Niko Junnila

VIRHERAPORTOINTI-JÄRJESTELMÄ

Opinnäytetyö
Tieto- ja viestintätekniikan koulutus

2019

Kaakkois-Suomen
ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tekijä	Tutkinto	Aika
Niko Junnila	Insinööri (AMK)	Toukokuu 2019

Opinnäytetyön nimi

Virheraportointijärjestelmä

57 sivua
1 liitesivu

Toimeksiantaja

GoodLife Technology Oy

Ohjaaja

Lehtori Niina Mässeli

Tiivistelmä

Tämä opinnäytetyö dokumentoi virheraportointijärjestelmän suunnittelun ja toteutuksen. Järjestelmä koostuu kolmesta erillisestä osasta: ohjelmakirjastosta, raportointiohjelmasta ja pilvipohjaisesta rajapinnasta. Toimeksiantajana työlle toimii GoodLife Technology Oy, jonka tuotteiden tueksi järjestelmä kehitetään.

Tavoitteena oli luoda järjestelmä, jolla voidaan kerätä ja tallentaa tietoja ohjelmissa esiintyvistä vikatilanteista. Kerättyjä tietoja voidaan myöhemmin hyödyntää virheiden paikantamisessa ja korjaamisessa, ja niistä voidaan muodostaa tilastoja luotettavuuden ja laadun mittareiksi. Järjestelmän ohella tahdottiin tutustua erilaisiin mekanismeihin, joiden päälle toteutus rakennettiin, ja tuottaa yleishyödyllistä ohjelmakoodia.

Toteutuksessa käytettiin hyväksi Microsoft Windows -käyttöjärjestelmän sovelluskehityspaketin kirjastojen tarjoamia toimintoja käsittelemättömien poikkeuksien havaitsemiseksi ja kaatumisvedoksien luomiseksi. Raportointiohjelman tekemisessä hyödynnettiin Windows Forms -kirjastoa ja .NET-komponenttikirjastoa. Rajapinta rakennettiin Microsoft Azure -pilvipalveluita käyttäen.

Tuloksena syntyi toimintakykyinen ja helposti laajennettavissa oleva ohjelmistokokonaisuus, joka hyödyntää yrityksen muissa tuotteissa käytettäviä teknologioita. Lisäksi järjestelmän tueksi syntyi runsaasti ohjelmakoodia, jota voidaan hyödyntää järjestelmän ulkopuolella esimerkiksi järjestelmätietojen hakemisessa ja lokiviestien kirjaamisessa.

Asiasanat

järjestelmä, ohjelmisto, palvelurajapinta, poikkeukset, raportointi, viat, virheet

South-Eastern Finland University of Applied Sciences

Author	Degree	Time
Niko Junnila	Bachelor of Engineering	May 2019

Thesis title

Crash Reporting System

57 pages
1 page of appendix

Commissioned by

GoodLife Technology Oy

Supervisor

Niina Mässeli, Senior Lecturer

Abstract

This thesis covers the design and implementation process of a crash reporting system. The system consists of three distinct modules which are a library, a reporting program, and an API running in the cloud. The development was commissioned by a company called GoodLife Technology Oy of which products the system aims to support.

The objective was to create a system that would collect and store information about detected faults in software. The collected data could then be analyzed and utilized in debugging, and statistical data could be composed to measure quality and reliability. Along with the system, the aspiration was to familiarize with the mechanisms the system relies upon, and to produce general-purpose code.

The implementation utilizes functionality provided by the SDK for Microsoft Windows operating systems to catch unhandled exceptions and to write memory dumps. The reporting program was built with .NET software framework and Windows Forms class library. The API was deployed on the cloud services provided by Microsoft Azure.

As a result, a functional and expandable crash reporting system came about. The system utilizes the same set of tools and technologies as the company's products. Furthermore, a considerable amount of source code was generated in support of the system. The resulting code base could be employed outside the system to query system information or to implement logging into software, to name a couple of examples.

Keywords

defects, errors, exceptions, libraries, reporting, service interface, software, system

SISÄLLYS

KÄSITTEIDEN MÄÄRITTELY

API	Application Programming Interface, ks. ohjelmointirajapinta
Argumentti	Aliohjelmalle välitettävän parametrin arvo
CI/CD	Continuous Integration / Continuous Delivery, jatkuva integraatio / jatkuva toimittaminen, käytäntö ohjelmistokehityksessä
CIM	Common Information Model, standardi tiedon mallintamisesta IT-ympäristöissä
HTTP	Hypertext Transfer Protocol, tiedonsiirtoprotokolla
IoT	Internet of Things, esineiden Internet
Kirjasto	Ohjelmistokehityksessä käytettävä kokoelma aliohjelmia
Ohjelmointirajapinta	Määritelmä ohjelmanosien välisestä vuorovaikutuksesta
Parametri	Aliohjelmalle välitettävä muuttuja
REST	Representational State Transfer, arkkitehtuurinen tyyli
SDK	Software Development Kit, ohjelmistokehityspaketti
UML	Unified Modeling Language, mallinnuskieli
VEH	Vectored Exception Handling, pinosta riippumaton poikkeuskäsittelyn mekanismi
WBEM	Web-Based Enterprise Management, standardijoukko
XML	Extensible Markup Language, merkintäkieli

1 JOHDANTO

Tässä opinnäytetyössä dokumentoin virheraportointijärjestelmän suunnittelun ja toteutuksen vaiheita pääpiirteisesti. Tavoitteenani on tutustua virheiden etsinnän menetelmiin ja tuottaa virheiden havaitsemiseen ja korjaamiseen hyödyllistä ohjelmakoodia. Järjestelmä kattaa ohjelmakirjaston, erillisen raportointiohjelman ja pilvipohjaisen rajapinnan, joiden avulla ohjelmistoissa esiintyvistä vioista voidaan kerätä, raportoida ja tallentaa tietoja myöhempää analysointia varten.

Virheenetsinnän lähtökohtana on myöntää, että ohjelmassa tulee olemaan vikoja. Paras lähestymistapa ongelmaan on välttää virheitä ohjelmoitaessa samalla toteuttaen virheenetsinnälle hyödyllisiä ominaisuuksia. (Gregoire ym. 2011, 928.) Koodivirheiden syntymistä ei voi kokonaan välttää, eikä kaikkia virheitä pystytä realistisesti havaitsemaan testauksen kattavuudesta huolimatta. Toteutettava järjestelmä tähtää virheiden etsinnän työkaluksi. Käyttäen tiettyjä ohjelmakielen ja käyttöjärjestelmän mekanismeja, voidaan ohjelman tilasta tallentaa virheenetsinnälle arvokkaita tietoja kaatumisen hetkellä. Kehittäjä voi myöhemmin hyödyntää kerättyjä tietoja paikantaessaan ja korjatessaan vikaa.

Työn toimeksiantaja on GoodLife Technology Oy, joka on keskittynyt erityisesti terveydenhuollon kuntoutuksen tarpeisiin suunnattuihin ratkaisuihin. Yritys työllistää tällä hetkellä noin kymmenen henkilöä ja toimii Kotkassa ja Helsingissä. Järjestelmä tulee osaksi yrityksen tuotteita mahdollistaen ongelmien tehokkaamman havaitsemisen ja korjaamisen samalla keräten tietoa tuotteiden vikaherkkyydestä.

Opinnäytetyö koostuu viidestä osasta. Ensimmäinen osa käsittelee aihealueen teoriaa, johon kuuluu mm. kehitystyön esittely tutkimusmetodina, sekä samankaltaisten opinnäytetöiden ja ohjelmistojen tarkastelua. Toinen osa koostuu järjestelmän yleiskuvan ja sen toimintojen suunnittelemisesta. Kolmannessa osassa ryhdytään toteuttamaan järjestelmää ja kokeilemaan sitä käytännössä. Viimeisessä luvussa tarkastellaan lopputulosta, käydään läpi jatkokehitysideoita, verrataan luotua järjestelmää muihin vastaaviin ja esitetään toimenpide-ehdotukset toimeksiantajalle.

2 TEORIA

Tässä luvussa käsitellään ensiksi teoriaa kehittämistyöstä tutkimusmenetelmänä, jonka jälkeen tutustutaan samansuuntaiseen opinnäytetyöhön. Näistä edetään tarkastelemaan jo olemassa olevia ratkaisuja, sekä esitellään työssä käytettävät työkalut ja teknologiat. Lopuksi perehdytään vianetsintään ja sen yleisimpiin menetelmiin.

2.1 Tutkimusmenetelmä

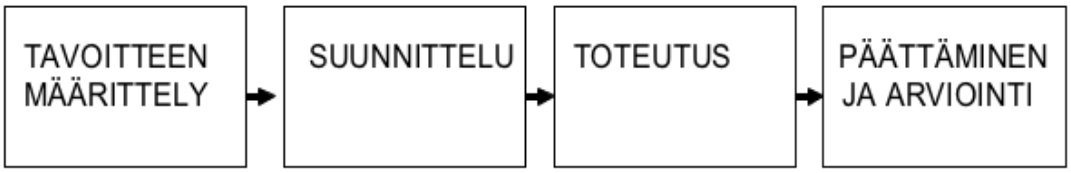
Työssä käytetään prosessuaalisen tutkimuksen menetelmänä tutkimuksellista kehittämistoimintaa. Kehittämistoiminnalle keskeistä on jonkin asetetun tavoitteen saavuttamiseen tähtäävä järjestelmällinen toiminta. Tavoitteena on usein jo olemassa olevan asian parantaminen tai muuttaminen tehokkaammaksi. (Toikko & Rantanen 2009, 14–16.)

Kehittämistoiminta voi olla luonteeltaan suunnittelu- tai prosessiorientoitunutta. Suunnitteluorientoituneessa kehittämistoiminnassa pyritään rajaamaan tarkasti kehittämiseen liittyvät tavoitteet ja prosessit. Suunnitelmasta poikkeamista vältetään. Tällaisella toiminnalla tavoitellaan prosessin ennakoitavuutta ja helppoa ohjailtavuutta. Prosessiorientoituneessa kehittämisessä tunnistetaan, että työn edetessä uutta tietoa syntyy jatkuvasti, ja hyväksytään toimintaympäristön ja -tapojen muuntuvuus. Toimijoiden oppiminen ja toisenlaisten tapojen kokeileminen synnyttää tietoa ja kokemuksia, mikä toimii kehittämisen ohjaavana tekijänä. (Toikko & Rantanen 2009, 49–50.) Tämä opinnäytetyö on luonteeltaan pääasiassa suunnitteluorientoitunutta, jotta aihealue ja lopputulos pysyvät mahdollisimman yhtenäisenä ja yksinkertaisena. Prosessiorientoitunut toiminta on kuitenkin tyypillisempi ohjelmistoprojekteille, johtuen kehitystyön luonteesta. Toiminnan painopiste siirtynee prosessiorientoituneemmaksi jatkokehityksen aikana.

Kehittämisprosessiin kuuluu viisi tehtävää, joista ensimmäinen on kehittämistoiminnan perusteleminen eli vastataan kysymykseen, mitä tehdään ja miksi tehdään. Toinen tehtävä on organisointi, jolla vastataan siihen, kuka tai ketkä tekevät ja millä resursseilla. Kolmas tehtävä pitää sisällään varsinaisen kehittävän toiminnan eli toteutuksen. Neljäs tehtävä on arviointi, jolla viitataan

edellä mainittujen tehtävien tarkasteluun ja pyrkimykseen tuottaa tietoa prosessin ohjauksen tarpeisiin. Viidentenä ja viimeisenä tehtävänä kehittämisprosessiin kuuluu pyrkimys tulosten levittämiseen. (Toikko & Rantanen 2009, 56–63)

Kehittämisprosessin malleista lineaarinen malli (kuva 1) kuvaa parhaiten ohjelmistokehitystä ja se muistuttaa perinteistä vesiputousmallia, johon kuuluu vielä viides vaihe, joka on ylläpito (ks. Kasurinen 2013, 23). Ohjelmistokehityksessä arviointivaihe pitää sisällään testauksen, eikä projekti välttämättä pääty vaan siirtyy ylläpitovaiheeseen. Ylläpitovaiheessa ohjelmiston vikoja korjataan ja uusia toimintoja saatetaan lisätä (Kasurinen 2013, 13). Tämä opinnäytetyö ei kuitenkaan käsittele ohjelmiston ylläpitoa, joten työ mukailee lineaarisen mallin vaiheita.



Kuva 1. Projektityön lineaarinen malli (Toikko & Rantanen 2009, 64)

Lineaarisen mallin ensimmäisessä vaiheessa tavoitteet rajataan selkeästi ja tarvittaessa jaetaan pienempiin osiin. Tavoitteet muodostavat perustan seuraavilla vaiheille. Tavoitteet voivat perustua yksittäiseen ideaan tai havaittuun tarpeeseen. (Toikko & Rantanen 2009, 64)

Suunnitteluvaiheessa selvitetään esimerkiksi, että projektin edellytykset täyttyvät, ja että projekti tukee organisaation tavoitteita. Projektia varten voidaan tuottaa erilaisia analyysejä, kuten riskianalyysi ja resurssianalyysi. Samalla päätetään projektin aikataulusta ja sen toteuttamiseen osallistuvista. Suunnitelmaa saatetaan tarkentaa toteutuksen aikana. (Toikko & Rantanen 2009, 64–65.) Ohjelmistotuotannossa suunnitteluvaiheeseen kuuluu mm. järjestelmän teknisen rakenteen, komponenttien ja tietorakenteen suunnittelu. Suunnitelmaa ei kannata tehdä liian jäykäksi, sillä Toikko & Rantanen (2009, 65) toteavat toteutusvaiheelle olevan tyypillistä, että laadittua suunnitelma saatetaan joutua muuttamaan.

Projekteilla on päätepiste sillä ne ovat ajallisesti rajattuja. Projekteilla on taipumus jatkuvuuteen, mutta uudet ideat on hyvä siirtää tuleviin projekteihin. Projektien päätösvaiheessa ja arvioinnissa tehdään loppuraportti ja esitetään ideat jatkoa ajatellen. (Toikko & Rantanen 2009, 65.)

2.2 Samankaltaiset opinnäytetyöt

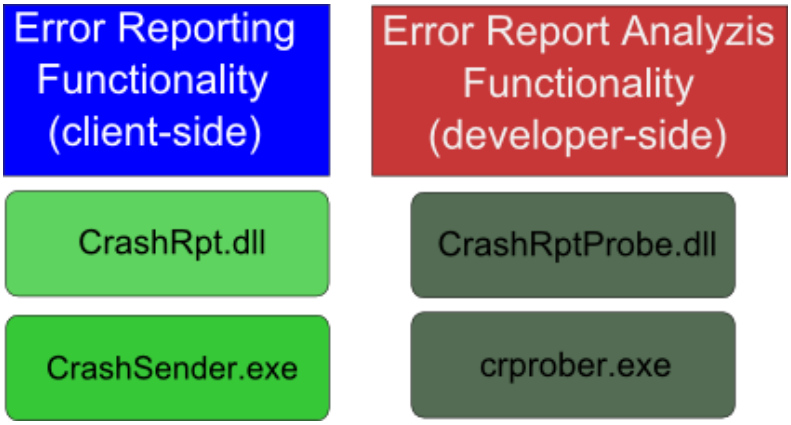
Kilpeläinen (2018) käsittelee opinnäytetyössään poikkeuksia, vianetsintää, kaatumisilmoitusten keräämistä ja analysointia, sekä työkaluja ja niiden käyttöönottoa. Hänen tavoitteenaan oli kehittää ratkaisu vikaraporttien keräämiseksi ja lähettämiseksi ohjelman kaatumisen hetkellä. Kilpeläinen päätyi ottamaan käyttöönsä Google Breakpad -kirjaston, sillä koki sen soveltuvan parhaiten käyttötarkoitukseensa. Johtopäätöksenä hän kirjoittaa, että kaatumisilmoitukset antavat helpon keinon saada yksityiskohtaisia kuvauksia virheistä, joskaan vikojen perimmäisen syyn selvittämiseksi ne eivät välttämättä yksinään riitä.

2.3 Samankaltaiset ohjelmistot

Tässä luvussa esitellään samankaltaisia ohjelmistoja ja käydään läpi niissä käytettyjä arkkitehtuurillisia ratkaisuja. Yhteistä ohjelmistoille on avoimen lähdekoodin lisensointi, mikä mahdollistaa niiden vapaan käytön ja muokkaamisen myös kaupallisessa käytössä. Vaikka mainitut ohjelmistot täyttävätkin suurimman osan tarpeista, oman järjestelmän toteuttamisessa on kuitenkin etunsa. Samojen työkalujen ja teknologioiden käyttö kuin yrityksen muissa tuotteissa, helpottaa järjestelmän käyttöönottoa. Lisäksi oman raportointijärjestelmän laajentaminen ja ylläpitäminen osana olemassa olevia järjestelmiä voi olla vaivattomampaa kuin kolmannen osapuolen ratkaisujen.

2.3.1 CrashRpt

CrashRpt on virheraportointikirjasto C++-ohjelmille, joiden kohdealusta on Windows-käyttöjärjestelmät. Sen ominaisuuksiin kuuluu kriittisten virheiden käsittely asiakasohjelmassa, automaattinen tiedonkeruu virhetilanteesta ja virheraportin luominen. Näiden lisäksi se sisältää ohjelman raporttien lähettämiseksi kehittäjälle. (CrashRpt 2015b.)

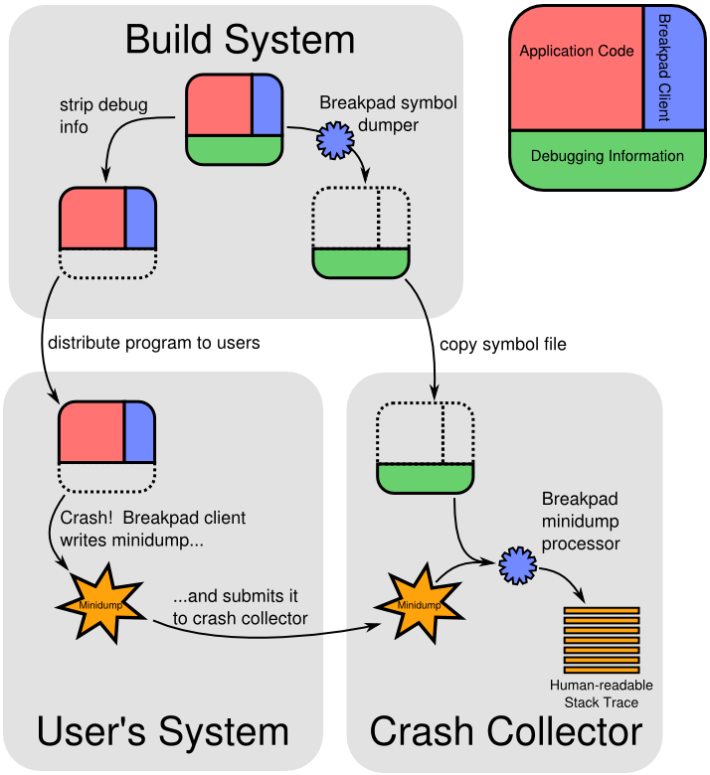


Kuva 2. CrashRpt arkkitehtuuri (CrashRPT 2015a)

CrashRpt:n virheraportointi koostuu kahdesta moduulista, jotka ovat CrashRpt.dll ja CrashSender.exe (kuva 2). CrashRpt.dll sisältää toimintoja poikkeusten käsittelyyn asiakasohjelmassa. CrashSender.exe vastaa virheraporttien lähettämisestä kehittäjälle. Kaksi moduulia tarvitaan, jotta kaatuva ohjelma voidaan sulkea ja virheraportti lähettää. (CrashRPT 2015a.)

2.3.2 Google Breakpad

Google Breakpad on symbolitaulujen ja kaatumisvedosten keräämiselle tarkoitettu kirjasto ja työkalukokoelma. Se muodostuu kolmesta komponentista, jotka ovat asiakasohjelmaan liitettävä kirjasto, symbolitaulut sovelluksesta erottava työkalu ja kaatumisvedokset analysoiva käsittelijä. Breakpad on käytössä monissa sovelluksissa, kuten esimerkiksi Google Chromessa ja Mozilla Firefoxissa. CrashRpt:n lailla se on tarkoitettu käytettäväksi C- ja C++-kielillä toteutetuissa sovelluksissa. Breakpad tukee Windows-, MacOS- ja Linux-alustoja, mutta tuottaa kaatumisvedokset kaikilla alustoilla Windowsin minidump-formaatissa. (Google Breakpad 2015.)



Kuva 3. Google Breakpadin arkkitehtuuri (Google Breakpad 2015)

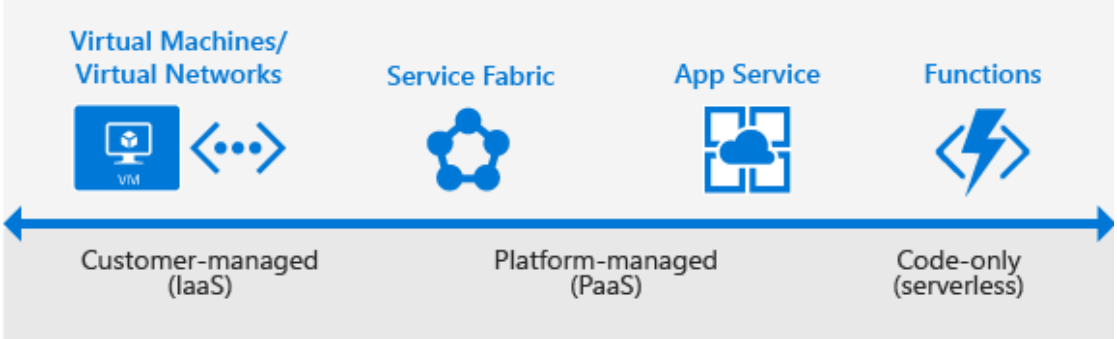
Kuvassa 3 esitetään Breakpadin kolmen komponentin toiminta. Käännösvaiheessa sovelluksesta erotetaan symbolitaulu, jota käytetään kaatumisvedosten analysoinnissa. Kun Breakpadin kirjastoa käyttävä sovellus kaatuu, luodaan virhetilanteesta kaatumisvedos, joka lähetetään analysoitavaksi. Analysointivaiheessa työkalu luo kaatumisvedoksen ja symbolitaulun avulla helposti luettavassa muodossa olevan koosteen sovelluksen pinosta.

2.4 Työkalut ja teknologiat

Kohdealustana on Microsoft Windows -käyttöjärjestelmät, joten luontevin vaihtoehto ohjelmointiympäristöksi on Microsoft Visual Studio. Visual Studio tarjoaa koodieditorin lisäksi työkalut mm. käyttöliittymien rakentamiseen ja Azure-resurssien hallintaan. Järjestelmän jokainen osa voidaan kehittää käyttäen Visual Studiota.

Pilvipalveluiden tarjoajana GoodLife Technologyllä on ollut käytössään Microsoft Azure. Azure on pilvialusta, joka käsittää laajan valikoiman erilaisia skaalautuvia palveluita virtuaalikoneista tietokantoihin ja web-sovelluksien isännöintiin (kuva 4). Azure Functions tarjoaa palvelimettoman ympäristön yksit-

täisten pienien ohjelmien suorittamiseen, ja on omiaan rajapinnan toteutukselle. Azure Functions on myös laajasti käytössä yrityksen tuotteissa mahdollistaen järjestelmän käyttöönoton vaivattomasti. Azuren resurssien käytössä apuna on Microsoft Storage Explorer, joka tarjoaa Visual Studion työkaluja kattavammat toiminnot taulujen ja säilöjen hallintaan.



Kuva 4. Azuren palvelut (Get started guide for Azure Developers 2017)

Kirjaston ohjelmointikielenä toimii C++, koska se on suunnattu sillä kirjoitettujen ohjelmien käytettäväksi, ja koska se ei itsessään sisällä valmiita mekanismeja moniin kirjaston tarjoamiin toimintoihin kuten lokin kirjaamiseen. Rapportointiohjelman ja palvelinrajapinnan toteutukseen käytetään C# -ohjelmointikieltä yhdessä .NET-ohjelmistokomponenttikirjaston kanssa. Yleisin .NET-toteutus on .NET Framework, joka on tarkoitettu sovellusten kehittämiseen Windowsille ja Azurelle (NET Framework Guide 2018). Se voidaan korvata tarvittaessa muulla .NET-yhteensopivalla alustalla, joita ovat esimerkiksi Mono ja .NET Core.

2.5 Vianetsintä

Vialla tarkoitetaan poikkeamaa, joka estää ohjelmaa toimimasta suunnitellulla tavalla (Kasurinen 2011, 50). Vianetsintä on järjestelmällistä toimintaa havaitun vian syyn selvittämiseksi. Ensimmäinen vaihe vian etsimisessä on yrittää toistaa vika. (Gregoire ym. 2011, 945.)

Karpovin (2008) mukaan vianetsintäkeinot voidaan jakaa seuraaviin ryhmiin:

- vianetsintätyökalut (interaktiiviset/graafiset)
- ajonaikainen diagnosointi
- yksikkötestaus
- funktionaalinen testaus (esim. musta laatikko -testaus)
- lokitietojen tarkastelu

- kaatumisvedoksien analysointi
- koodikatselmointi
- staattinen koodianalyysi

On tapauskohtaista, mikä keinoista toimii parhaiten. Jotkin metodit ovat saatavilla helposti sisäänrakennettuna kehitysympäristöissä, kun taas joidenkin keinojen käyttämiseksi on ohjelmaan oltava toteutettuna tarvittavat ominaisuudet. Tässä työssä keskitytään rakentamaan kirjasto, joka tarjoaa rajapinnan lokiviestien kirjoittamiseen ja kaatumisvedosten tallentamiseen.

3 SUUNNITTELU

Tässä luvussa esitellään, mistä komponenteista järjestelmä koostuu ja selvitetään niiden toimintoja ja vastuualueita. Lisäksi luvussa käsitellään järjestelmän käyttökohteet ja käyttötapaukset, sekä kuvataan moduulien välistä toimintaa osana kokonaisuutta. Järjestelmän nimeksi valittiin FaultSys, joka tulee sanoista Fault System. Komponenttien nimeäminen noudattaa samaa kaavaa kuvaten kyseisen komponentin roolia järjestelmässä.

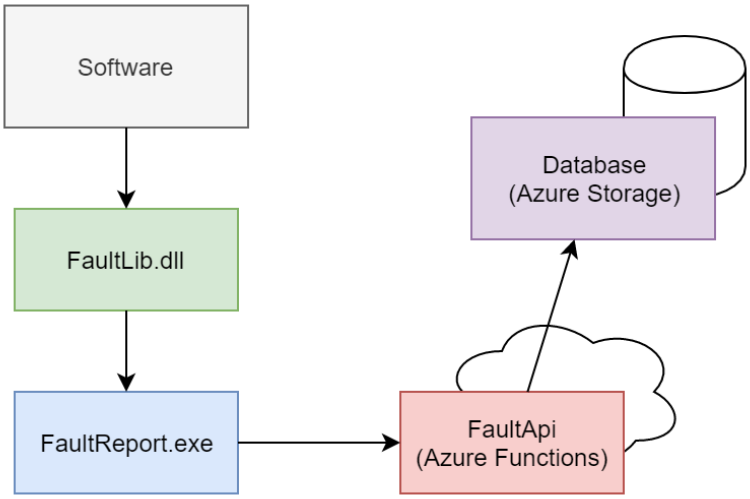
3.1 Käyttökohteet

Virheraportointijärjestelmä on tarkoitettu käytettäväksi osana Windows-ohjelmia, jotka ovat ohjelmoitu käyttäen C++-kieltä. Kaksi selvästi erotettavaa käyttökohdetta ovat loppukäyttäjille suunnatut ohjelma-asennukset ja IoT-laitteiden sovellukset. Ensimmäiseksi mainitussa käyttökohteessa järjestelmä voidaan säätää toimivaksi siten, että virhetilanteessa käyttäjälle näytetään vikailmoitus ja annetaan mahdollisuus raportoida ongelma kehittäjälle. Jälkimmäisessä tapauksessa järjestelmä voi toimia täysin automatisoidusti. IoT-asennuksissa tyypillisesti ohjelman kaatuminen käsitellään siten, että ohjelma käynnistetään heti uudelleen sulkeutumisen jälkeen. Tällöin virheraportointi voi toimia taustalla näkymättömissä.

3.2 Arkkitehtuuri

Järjestelmä koostuu kolmesta moduulista, jotka ovat ohjelmakirjasto FaultLib, raportointiohjelma FaultReport ja rajapinta FaultApi, joka koostuu pilvipalveluna toimivasta ohjelmakoodista ja tietokannasta. FaultLib ja FaultReport ovat

tarkoitettu asiakasohjelman käytettäväksi. FaultApin tarvitsemana pilvipalveluna käytetään Microsoft Azuren tarjoamia palveluita, jolloin varsinaista palvelininfrastruktuuria ei kuulu järjestelmään.

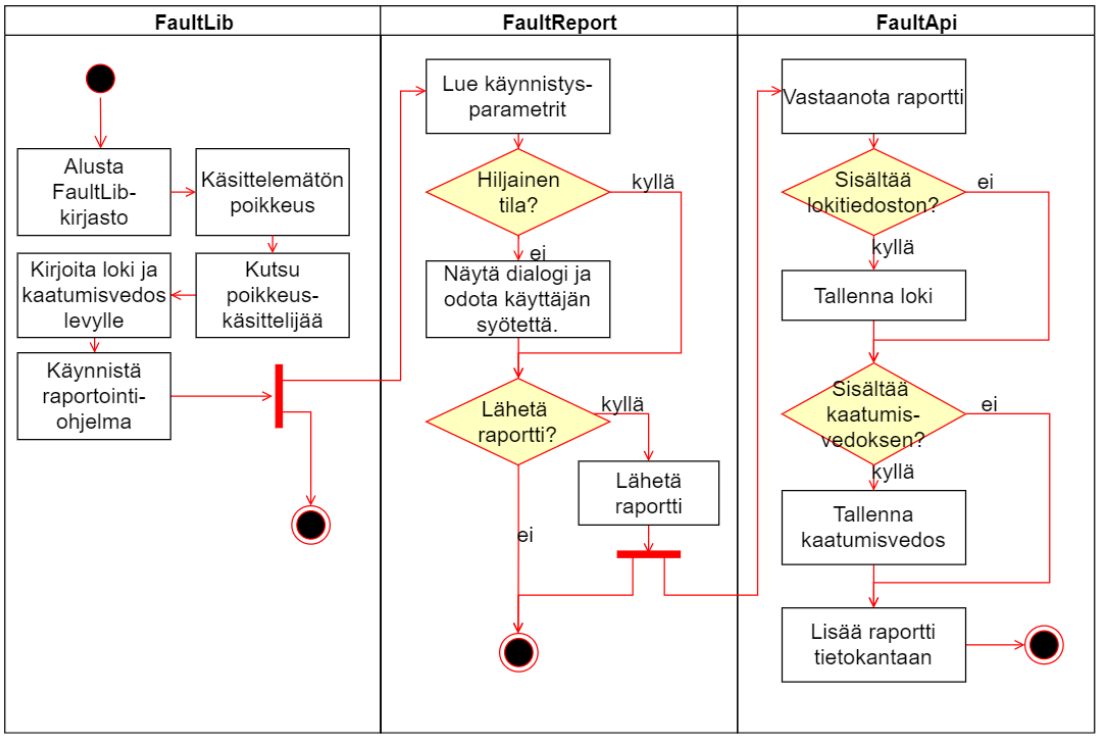


Kuva 5. Järjestelmän arkkitehtuuri

Kuvassa 5 esitellään järjestelmän arkkitehtuuri yleisellä tasolla. Ohjelmisto, joka käyttää kirjaston tarjoamaa poikkeuskäsittelijää, tuottaa kaatuessaan virheraportin. Ennen ohjelman sulkeutumista käynnistetään raportointiohjelma, joka lähettää raportin rajapinnalle. Rajapinnassa vastaanotetut tiedostot tallennetaan myöhempää tarkastelua varten.

3.3 Toimintakaavio

Toimintakaavio on järjestelmän toimintaa yleisellä tasolla kuvaava UML-kaaviotyyppi. Sille tyypillistä on toiminnon suorittamiseen osallistuvien toimijoiden jakaminen omiin lohkoihinsa. Kaaviolla pyritään esittämään tapahtumaketju alusta loppuun. (Kasurinen 2011, 35–36.)



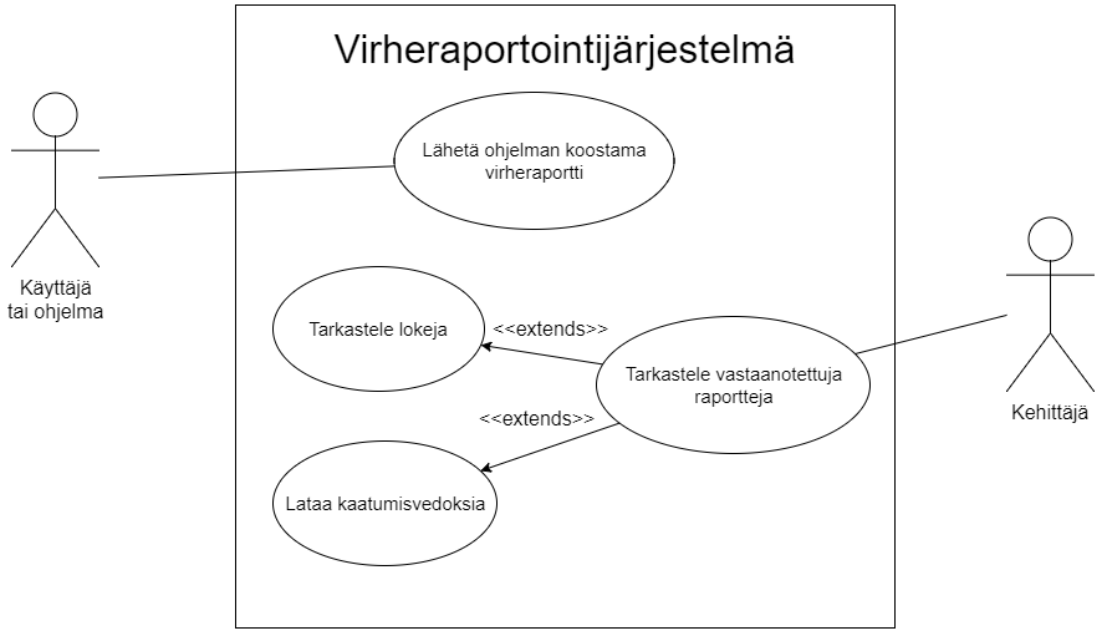
Kuva 6. Toimintakaavio

Kuvassa 6 esitetään tapahtumaketju pääpiirteittäin aloittaen viallisen ohjelman suorittamisen aloittamisesta päättyen raportin luomiseen tietokantaan. Järjestelmän moduulit jakavat kaavion kolmeen osaan. Kunkin moduulin suorituksen päättyminen on merkitty erikseen. Kaaviossa ei ole huomioitu mahdollisia virhetiloja. Järjestelmän toiminta alkaa luonnollisesti asiakasohjelman käynnistyksestä, jonka aikana kirjasto otetaan käyttöön. Kirjaston poikkeuskäsittelijää kutsutaan, kun asiakasohjelma kohtaa käsittelemättömän poikkeuksen, jolloin muodostetaan virheloki ja tallennetaan kaatumisvedos. Ohjelman suorituksen päätteeksi käynnistetään raportointiohjelma, johon järjestelmän toiminnan painopiste siirtyy. Raportointiohjelma lukee vastaanottamansa käynnistysparametrit ja suorittaa niiden mukaiset toiminnot. Raportin käsittely rajapinnassa alkaa, kun raportti vastaanotetaan. Raportin mukana saapuvat tiedostot tallennetaan ja raportille luodaan tietue, mikä päättää järjestelmän toiminnan kokonaisuudessaan.

3.4 Käyttötapauskaavio

Käyttötapauskaaviolla pyritään selvittämään, ketkä järjestelmää käyttävät ja mitkä ovat järjestelmän keskeisen toiminnot (Kasurinen 2011, 30–32). Järjestelmän toimijoina ovat sitä käyttävä ohjelma tai loppukäyttäjä sekä kehittäjä.

Mikäli järjestelmä on säädetty toimimaan siten, ettei käyttäjän toimenpiteitä vaadita, ohjelma itsessään on toimija.



Kuva 7. Käyttötapauskaavio

Kuva 7 esittää virheraportointijärjestelmän käyttötapaukset. Käyttäjä tai ohjelma lähettää virheraportin järjestelmälle ja kehittäjä voi tämän jälkeen tarkastella raportteja ja niihin liittyviä lokeja sekä ladata kaatumisvedoksia. Raporttien lähettäminen pitää sisällään automaattisen prosessin, jossa ohjelma kirjoittaa virhelokin ja kaatumisvedokset levylle ja välittää tiedon niistä raportointiohjelmalle sitä käynnistäessään. Raportointiohjelmalla rajapinnalle lähetetty raportti on käsittelyn jälkeen kehittäjän tarkasteltavissa.

3.5 Ohjelmakirjasto

Dynaamisesti linkitettävä kirjasto, eli lyhyesti DLL tai myös jaettu kirjasto, on ohjelmamoduuli, jota muut kirjastot tai ohjelmat voivat käyttää. Kirjastot ovat keino jakaa ohjelman toiminnallisuudet pienempiin osa-alueisiin, mikä helpottaa koodin uudelleenkäyttöä ja mahdollistaa ohjelman eri osien päivittämisen erikseen. Kirjastojen käyttäminen voi myös pienentää muistinkäyttöä, sillä useampi sovellus voi hyödyntää saman kirjaston koodia. (Dynamic-Link Libraries 2018.)

Tässä luvussa käsitellään kirjaston toimintoja ja niiden merkitystä järjestelmälle, sekä esitellään mekanismeja, joihin toimintojen toteutuksessa nojataan. Päätoiminnot ovat oletuspoikkeuskäsittelijä ja ohjelmointirajapinta lokitietojen ja kaatumisvedosten kirjoittamiselle. Niiden tukena on joukko pienempiä toimintoja, joista merkittävimmät ovat järjestelmätietojen ja kutsupinon hakeminen.

3.5.1 Poikkeuksien käsittely

Poikkeukset ovat tapa ilmoittaa koodissa toiselle osalle koodia poikkeuksellisesta tilanteesta tai tapahtuneesta virheestä. Poikkeus tapahtuu, kun koodi, joka törmää virhetilanteeseen, ns. heittää poikkeuksen, jolloin koodin normaali suoritus keskeytyy ja siirtyy poikkeuksen käsittelyyn. Käsittelystä vastaava koodi voi sijaita esimerkiksi samassa funktiossa kuin poikkeuksen heittänyt koodi tai jossain kohtaa ohjelman kutsupinoa. (Gregoire ym. 2011, 318.) Käsittelemätön poikkeus johtaa ohjelman suorittamisen päättymiseen (Gregoire ym. 2011, 331).

Structured Exception Handling (SEH), on mekanismi laite- ja ohjelmistotason poikkeuksien käsittelyyn Windows-käyttöjärjestelmässä (Structured Exception Handling 2018). Sen avulla voidaan asettaa ohjelmalle oletuspoikkeuskäsittelijä, joka käsittelee sellaiset poikkeukset, jotka muulloin jäisivät käsittelemättä (ks. SetUnhandledExceptionFilter function s.a.). Vaikkei tiettyä poikkeusta pystyisikään käsittelemään siten, että ohjelman suoritus voi jatkua, on silti hyödyllistä ilmaista tietoja virheestä (Gregoire ym. 2011, 329).

SEH ei kuitenkaan ole täydellinen ja tiettyjen poikkeuksien käsitteleminen sen avulla ei onnistu. Tällaisia tapauksia ovat kekomuistin korruptoituminen, pinon korruptoituminen ja pinon ylivuoto. Kekomuistin korruptoitumisen havaitsemiseksi voidaan käyttää pinosta riippumatonta VEH-poikkeuskäsittelijää (ks. Vectored Exception Handling 2018), joka antaa ilmoituksen poikkeuksesta riippumatta siitä käsitelläänkö itse poikkeusta vai ei. Pinon ylivuodon tai vioittumisen tapauksessa, jotta poikkeus voidaan luotettavasti käsitellä, on käsittely tehtävä poikkeuksen kohdanneen säikeen ulkopuolelta. Vaihtoehtoina on käyttää käyttöjärjestelmän mekanismia, kuten Windows Error Reporting -omi-

naisuutta (ks. About WER 2018), tai luoda ohjelman alussa poikkeuksien käsittelemiselle oma säikeensä. Jokaisella säikeellä on oma pinonsa, jolloin säikeen pinon vioittuminen ei vaikuta toiseen. (Peteronprogramming 2016a; Peteronprogramming 2016b; Peteronprogramming 2017.)

Oletuspoikkeuskäsittelijällä on keskeinen rooli järjestelmässä. Sen tehtäviin kuuluu kerätä mahdollisimman hyödyllistä tietoa virheeseen liittyen, jotta se tarjoaa hyvän lähtökohdan vian etsimiselle ja korjaamiselle. Käsittelijän toimintoihin kuuluu kaatumisvedoksen kirjoittaminen, raportointiohjelman kutsuminen ja virhelokin koostaminen. Virhelokiin kirjataan oletuksena ohjelman suorituksen aikaiset lokiviestit, järjestelmän laitteistotietoja ja kutsupino.

3.5.2 Kaatumisvedokset

Kaatumisvedokset ovat keino tallentaa tietoja ohjelman sen hetkisestä tilasta. Vedos voi sisältää ohjelman koko muistialueen tai Windowsin käyttämän minidump-tiedostoformaatin tapauksessa osia siitä. Kaatumisvedokseen tallennettavien tietojen laatu on valittavissa ja vähäpätöiset tiedot voidaan jättää pois tiedostokoon pienentämiseksi. Tavallisesti vedokseen tallennetaan prosessin säikeiden kutsupinot ja paikallisten muuttujien arvot. (Efficient Minidumps 2005.)

Kaatumisvedokset ovat hyödyllisiä esimerkiksi vaikeasti toistettavien vikojen etsinnässä (Gregoire ym. 2011, 947). Vedoksien analysoimiseksi tarvitaan alkuperäisen ohjelmabinaarin lisäksi kääntäjän sille tuottama symbolitaulun sisältämä tiedosto (Minidump Files 2018). Kaatumisvedokset voidaan avata esimerkiksi Visual Studiossa, joka tuottaa simuloidun prosessin, jonka suoritus on pysäytetty poikkeuksen aiheuttaneeseen koodinosaan. (Crash Dump Analysis 2018.)

Mikäli asiakasohjelma kohtaa käsittelemättömän poikkeuksen, kirjaston poikkeuskäsittelijä tekee ohjelman prosessista kaatumisvedoksen. Vedokseen tallennetaan oletuksena tietoja laitteistosta, prosessista, ohjelmamoduulista, säikeistä ja kohdatusta poikkeuksesta. Kaatumisvedos tallennetaan sen hetkiseen työskentelykansioon, mikäli sitä ei ole muuksi määritetty.

3.5.3 Loki

Kirjasto toteuttaa luokan lokiobjektille (ks. Telles & Hsieh 2001, 177), joka tarjoaa yksinkertaisen rajapinnan lokitietojen kirjoittamiselle. Ominaisuuksiin kuuluvat mm. kutsuvan koodin tiedostonimen ja rivinumeron kirjaaminen, aikaleimat millisekuntien tarkkuudella ja lokitasot, jotka ilmaisevat viestin vakavuuden. Loki voidaan näyttää ohjelman konsoli-ikkunassa tai tallentaa levylle.

Virheiden kirjaaminen lokiin, eli error logging, on virheviestien kirjoittamista pysyvään tallennustilaan, jotta ne ovat käytettävissä vielä ohjelman tai laitteen sammumisen jälkeen. Jälkikäteen virhelokista voidaan nähdä, mikäli ongelman havaitsemista ennen tapahtui virheitä, jotka ovat voineet olla osallisena virhetilanteen syntymiseen. Kaikkea ei kuitenkaan kannata kirjata lokiin, sillä suuren tietomäärän tallentaminen hidastaa ohjelman toimintaa ja voi aiheuttaa hämmennystä loppukäyttäjässä. (Gregoire ym. 2011, 929–930.)

Miksi lokitietojen kirjaus on hyödyllistä? Karpov (2008) listaa hyödyiksi kahdeksan tapausta, jotka hänen mielestään tekevät kirjaamisen hylkäämisestä vianetsintäkeinona vaikeaa:

- Tuotantoversio toimii eri tavalla kuin kehitysversio. Tämä voi johtua esimerkiksi viasta kääntäjässä tai alustamattomasta muistista ohjelmassa.
- Vikaa etsitään turvamekanismista. Etenkin laitteistopohjaiset turvamekanismit voivat sulkea muut vianetsintäkeinot (ks. luku 2.5) kokonaan pois.
- Sellaiset tapaukset, joissa kirjaaminen on yksinkertaisesti ainut käytettävissä oleva keino. Esimerkiksi vika on tapahtunut loppukäyttäjän ohjelmassa.
- Vianetsintä sulautetun järjestelmän ajureista.
- Nopeuttaa virheiden havaitsemista testauksen yhteydessä.
- Lokitiedostojen vertailu keskenään. Tässä voi käyttää tarkoitukseen sopivaa työkalua.
- Vianetsintä etäjärjestelmistä. Hyödyllinen esimerkiksi monen käyttäjän järjestelmissä.
- Rinnakkaisten sovelluksien vianetsintä. Esimerkiksi säikeiden välisen synkronisoitiin liittyvien ongelmien vianetsintä. (ks. Gregoire ym. 2011, 953.)

Virheloki on hyvä keino sellaisten vikojen etsinnässä, jotka eivät toistu. Tällöin on todennäköistä, että ennen vian ilmenemistä tapahtuneet virheet ovat osallisena siihen. Hyvin toteutetun ohjelman lokiviesteistä voi syy selvitä suoraan. (Gregoire ym. 2011, 947)

Kirjaamiseen liittyy myös ongelmia, jotka voivat merkittävästi heikentää sen käytettävyyttä vianetsinnässä. Ongelmat liittyvät koodin luettavuuteen, ylläpidettävyyteen ja suorituskykyyn. Lokin kirjoittaminen vaatii ylimääräistä koodia, mikä haittaa koodin yhtenäisyyttä ja lisää ylimääräistä logiikkaa. Ylimääräisen koodin suorittaminen ja lokiviestien tallentaminen hidastaa ohjelman suoritusta, jolloin kirjaaminen voi jopa luoda tai piilottaa aikariippuvaisia vikoja. Useimmiten lokit saatetaan jättää kokonaan huomiotta. Loki pitää erikseen lukea ja analysoida, jotta se olisi hyödyllinen. Lisäksi ongelmana ovat koodimuutoksien takia syntyneet tarpeettomat ja harhaanjohtavat lokiviestit. (Johnston 2018)

Käydään läpi vaatimuksia ja suuntaviivoja lokijärjestelmän toteutukselle: Lokin koodi on pystyttävä tarvittaessa poistamaan käytöstä esimerkiksi siten, ettei kehitysversiossa lokiin kirjattu tieto tule ollenkaan mukaan tuotantoversioon. Rajapinnan tulee olla mahdollisimman pieni ja kompakti, sekä lokin tallentamisen mahdollisimman nopeaa, jotta ne eivät vaikuta ohjelman suorituskykyyn ja toimintaan. Loki itsessään on oltava selkeä ja helposti analysoitava. Suositeltavaa on liittää lokiviestiin tietoja, jotka tarjoavat tälle kontekstin, kuten ajankohta, viestin lisänneen yksikön nimi, tiedostonimi ja rivinumero. Tapahtumien kirjaamisen lisäksi on hyödyllistä kirjata tietoja esimerkiksi tietokoneesta, jolla ohjelmaa suoritetaan. (Johnston 2018; Karpov 2008.)

3.5.4 Järjestelmätietojen hakeminen

Windows Management Instrumentation (WMI) on Microsoftin toteutus WBEM-standardien (ks. Web-Based Enterprise Management s.a.) mukaisesta hallintajärjestelmästä. Sen avulla voidaan hakea ja muokata järjestelmään liittyviä tietoja CIM-standardin (ks. Common Information Model s.a.) määrittämistä tietomalleista. WMI tarjoaa sovelluksille keinon hallita järjestelmiä myös etäyhteydellä. (About WMI 2018). Virheenetsinnässä on hyödyllistä kirjata tietoja laitteistosta ja siihen asennetuista muista sovelluksista. Liittämällä näitä tietoja

virhelokiin, voidaan virhettä analysoidessa havaita mahdollisia yhteensopivuusongelmia tiettyjen kokoonpanojen kanssa.

3.5.5 Kutsupinon tallentaminen

Kutsupino, eli call stack, pitää sisällään ohjelman tilan. Se koostuu kehyksistä, jotka pitävät sisällään funktiokutsun ja siihen liittyviä muuttujia. Jokainen funktiokutsu luo uuden kehyksen kutsupinoon. (Call stack s.a.) Kutsupinon sisältö voidaan käydä kehys kehykseltä läpi, ja niistä on koostettavissa hyödyllisiä tietoja, kuten kutsutun funktion muistiosoite ja nimi. Stack trace on pinon kehyksistä muodostettu raportti tietyltä ajanhetkeltä (Stack trace s.a.). Nopealla vilkaisulla voidaan nähdä missä osassa koodia vika esiintyy, kun lokiin on liitetty tietoja kutsupinosta.

3.6 Raportointiohjelma

Kun asiakasohjelma kaatuu, se tallentaa virhelokin ja kaatumisvedoksen sekä käynnistää raportointiohjelman, jonka vastuulle raportti siirtyy. Raportointiohjelma pyrkii ensisijaisesti lähettämään tiedot virheestä rajapinnalle. Lisäksi sen tehtäviin kuuluu toimia käyttäjälle näkyvänä käyttöliittymänä.

Ohjelman eri käynnistysparametreilla voidaan valita, mitä toimintoja halutaan käyttää. Esimerkiksi loppukäyttäjälle voidaan näyttää virhesanoma ja tarjota vaihtoehtoa virheraportin lähettämiseksi. Vaihtoehtoisesti ohjelma voidaan määrittää toimivan niin sanotussa hiljaisessa tilassa, tausta-ajona, jolloin raportit voidaan lähettää automaattisesti ilman käyttäjän toimien edellyttämistä.

Raportoija voidaan myös käynnistää taustalla ilman, että virhettä on tapahtunut. Tällöin voidaan yrittää aikaisempien virheraporttien lähettämistä uudelleen, mikäli aikaisemmat yritykset ovat epäonnistuneet. Onnistuneen raportin lähetyksen jälkeen kaikki siihen liitetyt tiedostot voidaan poistaa automaattisesti.

Raportointiohjelman käyttöliittymän toteuttamiseksi tarvitaan käyttöliittymäkirjasto, kuten Windows Forms. Se on .NET Frameworkiin kuuluva kirjasto (Windows Forms overview 2017). Toinen tarkoitukseen soveltuva käyttöliittymäkirjasto on GtkSharp (ks. GtkSharp 2018). Sen etuna on yhteensopivuus eri

alustojen kanssa, mutta sen käyttö edellyttäisi ylimääräisten riippuvuuksien lisäämistä projektiin. Käytettäväksi kirjastoksi valikoitui tästä syytä Windows Forms.

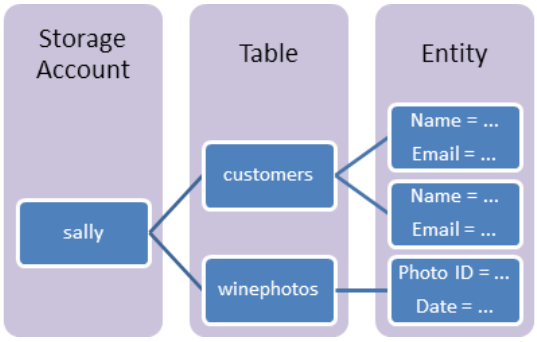
Ohjelmakirjasto ja raportointiohjelma ovat kaksi eri ohjelmamoduulia, sillä se mahdollistaa raportointiohjelman käytön yhdessä millä tahansa ohjelmointikielellä kirjoitetun ohjelman kanssa. Lisäksi ohjelman, joka on kohdannut käsittelemättömän poikkeuksen, tulisi päättää suorituksensa niin pian kuin mahdollista. Monimutkaisten toimintojen, kuten http-kyselyiden lähettäminen ja raporttien hallitseminen, on näin ollen luontevampaa jäädä toisen prosessin vastuulle.

3.7 Rajapinta

Rajapintana toimii pilvipalvelussa suoritettava ohjelmakoodi, joka vastaanottaa ja tallentaa virheraportit ja niiden mukana toimitetut lokitiedostot ja kaatumisvedokset. Toteutuksessa hyödynnetään Microsoft Azuren tarjoamia palveluita. Seuraavissa kappaleissa käydään läpi palvelut ja niiden toiminta, sekä niiden merkitys toteutuksessa.

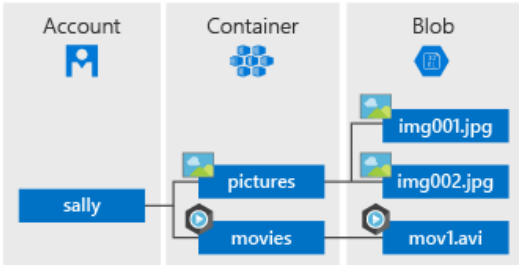
Azure Functions on pilvipohjainen ratkaisu pienten koodinosien, nk. funktioiden, suorittamiseen. Functions tukee muutamaa eri ohjelmointikieltä, joiden joukossa on C#. Funktioita voidaan suorittaa eri tavoin, kuten http-kyselyillä tai ajastettuina tietyin väliajoin. (An introduction to Azure Functions 2017.) Rajapintaa varten toteutetaan funktio, jossa vastaanotetut virheraportit käsitellään.

Azure Table storage on pilvipalvelu, johon voidaan tallentaa jäsennettyä dataa NoSQL:n omaisesti. Tauluihin tallennettava tieto esitetään entiteetteinä (kuva 8) eivätkä taulut pakota entiteettien noudattavan mitään tiettyä kaavaa. Entiteetin muodostuvat joukosta nimi-arvo-pareja. (Azure Table storage overview 2017.) Rajapintaan lähetetystä raportista luodaan yksi entiteetti, johon raportin tiedot tallennetaan.



Kuva 8. Table storagen komponentit (Azure Table storage overview 2017)

Azure Blob storage on jäsentämättömän datan, kuten teksti- ja binaaritiedostojen, tallentamiseen suunnattu palvelu. Se on optimoitu suurien tietomäärien säilyttämiseen ja jakamiseen. Sopivia käyttökohteita ovat tiedon varmuuskopiointi, videon ja äänen suoratoisto, tiedostojen jakaminen ja datan säilöminen analysointia varten. (Introduction to Azure Blob storage 2019.)



Kuva 9. Blob storagen resurssiryhmät (Introduction to Azure Blob storage 2019)

Blob storagessa tiedostot, eli blobit, tallennetaan säilöihin (kuva 9). Sen käyttötarkoitus järjestelmän toteutuksessa on toimia tallennustilana vastaanotetuille loki- ja vedostiedostoille, jotka eritellään omiin säilöihinsä. Tallennetuttujen kaatumisvedosten analysointi voidaan jatkokehitystä ajatellen myöhemmin automatisoida, kun ne ovat helposti saatavilla.

4 TOTEUTUS

Tässä luvussa tarkastellaan yksityiskohtaisemmin järjestelmän avainominaisuuksien toteutusta koodiesimerkkien avulla havainnollistamalla. Ensimmäisenä käydään läpi moduulien toteutusta aloittaen kirjastosta, edeten raportointiohjelmaan ja päättäen pilvipalvelun rajapintaan. Lopuksi havainnollistetaan järjestelmän käyttöä ja toteutetaan yksinkertainen testiohjelma. Järjestelmän moduuleille luodaan omat erilliset Visual Studio -soluutiot. Soluutiot pitävät sisällään projekteja, jotka sisältävät esimerkiksi lähdekooditiedostoja ja muita projektin resursseja.

4.1 Ohjelmakirjasto

Jaettavan kirjaston toteutuksessa käytettiin ohjelmointikielenä C++:aa. Kirjaston soluutio koostuu kolmesta projektista. Ensimmäinen projekteista on yleiset toiminnot sisältävä staattinen kirjasto, jossa on yleiskäyttöisiä luokkia, joita jaettava kirjasto käyttää, kuten loki ja järjestelmätietojen hakemisesta vastaavan luokka. Toinen projekteista on jaettavalle kirjastolle itsessään ja käsittää kirjaston toteuttamat funktiot ja tietorakenteet. Kolmas projekti on tarkoitettu testiohjelmalle, jolla kirjaston toimintoja voidaan kokeilla.

4.1.1 Rajapinnan toteutus

Ohjelmakirjaston kolme päätoimintoa ovat raportointiohjelman käynnistäminen sekä kaatumisvedoksien ja lokiviestien kirjoittaminen. Kirjaston ohjelmointirajapinta toteuttaa kuvassa 10 esitetyt funktiot, jotka mahdollistavat mainittujen toimintojen käyttämisen. Funktiot ovat nimetty ”fault”-etuliitteellä, jotta ne erottuvat muista funktioista. FAULTLIB on esiprosessorin makro, jotka kirjastoa käännettäessä korvataan tekstillä ”__declspec(dllexport)” ja vastaavasti kirjastoa käyttävää ohjelmaa käännettäessä ”__declspec(dllimport)”. Niiden avulla ilmaistaan, ollaanko funktiota tuomassa kirjastosta tai viemässä kirjastoon.

```
128  extern "C"
129  {
130      /* Initialization */
131      FAULTLIB void faultInit(FAULT_CONFIG& config);
132      FAULTLIB void faultUninit();
133
134      /* Report dialog */
135      FAULTLIB bool faultOpenDialog(
136          const char* szMessage = "",
137          const char* szDumpPath = "",
138          const char* szLogPath = "");
139
140      FAULTLIB void faultSetMessage(const char* szMessage);
141
142      /* Dump */
143      FAULTLIB bool faultDump(
144          PEXCEPTION_POINTERS pException,
145          const char* szPath,
146          MINIDUMP_TYPE type = MiniDumpNormal);
147
148      /* Logger */
149      FAULTLIB std::ostream& faultGetLogStream(int level, const char* szFile, int line);
150      FAULTLIB void faultSetLogLevel(int level);
151      FAULTLIB void faultSetLogFilePath(const char* szPath);
152      FAULTLIB void faultSetLogOutputStream(std::ostream& stream);
153  };
```

Kuva 10. DLL-kirjaston tarjoama rajapinta

Kirjaston alustamiseksi on kutsuttava faultInit-funktiota, jonka parametrina syötetään referenssi määritystietueeseen nimeltä FAULT_CONFIG (ks. luku 4.2.2). Funktio vastaa oletuspoikkeuskäsittelijän asettamisesta. Kirjaston resurssien vapauttamiseksi on kutsuttava faultUninit-funktiota. Muut otsikkotiedostossa esitetyt funktiot tarjoavat rajapinnan raportointiohjelman, kaatumisvedoksen ja lokin käyttöön.

4.1.2 Määritykset

FAULT_CONFIG-tietue sisältää kirjaston toimintaan liittyvät oleelliset määritykset, joihin kuuluvat mm. ohjelman tunniste, nimi ja versio, työskentelykansio ja erilaiset toimintoja ohjaavat liput. Liput ovat nimettyjä numeroarvoja, joille voidaan tehdä bittitason operaatioita mahdollistaen useamman lipun käyttämisen samassa muuttujassa. Kuvassa 11 on esitelty määritystietue sekä siihen liittyvät lipputyypit ja -arvot.

```
43   enum class FaultModeFlag : uint32_t
44   {
45       kModeSilent = 0,
46       kModeGui = 1
47   };
48
49   enum class FaultSettingsFlag : uint32_t
50   {
51       kUseOfflineCache = 1,
52       kClearFiles = 1 << 1,
53       kDefault = kUseOfflineCache | kClearFiles
54   };
55
56   enum class FaultReportFlag : uint32_t
57   {
58       kDisableReports = 1,
59       kIncludeDump = 1 << 1,
60       kIncludeLog = 1 << 2,
61       kIncludeStackTrace = 1 << 3,
62       kIncludeSystemInfo = 1 << 4,
63       kDefault =
64           kIncludeDump | kIncludeLog |
65           kIncludeStackTrace | kIncludeSystemInfo
66   };
67
68   #pragma region Operators···
89
90   struct FAULT_CONFIG
91   {
92       const char* szAppId;
93       const char* szApplicationName;
94       const char* szApplicationVersion;
95       const char* szWorkingDirectory;
96
97       FaultModeFlag modeFlag;
98       FaultSettingsFlag settingsFlags;
99       FaultReportFlag reportFlags;
100
101      FaultHandler pfnHandler;
102      FaultCallback pfnCallback;
103      MINIDUMP_TYPE minidumpType;
104
105  #pragma region ctor···
130  };
```

Kuva 11. Kirjaston määritykset

Lipputyypit ovat: FaultModeFlag, jolla ilmaistaan, missä tilassa raportointiohjelma tulee käynnistää; FaultReportFlag, joka kertoo mitä tietoja raporttiin kerätään vai luodaanko raporttia ollenkaan; FaultSettingsFlag, jolla ohjataan raportoinnin muita toimintoja, kuten käytetäänkö yhteydetöntä tilaa tai poistetaanko tiedostot, kun niitä ei enää tarvita (kuva 11).

4.1.3 Poikkeuskäsittelijä

Kirjasto tarjoaa oletuksena poikkeuskäsittelijän, joka hyödyntää SEH-mekanismia. Luvussa 3.5.1 esitettyjä erikoistapauksia, joita SEH-mekanismilla ei pystytä käsittelemään, ei ole huomioitu tässä toteutuksessa. Jatkokehitykselle

VEH-poikkeuskäsittelijän tai erillisen virheidenkäsittelylle suunnatun säikeen toteuttaminen on kuitenkin otollinen.

Poikkeuskäsittelijän tehtävä on määrityksien mukaisesti virhelokin ja kaatumisvedoksen kirjoittaminen ja raportointiohjelman käynnistys, ja sille voidaan antaa funktio-osoitin, jota kutsutaan käsittelyn lopussa, eli nk. callback. Callback-funktiossa voidaan esimerkiksi asettaa raportointiohjelmassa näkyvän virheviestin sisältö. Poikkeuskäsittelijä voidaan vaihtaa alustuksessa toiseen antamalla funktio-osoitin korvaavaan käsittelijään kirjaston määrityksissä.

```
33  struct FAULT_CALLBACK_DATA
34  {
35      PEXCEPTION_POINTERS pExceptionInfo;
36      const char* szDumpPath;
37      const char* szLogPath;
38  };
39
40  using FaultHandler = LONG(*)(PEXCEPTION_POINTERS);
41  using FaultCallback = void(*)(FAULT_CALLBACK_DATA*);
```

Kuva 12. Poikkeuskäsittelijään liittyvät funktio-osoittimet ja callback-tietue

Kuvassa 12 on ote kirjaston otsikkotiedostosta, jossa esitellään käsittelijän ja callback-funktion osoitintyypit. Poikkeuskäsittelijän funktio-osoitin FaultHandler noudattaa UnhandledExceptionFilter-funktion tyyppiä, jonka parametrinä on osoitin EXCEPTION_POINTERS-tietueeseen. EXCEPTION_POINTERS sisältää kuvauksen tapahtuneesta poikkeuksesta (EXCEPTION_POINTERS structure 2018). Callback-funktiolle lähetetään aiemmin mainitun tietueen lisäksi tiedostopolut kaatumisvedokseen ja virhelokiin.

```
5   LONG WINAPI DefaultHandler(PEXCEPTION_POINTERS pExceptionInfo)
6   {
7       const FAULT_CONFIG& config = _faultConfig_get();
8
9       LOG_ERROR("Fatal exception: " << std::hex << pExceptionInfo->ExceptionRecord->ExceptionCode);
10
11      std::string outputPath = config.szWorkingDirectory;
12      std::string moduleName = config.szApplicationName;
13
14      // Generate file name
15      tm t = Utils::GetUtcTime();
16      std::string id = Utils::BytesToHexString(Utils::GetRandomBytes(8));
17
18      std::string fileName;
19      {
20          std::stringstream ss;
21          ss << std::setfill('0')
22              << moduleName.c_str()
23              << "_"
24              << (t.tm_year + 1900)
25              << std::setw(2)<< (t.tm_mon + 1)
26              << std::setw(2) << t.tm_mday
27              << std::setw(2) << t.tm_hour
28              << std::setw(2) << t.tm_min
29              << "_"
30              << id;
31          fileName = ss.str();
32      }
33
34      Utils::CreateSubDirectory(outputPath);
35
```

Kuva 13. Poikkeuskäsittelijän toteutus

Poikkeuskäsittelijän toteutus alkaa hakemalla kirjaston määrityksistä ohjelman nimi ja työskentelykansio, johon virheloki ja vedos tallennetaan. Lisäksi lokiin kirjataan kohdatun poikkeuksien koodi. Näiden jälkeen luodaan tiedostonimi, jonka mukaan tallennettavat tiedostot nimetään. Tiedostonimi koostuu ohjelman nimestä, päivämäärästä ja satunnaisesta kahdeksantavuisesta tunnisteesta. (Kuva 13.)

```
36      // Make a minidump
37      std::string dumpPath = "";
38      if (config.reportFlags & FaultReportFlag::kIncludeDump)
39      {
40          dumpPath = outputPath + fileName + ".dmp";
41
42          if (Dump::Write(pExceptionInfo, dumpPath, config.minidumpType) ==
43          {
44              LOG_ERROR("Failed to write crash dump!");
45              dumpPath = "";
46          }
47      }
48
49      // Gather data
50      std::stringstream logData;
51      bool hasData = false;
52
53      if (config.reportFlags & FaultReportFlag::kIncludeLog)
54      {
55          hasData = true;
56
57          std::istream& in = Log::GetInputStream();
58          char buffer[4096];
59
60          while (in.read(buffer, sizeof(buffer)))
61          {
62              logData.write(buffer, sizeof(buffer));
63          }
64          logData.write(buffer, in.gcount());
65      }
66
67      if (config.reportFlags & FaultReportFlag::kIncludeSystemInfo)
68      {
69          hasData = true;
70          SysInfo sysInfo;
71          sysInfo.GetSystemReport(logData);
72      }
73
74      if (config.reportFlags & FaultReportFlag::kIncludeStackTrace)
75      {
76          hasData = true;
77          StackTrace::Get(pExceptionInfo->ContextRecord, logData);
78      }
79
```

Kuva 14. Kaatumisvedos ja virheloki poikkeuskäsittelijässä

Seuraavaksi käsittelijä kerää tietoja, joista raportti muodostuu. Ensimmäisenä luodaan kaatumisvedos ja sitten haetaan Log-luokan sisäiseltä stream-objektilta ohjelman ajon aikana kirjatut tapahtumat. Sen jälkeen haetaan järjestelmäraportti, josta selviää perustietoja laitteistosta ja ohjelmistoista, kuten prosessori, muistin määrä ja käyttöjärjestelmä. Lopuksi lokia varten koostetaan tietoja kutsupinosta poikkeuksen tapahtumahetkellä. (Kuva 14.)

```
80      // Write the log
81      std::string logPath = "";
82      if (hasData)
83      {
84          logPath = outputPath + fileName + ".log";
85
86          std::fstream log(logPath, std::ios::trunc | std::ios::out);
87          if (log.is_open())
88          {
89              log << logData.str();
90              log.close();
91          }
92          else
93          {
94              logPath = "";
95          }
96      }
97
98      // Invoke callback
99      if (config.pfnCallback)
100     {
101         FAULT_CALLBACK_DATA data;
102         data.pExceptionInfo = pExceptionInfo;
103         data.szDumpPath = dumpPath.c_str();
104         data.szLogPath = logPath.c_str();
105
106         config.pfnCallback(&data);
107     }
108
109     // Execute reporter
110     faultOpenDialog(_faultMessage_get(), dumpPath.c_str(), logPath.c_str());
111
112     std::terminate();
113     return EXCEPTION_CONTINUE_SEARCH;
114 }
115
```

Kuva 15. Poikkeuskäsittelijän loppuosa

Kerätyt lokitiedot tallennetaan lopuksi ”.log”-tiedostopäätteiseen tekstitiedostoon. Virhelokin ja kaatumisvedoksen käsittelyn jälkeen suoritetaan callback-funktion kutsuminen. Kun funktiosta on palattu, ja jotta tiedot voidaan lähettää, käynnistetään raportointiohjelman prosessi. Raportoijalle välitetään virheviestin lisäksi tiedostopolut virhelokiin ja vedokseen. Lopulta käsittely päättyy std::terminate()-kutsuun, joka päättää ohjelman suorituksen. (Kuva 15.)

4.1.4 Kaatusmivedokset

Kaatumisvedoksen kirjoittamiseksi sovelluksen on kutsuttava MiniDumpWriteDump-funktiota, joka sisältyy Windows-alustan ohjelmistokehityspakettiin (Crash Dump Analysis 2018). MiniDumpWrite vaatii argumenteikseen prosessin kahvan ja tunnisteen, kahvan luotuun tiedostoon, luotavan vedoksen tyypin ilmaisevat liput, osoittimen poikkeuksen tiedot sisältävään tietueeseen, osoittimen käyttäjän määrittelemiin tietoihin ja osoittimen callback-rutiiniin, jolle lisätiedot vedoksesta annetaan (MiniDumpWriteDump 2018).

```
3   bool Dump::Write(EXCEPTION_POINTERS* pExceptionPointers, const std::string& filepath, MINIDUMP_TYPE typeFlags)
4   {
5       HANDLE hDumpFile = CreateFileA(filepath.c_str(), GENERIC_READ | GENERIC_WRITE,
6           FILE_SHARE_WRITE | FILE_SHARE_READ, 0, CREATE_ALWAYS, 0, 0);
7
8       MINIDUMP_EXCEPTION_INFORMATION ExceptionInformation;
9       ExceptionInformation.ThreadId = GetCurrentThreadId();
10      ExceptionInformation.ExceptionPointers = pExceptionPointers;
11      ExceptionInformation.ClientPointers = TRUE;
12
13      bool bSuccess = MiniDumpWriteDump(GetCurrentProcess(), GetCurrentProcessId(),
14          hDumpFile, typeFlags, &ExceptionInformation, NULL, NULL);
15
16      CloseHandle(hDumpFile);
17
18      return bSuccess;
19  }
20
```

Kuva 16. Kaatumisvedoksen kirjoittaminen

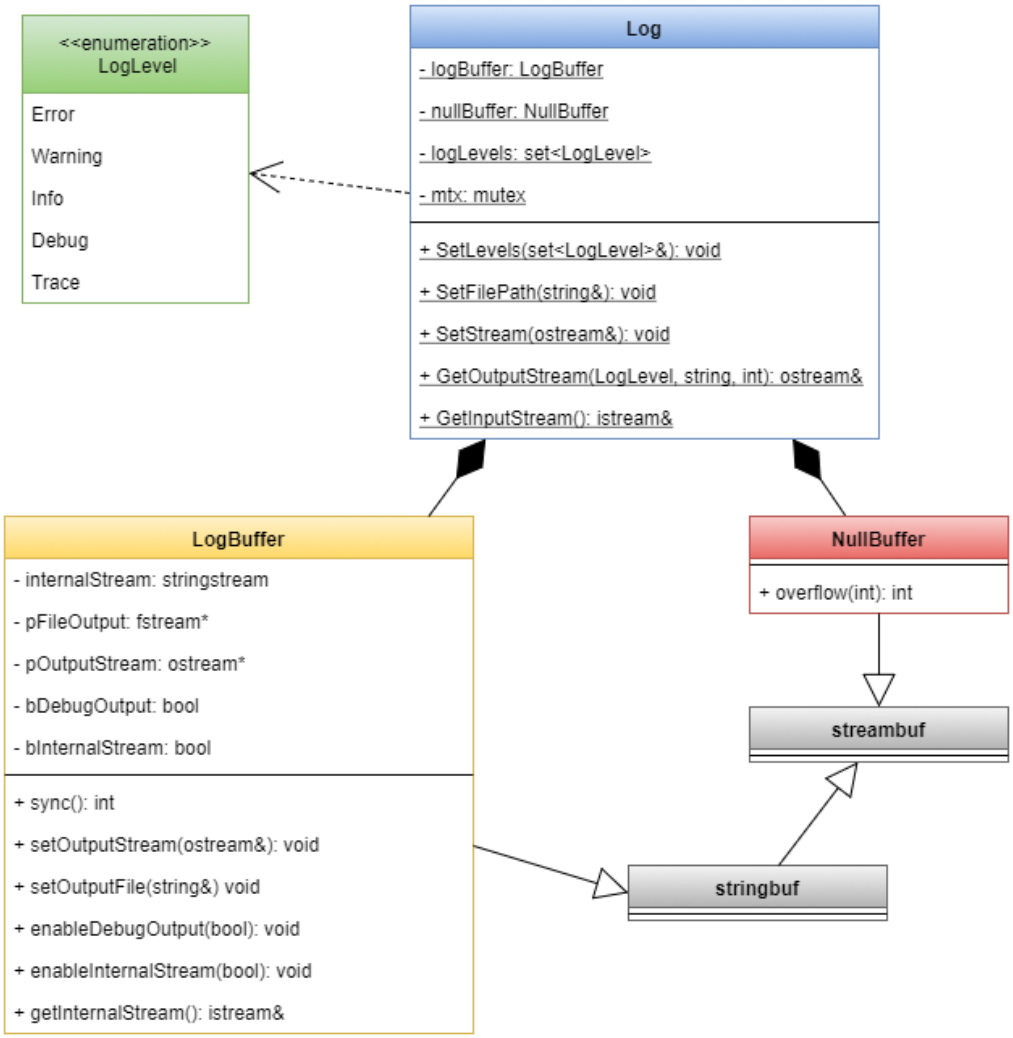
Kuvassa 16 esitetyssä funktiossa vedoksen luominen aloitetaan luomalla sille tiedosto, jonka jälkeen täytetään tietue tapahtuneen poikkeuksen tiedoilla, jotka poikkeuskäsittelijä on vastaanottanut. Prosessin kahva ja tunniste syötetään yhdessä näiden tietojen kanssa MiniDumpWriteDump-funktiolle, joka palauttaa totuusarvon ilmaisten toiminnon onnistumisen. Käyttäjän määrittämää dataa tai callback-rutiinin funktio-osoitinta ei lähetetä, mutta jatkokehityksessä niitä voidaan hyödyntää esimerkiksi lisätietojen liittämiseen suoraan vedokseen.

4.1.5 Loki

Loki toteutettiin staattisena luokkana, vastaanottaa viestejä virtaansa ja vastaa niiden suodattamisesta ja välittämisestä. Seuraavaksi tutustutaan lokiluokan toimintaperiaatteisiin, luokan jäseniin ja riippuvuuksiin, ja lokiluokan toteutukseen osana kirjastoa. Luokka sijaitsee staattisessa kirjastossa, jonka avulla sitä voidaan hyödyntää myös sellaisenaan erillään järjestelmästä.

Oleellinen osa lokijärjestelmän toteutusta ovat viestitasot (LogLevel), joilla ilmaistaan viestin vakavuus. Tasoja on viisi: ”Error”, joka on tarkoitettu virheviesteille; ”Warning”, joka on tarkoitettu huomionarvoisille viesteille, jotka eivät vastaa vakavuudeltaan virhettä; ”Info”, joka on yleislaatuisille viesteille; ”Debug”, jota voidaan käyttää virheenetsinnän apuna; ”Trace”, eli jäljet tai vihjeet, on tarkoitettu ohjelman yksityiskohtaisen toiminnan kirjaamiseen. Lokiluokan tapauksessa on päädytty ratkaisuun, että jokainen taso on kytkettävissä erikseen päälle tai pois, vaikka kirjaston rajapinnan funktiolla voidaan asettaa vain yksi numeerinen arvo, jonka mukaan arvoa vastaava ja sitä pienemmän arvon

omaavat tasot valitaan käyttöön. Pienin numeroarvo on tasolla ”Error” ja suurin tasolla ”Trace”. Mitä pienempi arvo, sitä suurempi vakavuusaste on kyseessä. Käytössä olevat tasot voidaan asettaa Log-luokan SetLogLevels-metodilla.



Kuva 17. Luokkakaavio lokiobjektista

Lokiluokka koostuu kahdesta oliosta, jotka ovat LogBuffer ja NullBuffer. Niiden tehtävä on puskuroida ja käsitellä vastaanottamansa viestit. NullBuffer yksinkertaisesti jättää viestit huomioimatta, tarjoten vain toteutuksen viestien vastaanotolle. Viestit, jotka eivät vastaa käytössä olevia lokitasoja, ohjataan NullBuffer-puskurille. Merkitykselliset viestit, jotka halutaan kirjata, ohjataan LogBuffer-puskuriin, joka pystyy kirjoittamaan viestit tiedostoon, tallentamaan viestit omaan sisäiseen merkkijonovirtaansa, lähettämään viestit debugger-ohjelmalle tai syöttämään viestit määrätylle stream-objektille. (Kuva 17.)

```
131   std::ostream & faultGetLogStream(int level, const char * szFile, int line)
132   {
133       return Log::GetOutputStream(static_cast<LogLevel>(level), szFile, line);
134   }
```

Kuva 18. Lokikutsu kirjaston funktiossa

Log-luokka ei näy kirjaston ulkopuolelle, joten sitä varten on kirjastoon tehtävä funktio, joka välittää kutsut luokalle. Lokiluokan GetOutputStream-metodin paluuarvona on referenssi stream-objektiin, joka käyttää jompaakumpaa aiemmin mainituista puskureista. Metodin sisääntuloparametrit ovat viestitaso, tiedostonimi ja rivinumero, jotka välitetään faultGetLogStream-funktiosta. Välityksen yhteydessä funktion parametrit muunnetaan kirjaston sisäisiin tyyppeihin. (Kuva 18.)

```
155   /* Log Stream Macros */
156   #define faultLogDebugStream     faultGetLogStream(FAULT_LOG_LEVEL_DEBUG, __FILE__, __LINE__)
157   #define faultLogInfoStream      faultGetLogStream(FAULT_LOG_LEVEL_INFO, __FILE__, __LINE__)
158   #define faultLogWarningStream   faultGetLogStream(FAULT_LOG_LEVEL_WARNING, __FILE__, __LINE__)
159   #define faultLogErrorStream     faultGetLogStream(FAULT_LOG_LEVEL_ERROR, __FILE__, __LINE__)
160   #define faultLogTraceStream     faultGetLogStream(FAULT_LOG_LEVEL_TRACE, __FILE__, __LINE__)
161
162   /* Log Line Macros */
163   #define faultLogDebug(x)    faultLogDebugStream << x << std::endl
164   #define faultLogInfo(x)     faultLogInfoStream << x << std::endl
165   #define faultLogWarning(x)  faultLogWarningStream << x << std::endl
166   #define faultLogError(x)    faultLogErrorStream << x << std::endl
167   #define faultLogTrace(x)    faultLogTraceStream << x << std::endl
168
```

Kuva 19. Lokimakrot

Käytön helpottamiseksi jokaiselle lokitasolle toteutetaan makrot, jotka ovat nähtävissä kuvassa 19. Ensimmäisenä on makrot, jotka hakevat lokin stream-objektit faultGetLogStream-funktiolla. Kääntäjä korvaa __FILE__- ja __LINE__-makrot tiedostonimellä ja rivinumerolla. Näin välitetään kirjurille automaattisesti kutsuvan koodin sijainti. Lokiviestien kirjaamiselle tehtiin vielä funktiomuotoiset makronsa, jotka hyödyntävät aiempia makroja ja lisäävät rivinlopun viestien perään. Nämä makrot mahdollistavat haluttujen viestitasojen poistamisen kokonaan lopullisesta ohjelmasta, jos niiden määritys jätetään tyhjäksi tai muutetaan kommentiksi.

4.1.6 Järjestelmätiedot

WMIConnector on kirjaston luokka, joka on yhdistää WMI-palveluun ja tarjoaa tavan kyselyiden ohjaamiseksi palveluun. WMIConnector-objektin luominen yhdistää automaattisesti palveluun, jonka jälkeen kyselyiden tekeminen on suoraviivaista luokan Query-metodilla.

Kuvassa 20 on toteutus Query-metodista. Luokan jäsenmuuttuja pSvc on osoitin IWbemServices-rajapintaan, jonka ExecQuery-funktiolla kyselyt suoritetaan. Query palauttaa osoittimen IEnumWbemClassObject-tietueeseen, jonka avulla voidaan käydä läpi kyselyn palauttamia WBEM-luokkaobjekteja. Kyselyn epäonnistuessa Query palauttaa tyhjän osoittimen.

```
95  IEnumWbemClassObject * WMIConnector::Query(const char * query)
96  {
97      if (pSvc == nullptr)
98          return nullptr;
99
100     IEnumWbemClassObject* pEnum = nullptr;
101
102     HRESULT hr = pSvc->ExecQuery(
103         bstr_t("WQL"),
104         bstr_t(query),
105         WBEM_FLAG_FORWARD_ONLY | WBEM_FLAG_RETURN_IMMEDIATELY,
106         nullptr,
107         &pEnum);
108
109     if (FAILED(hr))
110     {
111         LOG_ERROR("Query failed! (" << std::hex << hr << ")");
112         return nullptr;
113     }
114
115     return pEnum;
116 }
117
```

Kuva 20. WMI-kyselyiden toteutus

Kuvassa 21 toteutetaan funktio, joka palauttaa käyttöjärjestelmäversion merkkijonomuuttujana. Funktio alkaa kutsumalla aiemmin esiteltyä Query-metodia, johon tehdään kysely kaikista Win32_OperatingSystem-objekteista. Paluuarvona saadun osoittimen avulla haetaan osoitin ensimmäiseen objektiin, jonka ”Version”-kentästä haluttu tieto saadaan. Jos ilmenee virhe, paluuarvona on tyhjä merkkijono.

```
111  std::string SysInfo::GetOSVersion()
112  {
113      std::string ver = "";
114      IEnumWbemClassObject* pEnum = wmi.Query("SELECT * FROM Win32_OperatingSystem");
115
116      if (pEnum == nullptr)
117      {
118          return ver;
119      }
120
121      IWbemClassObject* pClassObject = nullptr;
122      ULONG uRet = 0;
123      HRESULT hr = pEnum->Next(WBEM_INFINITE, 1, &pClassObject, &uRet);
124
125      if (uRet != 0)
126      {
127          VARIANT vtProp;
128
129          hr = pClassObject->Get(L"Version", 0, &vtProp, 0, 0);
130          if (SUCCEEDED(hr))
131          {
132              ver = Utils::wstring2string(vtProp.bstrVal);
133          }
134          VariantClear(&vtProp);
135
136          pClassObject->Release();
137      }
138
139      pEnum->Release();
140      return ver;
141  }
```

Kuva 21. Käyttöjärjestelmäversion hakeminen

Samalla tavalla kyselyitä tekemällä päästään tutkimaan monia muita tietoja järjestelmästä. Esimerkiksi Win32_Processor-objekti sisältää tietoja prosessorista, Win32_BaseBoard emolevystä, Win32_VideoController näytönohjaimesta ja Win32_PhysicalMemory järjestelmän keskusmuistista. Kirjaston poikkeuskäsittelijä kirjaa järjestelmän sarjanumeron, käyttöjärjestelmän nimen ja version, prosessoreiden ja näytönohjaimien tiedot, sekä keskusmuistin määrän lokiin.

4.1.7 Kutsupino

Kutsupinon jokainen kehys käydään läpi DbgHelp-kirjaston (ks. Debug Help Library 2018) StackWalk64-funktion (ks. StackWalk64 function 2018) avulla, jota kutsutaan silmukassa, kunnes jokainen kehys on käsitelty. Jotta kehyksistä saataisiin hyödyllistä tietoa, ladataan symbolit kutsumalla SymInitialize-funktiota. Kehystä käsitellessä haetaan symbolitiedot SymFromAddr-funktiolla, jonka jälkeen voidaan kutsua SymGetLineFromAddr64-funktiota, joka onnistuessaan palauttaa tiedostonimen, rivinumeron ja muistiosoitteen. Jos kutsu

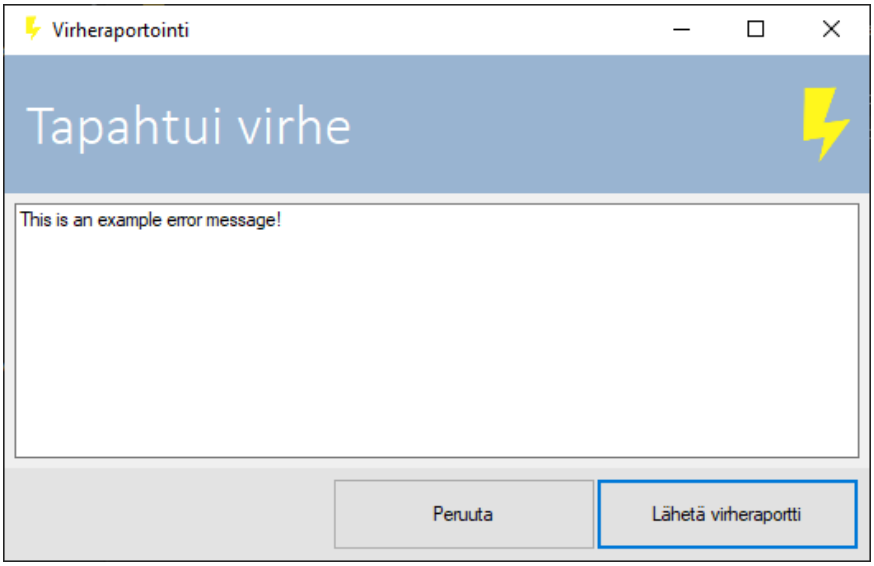
epäonnistuu, voidaan ohjelmamoduulin nimeä yrittää hakea GetModuleHandleEx- ja GetModuleFileName-funktioita käyttäen.

4.2 Raportointiohjelma

Raportointiohjelman toteutettiin C#-ohjelmointikielellä käyttäen .NET Framework -sovelluskehitysalustaa. Käyttöliittymä toteutetaan .NET Frameworkiin kuuluvan Windows Forms -kirjastoa hyödyntäen. Ohjelman tärkeimmät toteutettavat ominaisuudet ovat käyttötilat, käyttöliittymän dialogi ja raporttien lähettäminen. Tämä luku käsittelee käyttöliittymäsuunnittelua, lokalisointia, toimintojen ohjausta käynnistysparametrien avulla ja päätoimintojen, eli raporttien lähettämisen ja yhteydettömän tilan hallinnan, toteutusta.

4.2.1 Käyttöliittymä

Kuvassa 22 näkyy virheraportointiohjelman käyttöliittymä. Käyttöliittymän pääelementit ovat otsake, virheviesti ja toimintonappulat. Otsakkeen tehtävä on ilmoittaa selkeästi mistä ikkunan sisällöstä on kyse. Virheviesti tarjoaa tarkemman tiedon tapahtuman luonteesta ja viestin sisältö on asetettavissa ohjelman ulkopuolelta käynnistysparametrin avulla. Nappulat ilmaisevat käyttäjälle, mitkä ovat mahdolliset jatkotoimenpiteet. Kuvan tapauksessa käyttäjä voi sulkea ilmoituksen tai lähettää virheraportin. Raportointiohjelmaa voidaan käyttää myös pelkkänä virhesanomadialogina ilman virheraportin lähettämistä. Tällöin nappuloita olisi näkyvissä vain yksi ja sen toimintona olisi yksinkertaisesti ikkunan sulkeminen.

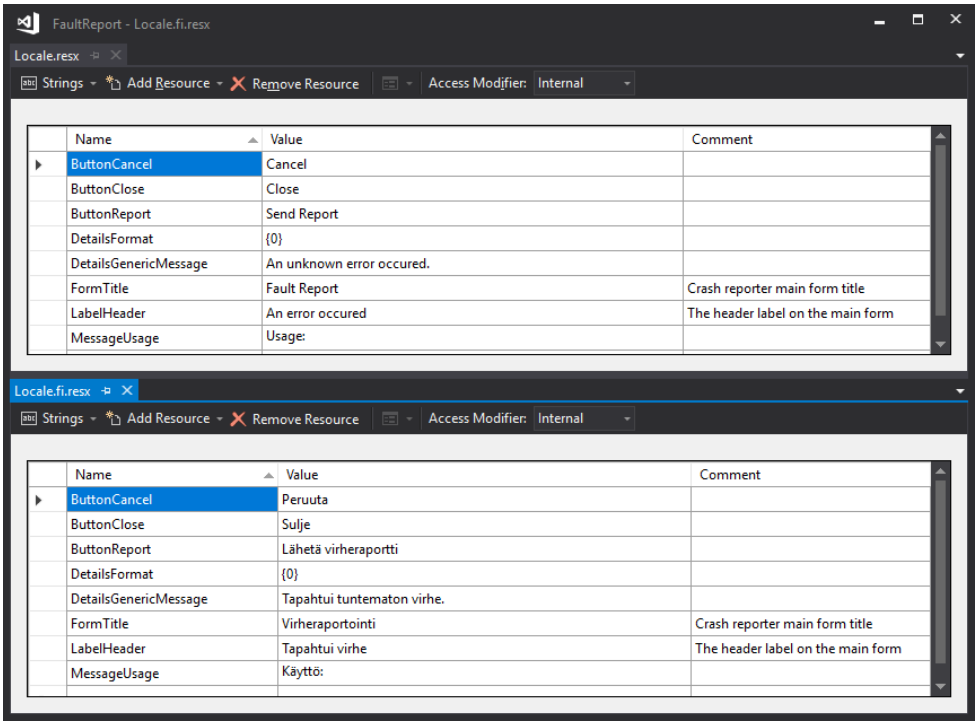


Kuva 22. Raportointiohjelman ikkuna

Jakob Nielsenin (1994) käyttöliittymäsuunnittelun kymmenen perusperiaatetta on pyritty ottamaan huomioon raportointiohjelman toteutuksessa. Tärkeimmät perusperiaatteet tämänkaltaisessa yksinkertaisessa sovelluksessa ovat olleet järjestelmän tilan selkeä ilmaiseminen ja käyttöliittymän minimalistisuus. Otsake kertoo ytimekkäästi mitä on tapahtunut ja sen yhteydessä käytettävä salamaikoni viestii tapahtuman luonteen äkillisyydestä ja odottamattomuudesta. Otsake eroaa väritykseltään muusta käyttöliittymästä, mikä ohjaa käyttäjän huomion siihen, jolloin ensimmäisellä vilkaisulla on selvää, mitä on tapahtunut. Muut käyttöliittymän osat tarjoavat vain välttämättömät toiminnot. Virheviestin sisältö määräytyy käynnistysparametrin mukana tulleesta merkkijonosta. Ideaalitilanteessa se sisältäisi tarvittavat tiedot ongelmasta ja sen korjaamisesta.

4.2.2 Lokalisointi

Käyttöliittymän lokalisointi tapahtuu lisäämällä merkkijonot resx-tiedostoihin. Jokaiselle merkkijonolle on oma avaimensa, jonka avulla lokalisoiduista resursseista haetaan käännös. Tiedostoja voi olla jokaiselle lokalisoitavalle kielelle omansa. Tekstit asetetaan käyttöliittymäelementteihin ohjelmakoodissa. Dialogin käyttöliittymäelementtien teksteistä luotiin lokalisoinnit suomeksi ja englanniksi. (Kuva 23.)

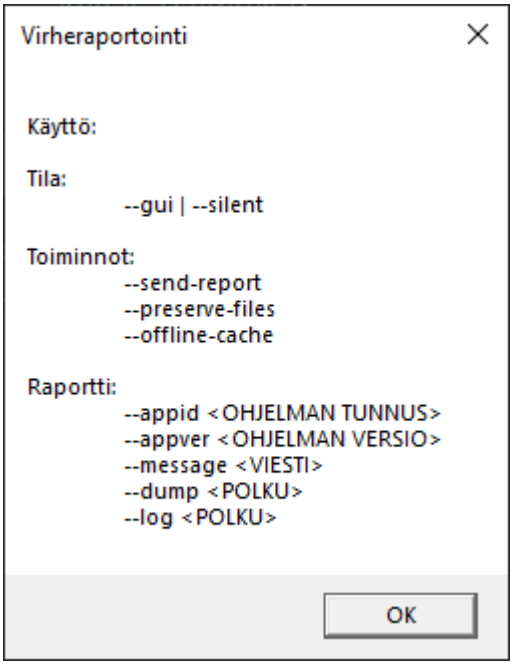


Kuva 23. Lokalisointi resurssitiedostoilla

Resx-tiedostot koostuvat XML-elementeistä (ks. W3Schools s.a.) ja ne voivat sisältää merkkijonojen lisäksi myös binääridataa, kuten kuvia ja ikoneja. Lokalisoitavien resurssitiedostojen nimen päätteeksi lisätään kielikoodi ennen tiedostopäätettä ja ohjelman käännösvaiheessa niistä luodaan omat käännöstiedostonsa. (Create resource files for .NET apps 2017.)

4.2.3 Käynnistysparametrit

Käynnistysparametreilla ohjataan ohjelman toimintoja ja välitetään argumentteina käsiteltävää tietoa. Raportointiohjelman parametreina ilmaistaan haluttu käyttötila, käytettävät lisätoiminnot ja raportin koostavat tiedot. Tässä luvussa esitellään ohjelman käyttämät parametrit ja esimerkkejä niiden käytöstä.



Kuva 24. Raportointiohjelman käynnistysparametrit

Mikäli ohjelma käynnistetään ilman parametreja, näytetään kuvan 24 mukainen viesti-ikkuna. Viestissä näytetään ryhmitellysti parametrit ja niihin odotetut argumentit. Ohjelman käyttötila valitaan joko ”gui”- tai ”silent”-parametrilla, jotka ohjaavat sitä, näytetäänkö dialogi-ikkuna vai ei. Toisin sanoen, edellytetäänkö käyttäjältä toimenpiteitä vai suoritetaanko ohjelma tausta-ajona.

Päätilojen lisäksi ohjelma lisätoimintoja ohjaavia parametreja on kolme, ja ne eivät ole pakollisia. ”Send-report”-parametri ilmaisee, että raportti on tarkoitus lähettää. Ilman sitä käyttöliittymätilassa näytetään vain virheviesti ja oletuksena ohjelma toimii tiedostojen siivoajana, paitsi silloin, jos ”preserve-files”-parametri on käytössä. Nimensä mukaisesti ”preserve-files” ohittaa tiedostojen siivoamisen tallennustilasta. Viimeinen parametri ”offline-cache” ottaa käyttöön tiedostojen säilyttämisen yhteydettömässä tilassa. Raportit, joiden lähettäminen on epäonnistunut, säilytetään, kunnes niiden lähettäminen onnistuu.

Raportin tiedot välitetään viimeisen parametriryhmän argumentteina. Raportti koostetaan ohjelman tunnuksesta, versiosta, virheviestistä ja mahdollisista loki- ja vedostiedostoista, joiden tiedostopolut ohjelmalle välitetään. Raportille pakollinen parametri on ”appid”, jotta raportti voidaan yhdistää tiettyyn ohjelmaan.

```
Rem Näyttää virheilmoituksen, ei lähetä virheraporttia.
start FaultReport.exe --gui --appid "test" --message "This is an example error message!"

Rem Yrittää odottavien raporttien uudelleenlähetystä taustalla.
start FaultReport.exe --silent --offline-cache --send-report

Rem Näyttää virheilmoituksen, lähettää raportin ja ei siivoa loki- ja vedostiedostoja.
start FaultReport.exe --gui --appid "test" --dump "dumps\vedos.dmp" --send-report --preserve-files

Rem Lähettää virheraportin taustalla. Jos lähetys epäonnistuu, raportti tallennetaan uudelleenlähetystä varten.
start FaultReport.exe --silent --appid "test" --message "Error message" --send-report --offline-cache
```

Kuva 25. Esimerkkejä raportointiohjelman käytöstä

Kuvassa 25 on esimerkkikomentoja, joilla raportointiohjelmaa voidaan käyttää. Ensimmäinen komento on esimerkki tilanteesta, kun ohjelmaa halutaan käyttää vain ilmoitusdialogina tapahtuneesta virheestä ilman raportin lähettämistä. Toinen komento on hyödyllinen silloin, kun halutaan yrittää yhteydettömän tilan aikana mahdollisesti luotujen raporttien lähetystä uudelleen. Kolmas komento näyttää virhedialogin ja tarjoaa raportin lähetystä. Raportin tiedostoja ei siivota, eikä epäonnistunutta lähetystä yritetä uudelleen. Viimeinen komennoista yrittää virheraportin lähettämistä taustalla ja säilyttää yhteydettömässä tilassa raporttia lähetettäväksi.

4.2.4 Raportin lähettäminen

Raportin lähettämiseksi on luotava http-asiakas, jolla pyynnöt lähetetään rajapinnalle. Pyynnön viesti muodostetaan MultiPartFormDataContent-objektin avulla, jonka kenttiin lisätään raportin tiedot merkkijonoina, ja johon mukana lähetettävät tiedostot liitetään. Viestin muodostamisen päätteeksi otsakkeeseen saatetaan liittää tarvittavia lisätietoja. Tässä tapauksessa rajapinnan vaatima avain lisätään otsakkeen ”x-functions-key”-kenttään. Viesti lähetetään rajapinnan osoitteeseen POST-pyyntönä. Rajapinta vastaa pyyntöön tilakoodilla, joka ilmaisee, käsiteltiinkö pyyntö onnistuneesti. (Kuva 26.)

```
13          public static async Task<bool> Send(Report report)
14          {
15              using (var client = new HttpClient())
16              {
17                  var form = new MultipartFormDataContent
18                  {
19                      { new StringContent(report.AppId), "appId" },
20                      { new StringContent(report.Created.ToUniversalTime().ToString("o")), "created" },
21                      { new StringContent(report.AppVersion ?? ""), "appVersion" },
22                      { new StringContent(report.Message ?? ""), "message" }
23                  };
24
25                  AddFileToForm(form, report.DumpPath, "dump");
26                  AddFileToForm(form, report.LogPath, "log");
27
28                  form.Headers.Add("x-functions-key", apiKey);
29
30                  try
31                  {
32                      var response = await client.PostAsync(apiUrl, form);
33                      return response.IsSuccessStatusCode;
34                  }
35                  catch(Exception ex)
36                  {
37                      Console.WriteLine(ex.Message);
38                  }
39
40                  return false;
41              }
42          }
```

Kuva 26. Raportin lähettäjä

On mahdollista, että raportin lähettäminen epäonnistuu verkko-ongelmien tai palvelinvirheen takia. Tällöin vastauksen tilakoodi ilmaisee pyynnön epäonnistumisen tai http-asiakas heittää poikkeuksen. Näissä tilanteissa, mikäli käynnistysparametrina on ollut ”offline-cache”, voidaan toimia yhteydettömän tilan mukaisesti.

4.2.5 Yhteydetön tila

Yhteydettömässä tilassa raportin tiedot tallennetaan myöhempää lähetysyritystä varten. Yhteydetön tila on toiminto, joka otetaan käyttöön käynnistämällä ohjelma ”offline-cache”-parametrilla yhdessä ”send-report”-parametrin kanssa. Toiminnon ollessa käytössä, luodaan XML-tiedosto ”FaultReport.pending”, johon epäonnistuneesti lähetettyjen raporttien tiedot kirjataan. Samalla ohjelma lukee tiedostosta jonossa olevat raportit ja yrittää niiden lähetystä uudelleen.

```xml
<?xml version="1.0" encoding="utf-8"?>
<reports>
    <report>
        <appid>test</appid>
        <created>2019-04-12T18:54:44.0617018+03:00</created>
        <appver>1.0</appver>
        <message>An exception c0000094 was thrown at 00007FF736EA14A7.</message>
        <dump>test.exe_201904121554_b7d42a555c16e485.dmp</dump>
        <log>test.exe_201904121554_b7d42a555c16e485.log</log>
    </report>
</reports>
```

Kuva 27. Raportin tiedot tallennettuna

Yhteydettömän tilan luoma tiedosto sisältää XML-elementin ”reports”, johon luodut ”report”-elementit sisältävät yksittäisten raporttien tiedot. Raportin elementti sisältää elementit kaikille raportin parametreille. Raporttiin lisätään ”created”-elementti, joka sisältää aikaleiman siitä ajanhetkestä, kun raportti luotiin. Kuvassa 27 on esimerkki tiedoston sisällöstä, joka sisältää yhden raportin ”test”-tunnukselliselle ohjelmalle. Kirjaston luomat virhelokin ja kaatumisvedoksen polut on välitetty ohjelmalle, jotta tiedostot voidaan myöhemmin liittää osaksi raporttia.

```
// Lukeminen XML-tiedostosta
var xml = XElement.Load(fileName);
var reports = xml.Descendants("report");
foreach (var item in reports)
{
    var report = new Report
    {
        AppId = item.Element("appid").Value,
        Created = DateTime.Parse(item.Element("created").Value),
        AppVersion = item.Element("appver").Value,
        Message = item.Element("message").Value,
        DumpPath = item.Element("dump").Value,
        LogPath = item.Element("log").Value
    };

    list.Add(report);
}

// Tallentaminen XML-tiedostoon
using (var writer = XmlWriter.Create(fileName))
{
    var xml = new XElement("reports");
    foreach (var report in reports)
    {
        xml.Add(new XElement("report",
            new XElement("appid", report.AppId),
            new XElement("created", report.Created.ToString("o")),
            new XElement("appver", report.AppVersion ?? ""),
            new XElement("message", report.Message ?? ""),
            new XElement("dump", report.DumpPath ?? ""),
            new XElement("log", report.LogPath ?? "")));
    }
    xml.WriteTo(writer);
    writer.Close();
}
```

Kuva 28. Lukeminen ja kirjoittaminen XML-tiedostoon

Kuvassa 28 esitellään XML-tiedoston käsittelyä ohjelmakoodissa. Tiedostojen lukeminen ja kirjoittaminen on mutkatonta .NET-luokkakirjaston tarjoamilla metodeilla. Raportin tiedot muunnetaan merkkijonoiksi ja tallennetaan XML-elementeiksi. Elementtien merkkijonot voidaan luettaessa jäsentää takaisin oikeaan tyyppiin, kuten DateTime-arvotyypiksi.

4.3 Rajapinta

Rajapinta noudattaa REST-arkkitehtuurityyliä (ks. Representational state transfer s.a.), sillä se on resurssikeskeinen, eikä kyselyihin liity tilan käsittelyä. Rajapinnasta vastaa Azure Functions -palvelussa suoritettava ohjelmakoodi, joka on esitelty kuvassa 29. Metodi suoritetaan, kun rajapintaan lähetetään POST-pyyntö ”/v1/report”-polkuun. Pyyntö ohjataan metodin req-parametriin. Metodin muihin parametreihin on sidottu tarvittavat Table storage ja Blob storage -resurssit. Metodin paluuarvona on HttpResponseMessage-tyyppinen objekti, joka sisältää vastauksen rajapinnan kutsujalle.

```
78      public static class Report_v1
79      {
80          [FunctionName("Report_v1")]
81          public static async Task<HttpResponseMessage> Run(
82              [HttpTrigger(AuthorizationLevel.Function, "post", Route = "v1/report")] HttpRequestMessage req,
83              [Table("Report")] CloudTable reportTable,
84              [Blob("dumps")] CloudBlobContainer dumps,
85              [Blob("logs")] CloudBlobContainer logs,
86              TraceWriter log)···
144
145         private static async Task<bool> UploadBlob(CloudBlobContainer target, string name, Stream file)···
157     }
```

Kuva 29. Rajapinnan metodi

Funktio käyttää autentikointiin esimerkkikoodissa yksinkertaista funktiokohtaista avainta, joka liitetään kyselyn otsakkeeseen. Jatkokehitystä ajatellen autentikointi voidaan toteuttaa vahvempia keinoja hyödyntäen, kuten esimerkiksi HMAC-tiivistettä (ks. HMAC s.a.) käyttäen. Lisäksi funktioon voidaan toteuttaa GET-kyselyiden käsittely, joka palauttaisi ohjelmatunnisteella listan vastaanotetuista raporteista ja vastaavasti raporttitunnisteella vastauksena olisi raportista kerätyt tiedot.

4.3.1 Vastaanotto

Raportin vastaanottaminen alkaa lukemalla vastaanotetun pyynnön HttpRequestMessage-objektin viestirunko, joka on HttpContent-tyyppisestä Content-jäsenestä. Tiedetään, että vastaanotettu raportti on multipart-muodossa oleva lomake, jolloin lukeminen tapahtuu ReadAsMultipartAsync-metodilla. Saadusta MultiPartMemoryStreamProvider-objektista, joka annetaan ReportContent-objektin parsittavaksi, voidaan lukea lomakkeen tiedot. Mikäli lomakkeesta ei saada selville raporttiin liittyvän ohjelman tunnusta, vastataan paluukoodilla ”BadRequest”, joka ilmaisee, ettei pyyntöä käsitelty virheellisyytensä vuoksi. (Kuva 30.)

```
88          var form = await req.Content.ReadAsMultipartAsync();
89
90          var content = new ReportContent();
91          await content.ParseContent(form);
92
93          if (string.IsNullOrEmpty(content.AppId))
94          {
95              return req.CreateResponse(HttpStatusCode.BadRequest);
96          }
```

Kuva 30. Raportin vastaanottaminen

Kuvassa 31 on esitetty ReportContent-luokka. Luokalla on jäseninä nk. ominaisuuksia, joita voidaan hyödyntää eri tavoin, kuten kentän arvon validoimiseen tai pääsyn määrittelyyn. Ominaisuudet ovat esitelty siten, että niihin sisältyy kenttä, joiden arvoihin pääsy on määritelty siten, että arvot ovat luettavissa luokan ulkopuolelta, mutta asetettavissa vain sisäpuolelta. Ominaisuuksien lisäksi luokalla on ParseContent-metodi, jonka tehtävä on lukea multipart-lomakkeesta arvot luokan kenttiin.

```
27  public class ReportContent
28  {
29      public string AppId { get; private set; } = null;
30      public string AppVersion { get; private set; } = null;
31      public string Message { get; private set; } = null;
32      public DateTime? Created { get; private set; } = null;
33      public Stream DumpStream { get; private set; } = null;
34      public string DumpName { get; private set; } = null;
35      public Stream LogStream { get; private set; } = null;
36      public string LogName { get; private set; } = null;
37
38      public async Task ParseContent(MultipartMemoryStreamProvider multipart)···
76  }
```

Kuva 31. ReportContent-luokka

ParseContent-metodi käy multipart-lomakkeen jokaisen kentän läpi, ja vertaa kenttien nimiä luokan jäseniin. Jos jäsentä vastaava kenttä löytyy, luetaan kentän tieto sellaisenaan tai muunnetaan oikeaan muotoon. Tiedostojen kohdalla parsitaan tiedostonimi ja luetaan tiedoston data Stream-objektiin, josta se voidaan seuraavaksi tallentaa.

4.3.2 Tallentaminen

Kun raportin sisältö on luettu, tallentaminen aloitetaan liitetiedostoista. Raportille luodaan yksilöivä tunniste, joka liitetään tiedostojen nimiin. Tiedostot ladataan sen jälkeen CloudBlobContainer-säilöihin, jota varten luokkaan toteutetaan metodi nimeltä UploadBlob. UploadBlob-metodille annetaan parametreina säilöön viittaava objekti, ladattavan tiedoston nimi ja tiedostodatan sisältämä Stream-objekti. Mikäli jonkin tiedoston lataus epäonnistuu, voidaan olettaa, että rajapinnan määritykset eivät ole kunnossa, jolloin raportin käsittely on lopetettava. Palvelinvirheestä ilmaiseva http-tilakoodi on InternalServerError. (Kuva 32.)

```
98          var guid = Guid.NewGuid();
99          string blobDump = null;
100         string blobLog = null;
101
102         if (content.DumpStream != null)
103         {
104             blobDump = $"{guid.ToString()}_{content.DumpName}";
105             if (await UploadBlob(dumps, blobDump, content.DumpStream) == false)
106             {
107                 log.Error($"Failed to upload blob {blobDump}!");
108                 return req.CreateResponse(HttpStatusCode.InternalServerError);
109             }
110         }
111
112         if (content.LogStream != null)
113         {
114             blobLog = $"{guid.ToString()}_{content.LogName}";
115             if (await UploadBlob(logs, blobLog, content.LogStream) == false)
116             {
117                 log.Error($"Failed to upload blob {blobLog}!");
118                 return req.CreateResponse(HttpStatusCode.InternalServerError);
119             }
120         }
```

Kuva 32. Vedoksen ja lokitiedoston tallentaminen

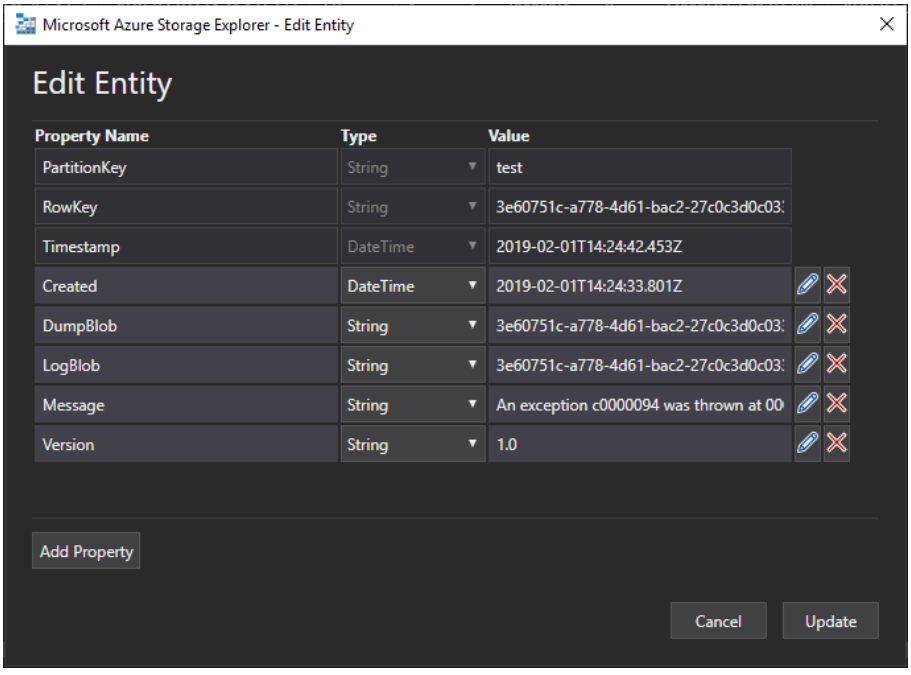
Liitetiedostojen tallentamisen jälkeen raportille luodaan entiteetti, eli rivi tai tietue, josta ilmenee virheen tiedot ja tallennettujen tiedostojen nimet. Table storagen entiteeteillä on kaksi avainta: osioavain ja riviavain. Riviavain toimii yksilöivänä tietona, kun taas osioavain voidaan ajatella tietyn entiteettijoukon

tunnisteena. Tässä tapauksessa riviavaimena käytetään raportin yksilöivää tunnistetta ja osioavaimena on käytetty ohjelmatunnistetta. (Kuva 33.)

```
122             var entity = new ReportEntity
123             {
124                 PartitionKey = content.AppId,
125                 RowKey = guid.ToString(),
126                 Message = content.Message,
127                 Version = content.AppVersion,
128                 Created = content.Created,
129                 DumpBlob = blobDump,
130                 LogBlob = blobLog
131             };
132
133             var insert = TableOperation.Insert(entity);
134             TableResult result = await reportTable.ExecuteAsync(insert);
135
136             if (result.HttpStatusCode != 204)
137             {
138                 log.Error($"Failed to create report! ({result.HttpStatusCode})");
139                 return req.CreateResponse(HttpStatusCode.InternalServerError);
140             }
141
142             return req.CreateResponse(HttpStatusCode.OK);
143         }
```

Kuva 33. Raportin tallentaminen entiteetiksi

Azure Storage Explorerilla voidaan nähdä tauluun syntynyt entiteetti. Kuvassa 34 ohjelmalla on avattu testiohjelman tuottama virheraportti. Raportista ilmenee raportin yksilöllisen tunnisteen lisäksi ohjelmatunniste ja -versio, millä nimillä tallennetut loki- ja vedostiedostot löytyvät, ohjelman tuottama virheviesti ja milloin raportti sekä entiteetti on luotu.



Kuva 34. Raportin entiteetti Storage Explorerilla avattuna

Kun raportin tiedot on tallennettu, rajapinta palauttaa vastauksen OK, ilmaisten raportin käsittelyn päättyneen onnistuneesti. Taulusta voidaan nyt hakea raportteja, kuten mistä tahansa tietokannasta. Raporttiin sidotut tiedostot on lisätty omiin säilöihinsä, jossa ne ovat valmiina hyödynnettäviksi. Seuraavaksi toteutetaan testiohjelma järjestelmän toiminnan tarkistamiseksi.

4.4 Testiohjelma

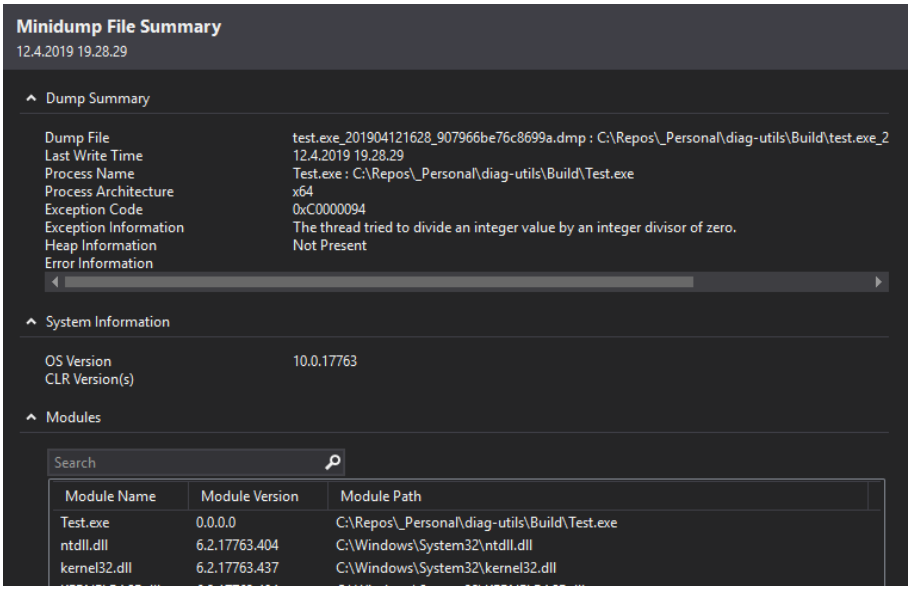
Järjestelmän testausta varten luodaan testiohjelma, joka hyödyntää kirjastoa. Ohjelman tehtävä on aiheuttaa tahallisesti poikkeus, josta kirjaston poikkeuskäsittelijä luo raportin. Liitteessä 1 on testiohjelman lähdekoodi, joka muodostuu kolmesta funktiosta, jotka ovat ”main”, ”callback” ja ”faulty”. Ensimmäisenä mainittu on ohjelman pääfunktio, eli ensimmäinen suoritettava funktio ohjelman käynnistyksestä. Toinen on funktio, jota kutsutaan poikkeuskäsittelijästä, ja joka asettaa virheviestin raportointiohjelmalle. Ohjelman kaatava funktio on ”faulty”, joka aiheuttaa poikkeuksen yrittämällä jakolaskua nollalla.

Ohjelman ”main”-funktiossa luodaan ensimmäisenä määritykset kirjaston alustusta varten. Raportointiohjelma halutaan käynnistyvän graafisena käyttöliittymänä ja muissa määrityksissä käytetään oletuksia, pois lukien callback-parametrin, jonka argumentiksi asetetaan aiemmin mainittu callback-funktio. Määrityksien täyttämisen jälkeen kirjasto alustetaan ja säädetään lokin asetukset. Jotta kaikki tieto saadaan näkyville, lokitasoksi asetetaan ”Trace” ja lisäksi lokiviestit kirjoitetaan erilliseen tiedostoon ja konsoli-ikkunaan. Kun kaikki on valmista, kaadetaan ohjelma kutsumalla faulty-funktiota. (Liite 1)

```
[2019-04-12 18:22:51.303] [INFO] (main.cpp:65) FaultLib Initialized
[2019-04-12 18:22:51.304] [INFO] (main.cpp:14) Begin crashing!
[2019-04-12 18:22:51.306] [ERROR] (handler.cpp:9) Fatal exception: c0000094
=== System Information ===
S/N               : ***********
OS Name           : Microsoft Windows 10 Education (64-bittinen)
OS Version        : 10.0.17763
CPU0              : Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz
RAM               : 16384 MB
VideoController1  : Intel(R) HD Graphics 520
=== End of System Information ===
=== Stack Trace ===
    at main in c:\repos\_personal\diag-utils\test\main.cpp : line: 67: address:
    0x7ff6036a1586
    at __scrt_common_main_seh in
    d:\agent\_work\1\s\src\vctools\crt\vcstartup\src\startup\exe_common.inl : line:
    120: address: 0x7ff6036a25f2
    at BaseThreadInitThunk, address: 0x250b7cfc208 in C:\WINDOWS\System32\KERNEL32.DLL
    at RtlUserThreadStart, address: 0x6b006e0075 in C:\WINDOWS\SYSTEM32\ntdll.dll
=== End of Stack Trace ===
```

Kuva 35. Virheloki

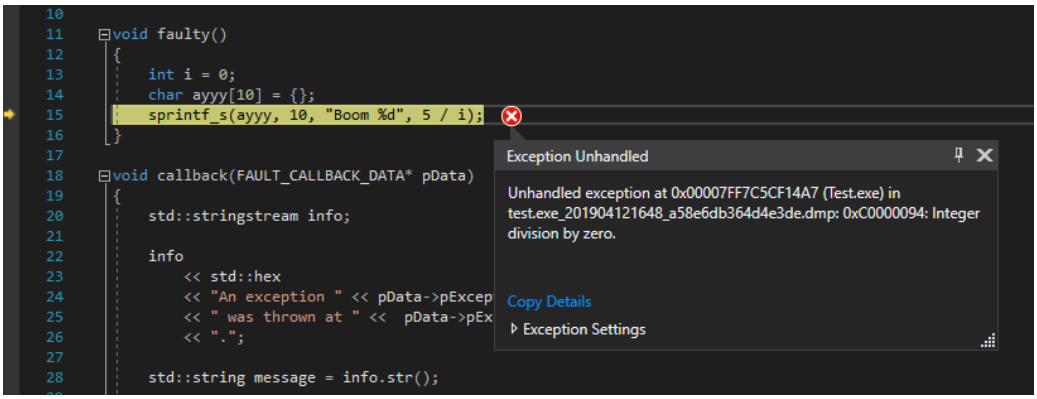
Ohjelma kaatuu ja poikkeuskäsittelijä tekee tehtävänsä. Virheloki sisältää lokiviestien lisäksi järjestelmän ja kutsupinon tiedot. Kuvassa 35 on esimerkki virhelokin sisällöstä, josta ilmenee tapahtumien ajankohta ja poikkeuksen aiheuttaneen lähdekoodin sijainti – ainakin summittaisesti. Lokin tuottanut ohjelma on käännetty release-konfiguraatiolla, joten siitä ei ilmene yhtä suurella tarkkuudella tapahtumien kulkua, kuin debug-tilassa käännetyn ohjelman lokista.



Kuva 36. Yhteenveto kaatumisvedoksesta

Virhelokin lisäksi virheenetsinnän avuksi on kirjoitettu kaatumisvedos. Kaatumisvedos voidaan analysoida erillisellä ohjelmalla. Visual Studiolla avatun

kaatumisvedoksen yhteenvedosta nähdään, että kyseessä on nollalla jakamisesta aiheutunut poikkeus. Yhteenvedosta nähdään myös ohjelman suoritukseen liittyneet ohjelmamoduulit. (Kuva 36.)



Kuva 37. Poikkeuksen aiheuttanut koodirivi

Kaatumisvedoksen avulla voidaan simuloida ohjelman tilaa poikkeuksen hetkellä, mikä helpottaa ongelman paikallistamisessa. Visual Studiossa vedoksen analysoiminen on helppoa interaktiivisten työkalujen avulla. Kun lähdekoodi on saatavilla, korostaa Visual Studio ongelmallisen koodirivin. (Kuva 37.)

5 TULOKSET JA POHDINTA

Kehitystyön tarkoituksena oli suunnitella ja toteuttaa virheraportointijärjestelmä. Lopputuloksena syntyi ohjelmistokokonaisuus, joka hyödyntää yrityksen jo käytössä olevia palveluja, ja samalla tarjoaa hyvän pohjan järjestelmän jatkokehitykselle. Lisäksi syntyi runsaasti hyödyllistä ohjelmakoodia, jota voidaan hyödyntää järjestelmän ulkopuolella samankaltaisten toimintojen toteutukseen. Järjestelmän perustoimintojen kehittämiseen meni verrattain vähän aikaa – karkeasti arvioiden noin 9 henkilötyöpäivää. Nykyisellään toteutus kuitenkin jättää paljon tilaa parannuksille.

Toteutuksen aikana ilmeni suunnitelmassa ja käytetyissä mekanismeissa puutteita, joista nousi esiin ideoita jatkokehitykselle. Ohjelmakirjaston jatkokehitysideoita ovat poikkeuksien käsitteleminen VEH-funktiolla, erillisen säikeen käyttäminen poikkeuksien käsittelyssä ja tietojen liittäminen suoraan osaksi vedosta. Raportointiohjelman toimintoja voidaan monipuolistaa lisäämällä käyttäjälle mahdollisuus kuvailla tapahtumia, jotka johtivat ohjelman kaatumiseen; luotettavuutta parantaa hyödyntämällä tarkistussummia, joilla voidaan

havaita vioittuneet tiedostot; raporttien hallintaa yksinkertaistaa pakkaamalla kaikki tiedostot yhteen arkistoon. Rajapinta lienee järjestelmän yksinkertaisin osa, mutta omaa liiketoiminnan ja kehitystyön kannalta suurimman potentiaalin: raporteista voidaan muodostaa tilastoja ohjelmien luotettavuuden ja laadun mittaamiseen; vastaanotetuista raporteista voidaan lähettää sähköposti-ilmoituksia esimerkiksi tiettyjen kriteereiden täyttyessä; rajapintaa voidaan laajentaa raporttien hakemiseen, suodattamiseen ja analysointiin. Yleisiä parannusehdotuksia – koskien koko järjestelmää – ovat alustatuen laajentaminen Linux-järjestelmiin ja integraatio osaksi CI/CD-järjestelmää.

Verrattuna olemassa oleviin ratkaisuihin, kuten CrashRpt ja Google Breakpad, järjestelmä ei tarjoa varsinaisesti mitään uutta, vaan on nykytilassa ominaisuuksiltaan paljon karsitumpi, mikä ei välttämättä ole huono asia. Yhteensopivuus yrityksen tuotteiden kanssa on luonnostaan parempi, sillä kehitystyön lähtökohtana on ollut käyttää samoja työkaluja ja teknologioita kuin yrityksen tuotteet. Verrattuna Google Breakpadiin, järjestelmän merkittävä heikkous on monialustatuen puute, mikä ei yrityksen tuotteiden kohdalla ole tällä hetkellä ongelma, mutta voi tulevaisuudessa sitoa resursseja.

Toimeksiantajalle ehdotan järjestelmän käyttöönottoa ensin yhdessä tuotteessa kokeeksi – tai kenties aluksi vain kehitysympäristössä – jotta käyttökokemuksia voidaan kerätä ja järjestelmän mahdolliset heikkoudet ja parannuskohteet saadaan selville. Näiden tietojen perusteella järjestelmää voidaan kehittää vastaamaan paremmin yrityksen tarpeita ja parantaa sen tuotantokelpoisuutta. Myöhemmässä vaiheessa järjestelmän käyttöönottoa voidaan laajentaa tarkoituksenmukaisesti muihin tuotteisiin, kunhan järjestelmän jatkokehitykselle on käytössä tarpeeksi resursseja.

LÄHTEET

About WER. 2018. Microsoft. WWW-dokumentti. Päivitetty 31.5.2018. Saatavissa: https://docs.microsoft.com/en-us/windows/desktop/wer/about-wer [viitattu 16.4.2019].

About WMI. 2018. Microsoft. WWW-dokumentti. Päivitetty 31.5.2018. Saatavissa: https://docs.microsoft.com/en-us/windows/desktop/wmisdk/about-wmi [viitattu 1.4.2019].

An introduction to Azure Functions. 2017. Microsoft. WWW-dokumentti. Päivitetty 3.10.2017. Saatavissa: https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview [viitattu 10.3.2019].

Azure Table storage overview. 2017. Microsoft. WWW-dokumentti. Päivitetty 3.11.2018. Saatavissa: https://docs.microsoft.com/en-us/azure/cosmos-db/table-storage-overview [viitattu 10.3.2019].

Call stack s.a. Wikipedia. Microsoft. WWW-dokumentti. Päivitetty 16.4.2019. Saatavissa: https://en.wikipedia.org/wiki/Call_stack [viitattu 16.4.2019].

Common Information Model s.a. DMTF. WWW-dokumentti. Saatavissa: https://www.dmtf.org/standards/cim [viitattu 30.3.2019].

Crash Dump Analysis. 2018. Microsoft. WWW-dokumentti. Päivitetty 31.5.2018. Saatavissa: https://docs.microsoft.com/en-us/windows/desktop/dxtecharts/crash-dump-analysis [viitattu 1.4.2019].

CrashRpt. 2015a. Architecture Overview. WWW-dokumentti. Päivitetty 29.4.2015. Saatavissa: http://crashrpt.sourceforge.net/docs/html/architecture_overview.html [viitattu 8.3.2019].

CrashRpt. 2015b. Getting Started. WWW-dokumentti. Päivitetty 29.4.2015. Saatavissa: http://crashrpt.sourceforge.net/docs/html/getting_started.html [viitattu 9.3.2019].

Create resource files for .NET apps. 2017. Microsoft. WWW-dokumentti. Päivitetty 30.3.2017. Saatavissa: https://docs.microsoft.com/en-us/dotnet/framework/resources/creating-resource-files-for-desktop-apps [viitattu 8.4.2019].

Debug Help Library. 2018. Microsoft. WWW-dokumentti. Päivitetty 31.5.2018. Saatavissa: https://docs.microsoft.com/en-us/windows/desktop/debug/debug-help-library [viitattu 12.4.2019].

Dynamic-Link Libraries. 2018. Microsoft. WWW-dokumentti. Päivitetty 31.5.2018. Saatavissa: https://docs.microsoft.com/en-us/windows/desktop/dlls/dynamic-link-libraries [viitattu 10.3.2019].

Effective Minidumps. 2005. DebugInfo. WWW-dokumentti. Päivitetty 7.2.2005. Saatavissa: http://www.debuginfo.com/articles/effminidumps.html [viitattu 8.4.2019].

EXCEPTION_POINTERS structure. 2018. Microsoft. WWW-dokumentti. Päivitetty 5.12.2018. Saatavissa: https://docs.microsoft.com/en-us/windows/desktop/api/winnt/ns-winnt-_exception_pointers [viitattu 12.4.2019].

Get started guide for Azure developers. 2017. Microsoft. WWW-dokumentti. Päivitetty 18.10.2017. Saatavissa: https://docs.microsoft.com/en-us/azure/guides/developer/azure-developer-guide [viitattu 9.3.2019].

Get started with Storage Explorer. 2017. Microsoft. WWW-dokumentti. Päivitetty 17.7.2017. Saatavissa: https://docs.microsoft.com/en-us/azure/vs-azure-tools-storage-manage-with-storage-explorer?tabs=windows [viitattu 9.3.2019].

Google Breakpad. 2016. Google. WWW-dokumentti. Päivitetty 18.11.2016. Saatavissa: https://github.com/google/breakpad/blob/master/docs/getting_started_with_breakpad.md [viitattu 9.3.2019].

Gregoire, M., Solter, N. & Kleper, S. 2011. Professional C++. 2. painos. Indianapolis: Wiley.

HMAC s.a. Wikipedia. WWW-dokumentti. Päivitetty 7.4.2019. Saatavissa: https://en.wikipedia.org/wiki/HMAC [viitattu 16.4.2019].

Introduction to Azure Blob storage. 2019. Microsoft. WWW-dokumentti. Päivitetty 3.1.2019. Saatavissa: https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction [viitattu 10.3.2019].

Johnston, P. 2018. The Dark Side of Error Logging. WWW-dokumentti. Saatavissa: https://arne-mertz.de/2018/03/the-dark-side-of-error-logging/ [viitattu 1.4.2019].

Karpov, A. 2008. Building of systems of automatic C/C++ code logging, WWW-dokumentti. Saatavissa: https://www.viva64.com/en/a/0023/ [viitattu 9.3.2019].

Kasurinen, J-P. 2013. Ohjelmistotestauksen käsikirja. 1. painos. Jyväskylä: Docendo.

Kilpeläinen, V. 2018. Kaatumisilmoitusten kerääminen ja analysoinnin automatisointi. Tampereen Ammattikorkeakoulu. Tieto- ja viestintätekniikka. Opinnäytetyö. Saatavissa: http://urn.fi/URN:NBN:fi:amk-201804245321 [viitattu 9.3.2019].

Minidump Files. 2018. Microsoft. WWW-dokumentti. Päivitetty 31.5.2018. Saatavissa: https://docs.microsoft.com/en-us/windows/desktop/debug/minidump-files [viitattu 1.4.2019].

MiniDumpWriteDump function. 2018. Microsoft. WWW-dokumentti. Päivitetty 5.12.2018. Saatavissa: https://docs.microsoft.com/en-us/windows/desktop/api/minidumpapiset/nf-minidumpapiset-minidumpwritedump [viitattu 16.4.2019].

NET Framework Guide. 2018. Microsoft. WWW-dokumentti. Päivitetty 10.4.2018. Saatavissa: https://docs.microsoft.com/fi-fi/dotnet/framework/ [viitattu 9.3.2019].

Nielsen, J. 1994. 10 Usability Heuristics for User Interface Design. WWW-dokumentti. Päivitetty 24.4.1994. Saatavissa: https://www.nngroup.com/articles/ten-usability-heuristics/ [viitattu 10.3.2019].

Peteronprogramming. 2016a. Crashes you can't handle easily #1: SEH failure on x64 Windows. Blogikirjoitus. Päivitetty 29.5.2016. Saatavissa: https://peteronprogramming.wordpress.com/2016/05/29/crashes-you-cant-handle-easily-1-seh-failure-on-x64-windows/ [viitattu 6.4.2019].

Peteronprogramming. 2016b. Crashes you can't handle easily #2: Stack overflows on Windows. Blogikirjoitus. Päivitetty 10.8.2016. Saatavissa: https://peteronprogramming.wordpress.com/2016/08/10/crashes-you-cant-handle-easily-2-stack-overflows-on-windows/ [viitattu 6.4.2019].

Peteronprogramming. 2017. Crashes you can't handle easily #3: STATUS_HEAP_CORRUPTION on Windows. Blogikirjoitus. Päivitetty 30.7.2017. Saatavissa: https://peteronprogramming.wordpress.com/2017/07/30/crashes-you-cant-handle-easily-3-status_heap_corruption-on-windows/ [viitattu 6.4.2019].

Representational state transfer s.a. Wikipedia. WWW-dokumentti. Päivitetty 17.4.2019. Saatavissa: https://en.wikipedia.org/wiki/Representational_state_transfer [viitattu 19.4.2019].

RFC 7231. 2014. HyperText Transfer Protocol (HTTP/1.1): Semantics and Content. Osa 3.1.1.4: Multipart Types.

SetUnhandledExceptionFilter function s.a. Microsoft. Saatavissa: https://msdn.microsoft.com/en-us/library/windows/desktop/ms680634(v=vs.85).aspx [viitattu 6.4.2019].

Stack trace s.a. Wikipedia. WWW-dokumentti. Päivitetty 15.1.2019. Saatavissa: https://en.wikipedia.org/wiki/Stack_trace [viitattu 1.4.2019].

StackWalk64 function. 2018. Microsoft. WWW-dokumentti. Päivitetty 5.12.2018. Saatavissa: https://docs.microsoft.com/en-us/windows/desktop/api/dbghelp/nf-dbghelp-stackwalk64 [viitattu 12.4.2019].

Structured Exception Handling. 2018. Microsoft. WWW-dokumentti. Päivitetty 31.5.2018. Saatavissa: https://docs.microsoft.com/en-us/windows/desktop/debug/structured-exception-handling [viitattu 11.3.2019].

Telles, M. & Hsieh, Y. 2001. The Science of Debugging. E-kirja. Saatavissa: https://ebookcentral.proquest.com/lib/xamk-ebooks/detail.action?docID=3384706 [viitattu 1.4.2019].

The 3-Clause BSD License s.a. Open Source Initiative. WWW-dokumentti. Saatavissa: https://opensource.org/licenses/BSD-3-Clause [viitattu 9.3.2019].

Toikko, T. & Rantanen, T. 2009. Tutkimuksellinen kehittämistoiminta. Tampere: Tampereen yliopistopaino Oy. Saatavissa: http://urn.fi/URN:ISBN:978-951-44-7732-4 [viitattu 8.3.2019].

UnhandledExceptionFilter function s.a. Microsoft. WWW-dokumentti. Saatavissa: https://msdn.microsoft.com/en-us/library/windows/desktop/ms681401(v=vs.85).aspx [viitattu 12.4.2019].

Vectored Exception Handling. 2018. Microsoft. WWW-dokumentti. Päivitetty 31.5.2018. Saatavissa: https://docs.microsoft.com/en-us/windows/desktop/debug/vectored-exception-handling [viitattu 16.4.2019].

W3Schools s.a. Introduction to XML. WWW-dokumentti. Saatavissa: https://www.w3schools.com/xml/xml_whatis.asp [viitattu 8.4.2019].

Web-Based Enterprise Management s.a. DMTF. WWW-dokumentti. Saatavissa: https://www.dmtf.org/standards/wbem [viitattu 30.3.2019].

Windows Forms overview. 2017. Microsoft. WWW-dokumentti. Päivitetty 30.3.2017. Saatavissa: https://docs.microsoft.com/en-us/dotnet/framework/winforms/windows-forms-overview [viitattu 10.3.2019].

KUVALUETTELO

Liite 1

TESTIOHJELMAN LÄHDEKOODI

```
#include <iostream>
#include <sstream>
#include <string>

// Include FaultLib header and library
#include <FaultLib.h>
#pragma comment(lib, "FaultLib.lib")

// This function will try to divide by zero.
// It's laid out this way to confuse the compiler.
// Otherwise, the compiler refuses to process the function.
void faulty()
{
    faultLogInfo("Begin crashing!");
    int i = 0;
    char ayyy[10] = {};
    sprintf_s(ayyy, 10, "Boom %d", 5 / i); // Boom boom boom
}

// This is our callback function which the exception handler will call.
void callback(FAULT_CALLBACK_DATA* pData)
{
    // Let's make a cryptic error message no mortal can comprehend
    std::string message;
    std::stringstream info;
    {
        info << std::hex
            << "An exception " << pData->pExceptionInfo->ExceptionRecord->ExceptionCode
            << " was thrown at " << pData->pExceptionInfo->ExceptionRecord->ExceptionAddress
            << ".";

        message = info.str();
    }

    faultLogTrace(message);

    // Set crash reporter dialog message
    faultSetMessage(message.c_str());
}

int main()
{
    // FaultLib Configuration
    FAULT_CONFIG conf(
        "test",                         // Application identifier
        "test.exe",                     // Application name
        "1.0",                          // Application version
        "",                             // Working directory (use current)
        FaultModeFlag::kModeGui,        // Mode (use GUI)
        FaultSettingsFlag::kDefault,    // Settings (default: kUseOfflineCache | kClearFiles)
        FaultReportFlag::kDefault,      // Report configuration (default: kIncludeDump | kIn-
cludeLog |  kIncludeStackTrace | kIncludeSystemInfo)
        nullptr,                        // Custom handler (use default)
        callback,                       // Callback function
        MiniDumpNormal                  // MiniDump type flag
    );

    // Initialize FaultLib
    faultInit(conf);

    // Configure logger
    faultSetLogLevel(FAULT_LOG_LEVEL_TRACE);    // Set log level to Trace
    faultSetLogFilePath("test.log");            // Write log to file
    faultSetLogOutputStream(std::cout);         // Print log to console

    faultLogInfo("FaultLib Initialized");

    faulty(); // This will crash the program

    // We won't reach this point...
    faultUninit();
    return 0;
}
```