti mikä tahansa luokka, joka toteuttaa rajapinnan IActualJob, eli toteuttaa funktion ”ActualJobResult Execute()” sekä rajapinnan IDisposen. Execute ajetaan aina ajastimen umpeutuessa, ja tehtävän poiston yhteydessä ajetaan Dispose.

5.4.4 SimpleLogger

SimpleLogger tehtiin suorittamaan yksinkertaista tiedostoon lokitusta varten. SimpleLogger sisältää yhden staattisen luokan Logger, jonka rakenne on esitettynä kuvassa 9, josta nähdään, että luokalla on vain yksi ulos näkyvä metodi Log.



KUVA 9. Staattinen Logger luokka

SimpleLogger on toteutettu niin, että pääsäie luo staattisessa rakentajassa uuden säikeen, joka tarkastelee loki puskuria, ja kirjoittaa puskurin tyhjäksi tietyin väliajoin. Metodi Log taasen lisää lokitettavan tiedon puskuriin.

5.4.5 TCPConnectionHandler

Nimensä mukaisesti TCPConnectionHandler käsittelee TCP yhteyttä laitteiden välillä. Kirjasto sisältää kaksi pääluokkaa TcpConnectionClient ja TcpConnectionServer, jotka yhdessä toteuttavat asiakas-palvelin mallin.

TcpConnectionServerillä on asiakas yhteyksiä kuunteleva säie sekä lista asiakkaista. Jokaisella listan asiakkaalla taas on oma säikeensä kommunikointia varten, joka poistetaan kun yhteys katkaistaan. Säie hoitaa sekä yhteydeltä tulevan datan kuuntelun että datan lähettämisen käsittelemällä ensin saapuvan data, joka välitetään eteenpäin eventtinä, mikäli dataa on tullut, ja tämän jälkeen säie lähettää sekä tyhjentää oman yhteytensä lähetysdata puskurin. Kuitenkaan säie per yhteys ei ole optimaalisin mahdollinen ratkaisu, koska säikeiden määrä voi nousta hyvin korkeaksi, mikäli yhteyksiä on useita. Ensiapuna TcpConnectionServerille voidaan määritellä yhteyksien enimmäismäärän, mut-

ta toteutusta pitäisi muuttaa isomminkin, jotta saataisiin parempi lopputulos pienemmällä resurssihukalla.

TcpConnectionClient toimii vastaavalla periaatteella, mutta sillä on vain yksi yhteys sille määritetyssä osoitteessa olevaan TcpConnectionServer palvelimeen. Tavallisesti TcpConnectionClient ottaa yhteyden palvelimeen, mutta tcpConnectionServer voi myös lähettää yhteydenottopyynnön UDP:lla IP osoitteen ja porttinumeron perusteella, jolloin vastaava TcpConnectionClient koittaa ottaa yhteyden palvelimeensa.

Luokat on toteutettu niin, että kaikki kuunteluportit ovat muutettavissa halutuiksi. Tiedonsiirron yhteydessä on serialisointiin käytetty kirjastoa protobuf-net, mutta toteutuksen yksinkertaisuuden vuoksi lähes mikä tahansa serialisaattori olisi kelvannut. Protobuf-net valittiin vain siksi, että siitä sattui olemaan aiempaa kokemusta.

Paitsi että TCPConnectionHandleriin on tehty varsinaiset yhteys luokat, joiden välillä voidaan lähettää dataa, niin kirjastoon on myös toteutettu erilliset abstraktit tiedonsiirtoluokat TcpReceiverBase ja TcpSenderBase, jolla on joko TcpConnectionClient tai -Server yhteys instanssi tiedon lähetystä varten. Näistä luokista on tarkoitus periä luokat, joilla on identtiset tiedon lähetys- ja vastaanottometodien nimet ja parametrit.

Esimerkkinä kuvassa 10 on esitetty Clientin ClientToMonitorSender luokka, joka perii abstraktin luokan TcpSenderBase, toteuttaa rajapinnan IClientToMonitor ja jolla on metodit TestMethod, RunServer ja SendTestAlertEmail. Metodit kutsuvat kantaluokaltaan saatua metodia Send, joka lähettää omistamaansa yhteyttä pitkin (kuvassa 10 parametri senderClient) sille annetut objekti tyyppiset parametrit, sekä lähettävän metodin nimen, joka onnistuu attribuutilla [System.Runtime.CompilerServices.CallerMemberName].

```
public class ClientToMonitorSender : TcpSenderBase, IClientToMonitor
{
    public ClientToMonitorSender(TcpConnectionClient senderClient)
        : base(senderClient)
    {
    }

    public void TestMethod(string s, int i)
    {
        Send(parameters: new object[] { s, i });
    }

    public void RunServer(bool run)
    {
        Send(parameters: new object[] { run });
    }

    public void SendTestAlertEmail(string subject, string body)
    {
        Send(parameters: new object[] { subject, body });
    }
```

KUVA 10. ClientToMonitorSender luokan toteutus

MainMonitorissa on taas luotu kuvassa 11 oleva luokka ClientToMonitorReceiver, joka perii luokan TcpReceiverBase, toteuttaa saman rajapinnan kuin Clientin lähettävä luokka IClientToMonitor, sekä vastaavat metodit TestMethod, RunServer ja SendTestAlertEmail vastaavilla parametreilla. Nyt kun MainMonitorin TcpConnectionServerin DataReceived event laittaa ajamaan ClientToMonitorReceiverin kantaluokasta tullut metodi Receive, jolle annetaan saapunut data, niin metodi ohjaa koodin suorituksen saapuneen datan perusteella vastaavaan metodiin vastaavilla parametreilla, mikä onnistuu reflektion avulla. Receive metodin toteutus on esitetty kuvassa 12, josta nähdään, kuinka metodi haetaan reflektion avulla, jolle sitten annetaan saadut parametrit ja lopuksi metodi suoritetaan kutsumalla metodia Invoke.

```
public class ClientToMonitorReceiver : TcpReceiverBase, IClientToMonitor
{
    private MainMonitor m_mainMonitor;
    private ConfigurationHandler m_confHandler;

    public ClientToMonitorReceiver(MainMonitor mainMonitor, ConfigurationHandler confHandler)
    {
        m_mainMonitor = mainMonitor;
        m_confHandler = confHandler;
    }

    public void RunServer(bool run)
    {
        if (run)
        {
            m_mainMonitor.Start();
        }
        else
        {
            m_mainMonitor.Stop();
        }
    }

    public void TestMethod(string s, int i)
    {
        Console.WriteLine("String: " + s);
        Console.WriteLine("int: " + i);
    }

    public void SendTestAlertEmail(string subject, string body)
    {
        Alerting.AlertService.SendMail(subject, body);
    }
```

KUVA 11. ClientToMonitorReceiver luokan toteutus

```
    /// <summary>
    /// Redirects received transferdata to right method
    /// </summary>
    /// <param name="t">received data</param>
    /// <returns>Whether redirect is successful or not</returns>
    public bool Receive(TcpTransferData t)
    {
        System.Reflection.MethodInfo method = this.GetType().GetMethod(t.MethodName);
        if (method == null)
        {
            // Couldn't find corresponding method
            return false;
        }
        var parameters = method.GetParameters();

        object[] methodArgs = new object[parameters.Length];

        for (int i = 0; i < parameters.Length; i++)
        {
            using (MemoryStream stream = new MemoryStream(t.MethodArgs[i]))
            {
                methodArgs[i] = Serializer.Deserialize(parameters[i].ParameterType, stream);
            }
        }
        method.Invoke(this, methodArgs);

        return true;
    }
```

KUVA 12. TcpReceiverBase luokan Receive metodin toteutus

6 YHTEENVETO

Kappaleessa analysoidaan projektin lopputulosta, sekä käydään läpi toteutettuun ohjelmistoon liittyviä kehitysajatuksia ja jatkotoimenpiteitä.

Opinnäytetyön tavoitteena oli parantaa toimeksiantajayrityksen tietojärjestelmän luotettavuutta ja yksinkertaistaa valvontaa tietäen, että projektin aikana ei saataisi lopullista ohjelmistoa valmiiksi.

Projektin päättyessä toteutettu ohjelmisto antoi vain hieman lisähyötyä valvontaan, mutta siihen kuitenkin saatiin luotua puitteet, joiden pohjalta valvonnan jatkokehittäminen olisi helpompaa ja melko nopeaa.

Opinnäytetyön aikana opittiin paljon .Net Frameworkin valmiskirjastoista, C#:n ominaisuuksista, ohjelmointitekniikoista (kuten reflektio), kolmannen osapuolen toteutuksista, sekä tietenkin monitoroinnista yleensä.

6.1 Kehitettävää ja jatkotoimenpiteet

Toteutettuun ohjelmistoon on tulevaisuudessa suunniteltu tässä kappaleessa esitettyjä kehitysideoita toteutettavaksi.

TcpConnectionServer on turhan raskaasti toteutettu siltä osin, että sen jokaiselle yhteydelle on oma säikeensä. Tämä voitaisiin korjata esimerkiksi suorittamalla kaikkien yhteyksien kuuntelu ja lähettäminen yhdessä säikeessä. Tässä ongelmaksi voi tulla, ettei yksi säie ehdi suoriutua kaikista tulevista ja lähetettävistä tiedoista, jolloin voitaisiin esimerkiksi hallitusti lisätä käytettävien säikeiden määrää.

Lokitus ajetuista jobeista, jobien tuloksista ja hälytyksistä olisi parempi säilöä tietokantaan kuin tekstitiedostoon, sillä silloin raporttien luominen olisi huomattavasti helpompaa, sekä tietojen haku onnistuisi hallitummin esimerkiksi kuvaajien piirtämistä varten.

Ohjelmaan olisi tarkoitus toteuttaa graafinen käyttöliittymä, kuten kappaleessa 5.3 on mainittu ja esitetty.

Hälyttäminen pelkkään sähköpostiin ei ole riittävä, vaan tarvittaisiin mahdollisuus lähettää hälytys myös tekstiviestillä sekä yrityksen kommunikaatio ohjelmiin.

Ohjelmisto on tällä hetkellä mahdollista ajaa vain Windows ympäristössä. Jotta ohjelmistoon saataisiin laajempi alusta tuki, niin se voitaisiin koittaa tehdä Monolla tai .Net Corella, jotka toteuttavat CLI:n ja sallivat laajemman alustatuen kuin .Net Framework.

LÄHTEET

Entteri Oy. Viralliset yrityksen verkkosivut. Yrityksestä. Luettu 20.11.2017. http://www.assisdent.fi/asiakaspalvelu/entteri-oy/

CollectD. Viralliset verkkosivut. collectd – The system statistics collection daemon. Luettu 20.11.2017. https://collectd.org/

Datadog. Oliver P. StatsD. what it is and how it can help you. Julkasitu 7.8.2013. Luettu 20.11.2017. https://www.datadoghq.com/blog/statsd/

Graphite. Viralliset dokumentaatio verkkosivut. Overview. Luettu 20.11.2017. http://graphite.readthedocs.io/en/latest/

Graphite. Viralliset dokumentaatio verkkosivut. Tools That Work With Graphite. Luettu 20.11.2017. http://graphite.readthedocs.io/en/latest/tools.html

Grafana. Viralliset verkkosivut. Luettu 20.11.2017. https://grafana.com/grafana

HackerEarth Engineering. Monitoring and alert system using Graphite and Cabot. Julkaistu 21.3.2017. Luettu 20.11.2017. http://engineering.hackerearth.com/2017/03/21/monitoring-and-alert-system-using-graphite-and-cabot/

Cabot. Viralliset verkkosivut. Graphite checks. Luettu 20.11.2017. http://cabotapp.com/use/graphite-checks.html

MSDN. Class View and Object Browser Icons. Luettu 20.11.2017. https://msdn.microsoft.com/en-us/library/y47ychfe.aspx

Microsoft. Introduction to the C# Language and the .NET Framework. Julkaistu 20.7.2015. Luettu 20.11.2017. https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework

Ecma international. C# Language Specification. Neljäs painos. Sivu 15. Julkaistu kesäkuussa 2006. Luettu 20.11.2017. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf

CodeProject. Protobuf-net: the unofficial manual. Julkaistu 25.8.2013. Luettu 20.11.2017. https://www.codeproject.com/Articles/642677/Protobuf-net-the-unofficial-manual

Topshelf. Viralliset dokumentaatio verkkosivut. Topshelf Key Concepts. Luettu 20.11.2017. https://topshelf.readthedocs.io/en/latest/overview/faq.html

Graphite. Viralliset dokumentaatio verkkosivut. Functions. Luettu 20.11.2017. http://graphite.readthedocs.io/en/latest/functions.html

MSDN. X509Certificate2 Class. Luettu 20.11.2017. https://msdn.microsoft.com/en-us/library/system.security.cryptography.x509certificates.x509certificate2(v=vs.110).aspx