Riku Kaipainen

RAJAPINNAN TOTEUTTAMINEN ASIAKKAAN TUOTTEEN JA TESTIJÄRJESTELMÄN VÄLILLE

RAJAPINNAN TOTEUTTAMINEN ASIAKKAAN TUOTTEEN JA TESTIJÄRJESTELMÄN VÄLILLE

Riku Kaipainen
Opinnäytetyö
Syksy 2020
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, ohjelmistokehityksen suuntautumisvaihtoehto

Tekijä: Riku Kaipainen
Opinnäytetyön nimi suomeksi: Rajapinnan toteuttaminen asiakkaan tuotteen ja testijärjestelmän välille
Opinnäytetyön nimi englanniksi: Implementation of interface between customers product and testing system
Työn ohjaaja(t): Teemu Korpela
Työn valmistumislukukausi ja -vuosi: syksy, 2020
Sivumäärä: 34

Opinnäytetyön tavoitteena on kuvata OptoFidelityn asiakasprojektia varten kehitetyn Python-paketin kehityksen eri vaiheita ja sen integrointia kehitettävään testijärjestelmään.

Asiakkaalla on monta erilaista tuotetta, jotka käyttävät joko RS485-sarjaliikenneväylää tai TCP/IP-socket-palvelinta ja tarjoavat rajapinnan, jonka avulla testijärjestelmä pystyy lukemaan tarvittavia tietoja tuotteelta ja käynnistämään testirutiineja.

Lopputuloksena kehitettävä paketti otettiin käyttöön testausjärjestelmässä asiakkaan tuotteen testaamiseksi.

Asiasanat: ohjelmistokehitys, testijärjestelmät, rajapinnat, laiteajurit, Python

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology, Option of Software Development

---

Author(s): Riku Kaipainen
Title of thesis: Implementation of interface between customers product and testing system
Supervisor(s): Teemu Korpela
Term and year when the thesis was submitted: autumn, 2020
Pages: 34

The objective of this thesis is to describe the different development phases and the integration of a Python package developed for OptoFidelity's customer project where a testing system is being developed.

The customer has multiple products that use either RS485 serial communication or TCP/IP sockets to serve an interface, that the testing system can use to retrieve required information from the product and initiate testing sequences.

As a result, the developed package was used on the testing system to control the customers products.

Keywords: software development, testing systems, interfaces, hardware drivers, Python

SISÄLLYS

SANASTO

CRC	Virheehavaitsemisalgoritmi, jonka avulla voidaan nähdä, onko viestien sisältö vaihtunut tai kadonnut siirron yhteydessä. Englanniksi *Circular Redundancy Check*.
Koristaja	Useassa ohjelmointikielessä käytettävä konsepti funktiosta, joka käärii toisen funktion toteutuksen itseensä siten, että ensin käärivä funktio ajaa jonkin loogisen pätkän koodia ja suorittaa käärityn funktion määritetyssä kohdassa. Englanniksi *Decorator*.
Paradigma	Tarkoittaa jonkin tieteenalan yleisesti hyväksymää oppirakennelmaa, ajattelutapaa tai suuntausta.
Python-paketti	Python-ohjelmointiekielessä käytettävä kokoelma koodia, joka sitoo loogisen ohjelmakokonaisuuden luokat ja metodit yhteen. Tämän paketin ominaisuuksia voidaan käyttää asentamisen jälkeen muissa Python-paketeissa.
RS485	Potentiaalieroon perustuva sarjaliikenneväylä standardi, jonka avulla voidaan siirtää dataa kahden tai useamman väylää käyttävän laitteen välillä.
Rajapinta	Tarkoittaa kahden ohjelman välisen kommunikaation määritelmää, jonka avulla ohjelmat voivat jakaa tietoa ja suorittaa operaatiota keskenään. Englanniksi *Application Programming Interface*.
SOLID	Lyhenne periaatteesta, jossa keskitytään olio-ohjelmointiparadigmassa olevien ominaisuuksien oikeaan käyttöön erittelemällä luokat yhden tehtävän suorittaviksi kokonaisuuksiksi. Englanniksi *Single-responsibility-principle, Open-closed-principle, Liskov*

	substitution principle, Interface segregation principle, Dependency inversion principle.
Skripti	Tarkoittaa yksittäistä ajettavaa tietokoneohjelmaa, joka on ohjelmoitu jollakin skriptikielellä kuten Pythonilla.
Socket	Verkkoliikenteessä käytettävä tapa tarjota muille verkossa oleville laitteille päätepiste, johon yhdistämällä voidaan lähettää ja vastaanottaa dataa jonkin verkkoprotokollan avulla kuten TCP/IP. Yhteydessä on aina vähintään yksi isäntä, joka tarjoaa päätepisteen, sekä yksi alainen, joka muodostaa yhteyden päätepisteeseen.
TCP/IP	Verkkoliikennöinnissä käytettävä tietoliikenneprotokolla, joka varmistaa kahden laitteen välisessä kommunikoinnissa yhteyden varmistuksen ja pakettien perille menemisen. Englanniksi *Transmission Control Protocol / Internet protocol*.

1 JOHDANTO

Olen toiminut OptoFidelityllä vuodesta 2019 alkaen. Aloitin osa-aikaisena harjoittelijana, mutta ajan myötä työmääräni sekä vastuuni ovat kasvaneet paljon. Tehtäviini on kuulunut pääasiassa ohjelmistokehityksen tehtävät, kuten ohjelmiston suunnitteleminen, kirjoittaminen sekä dokumentointi. Lisäksi olen avustanut asiakkaitamme vierailuilla, joilla olen asentanut heille ohjelmistoa sekä laitteistoa, pitänyt koulutusta kehittämästämme ohjelmistosta ja auttanut heitä erilaisissa virhetilanteissa.

Työtehtäväni ovat keskittyneet uuteen asiakasprojektiin, jota tässä opinnäytetyössä kuvaillaan. Asiakasprojektin tavoitteena on kehittää asiakkaalle uusittua versiota Fusion-testijärjestelmästä (1), joka täyttää asiakaan tarpeet.

KUVA 1. Fusion-testijärjestelmä (1)

1.1 Opinnäytetyön lähtökohdat

OptoFidelityn asiakasprojektissa määriteltiin, että asiakas tarvitsee Fusion-testijärjestelmästä muokatun version, joka toimii heidän tuotteidensa kanssa, jotka eivät ole ennestään yhteensopivia nykyisen järjestelmän kanssa. Tuotteille ei pystytty keskustelemaan käyttäen nykyisiä ajureita, joten niitä varten täytyi kehittää uudet ajurit.

Asiakkaalla oli useampi tuote, jotka käyttivät joko RS485-sarjaliikenneväylää (2) tai TCP/IP-socket-palvelinta (3) kommunikoimiseen (tästä eteenpäin TCP/IP-socket esiintyy sanalla ”socket”). Tehtävänä oli siis suunnitella ja ohjelmoida ajurit, joilla näille rajapinnoille pystyttiin kommunikoimaan.

Projektissa hyödynnettiin Fusion-testijärjestelmän olemassa olevaa ohjelmistoa, jota lähdettiin muokkaamaan asiakkaan tarpeisiin sopivaksi. OptoFidelityn kehittämään Fusion-testausärjestelmä tarjoaa ison määrän erilaisia työkaluja, joita voidaan käyttää yhteistyössä 3-akselisen robotin kanssa suorittamaan erilaisia testausoperaatioita laitteille, jotka ovat sen työalueella.

Testausjärjestelmän ohjelmisto pyörii Debian-käyttöjärjestelmässä, joten kehitettävän paketin toteutuksessa on otettava huomioon, minkä tyyppiset ratkaisut ovat yhteensopivia järjestelmän kanssa.

Lisäksi testijärjestelmän laitteisto päivittyi projektin myötä, koska asiakkaan tuotteet vaativat fyysisen rajapinnan, jonka kautta tuotteiden kanssa voidaan kommunikoida käyttäen asiakkaan määrittelemää rajapintaa. Testijärjestelmään lisättiin myös uusia työkaluja, jotka olivat tarpeellisia asiakkaan tuotteiden testaukseen.

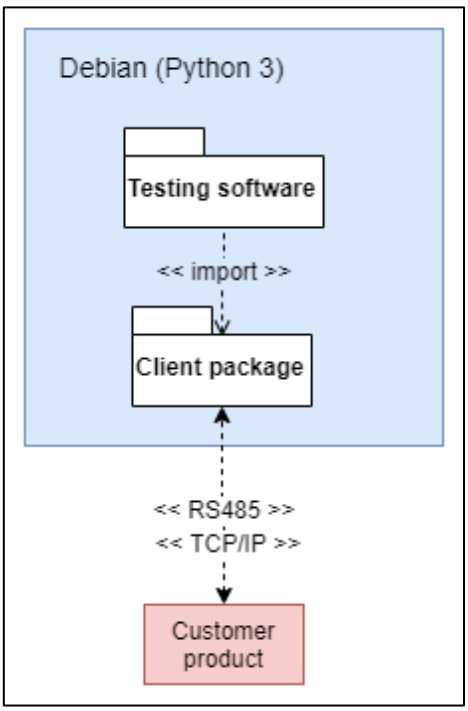
1.2 Opinnäytetyön tavoitteet

Tämän opinnäytetyön tavoitteena on kuvata asiakasprojektia varten kehitetyn Python-paketin kehityksen eri vaiheita ja toteutuksen onnistumista.

Python-paketin oli pystyttävä kommunikoimaan asiakkaan rajapinnan kanssa siten, ettei pakettia käyttävän ohjelmiston tarvitse ymmärtää, miten rajapinnan tarkempi toteutus toimii. Testijärjestelmän oli pystyttävä antamaan yksinkertaisia

käskyjä paketille, joka puolestaan takaa käskyjen toteuttamisen asiakkaan tuotteella.

Tämä opinnäytetyö keskittyy paketin kehitykseen ja integrointiin, joten itse asiakasprojektin tuloksia ei käsitelty tässä työssä, ellei paketin tulokset vaikuttaneet siihen olennaisesti. Kuvassa 2 nähdään kehitettävän paketin riippuvuudet testijärjestelmään ja tuoterajapintaan.



KUVA 2. Kehitettävän paketin kuvaus käytettävässä järjestelmässä

2 SUUNNITTELU

Python-paketin kehityksen kannalta oli tärkeää suunnitella jonkin tyyppinen rajapinta jo aikaisessa vaiheessa, koska paketti tullaan integroimaan isompaan järjestelmään. Väistämättä koodin toteutus tuli vaihtumaan pienissä määrin paketin kehittyessä, mutta se helpotti paketin integroimista käytettävään järjestelmään, sillä kehitettävän ohjelman rakenteessa voitiin ottaa jo kehitysvaiheessa huomioon käytettävän paketin eri toimintaperiaatteet ja käytännöt. Käytännössä tämä tarkoitti erilaisten luokkien suunnittelemista ja niiden metodien sekä muuttujien kartoittamista.

Aikaisessa vaiheessa määriteltiin, minkä tyyppinen kehitettävän rajapintaluokan piti olla. Kyseisen luokan täytyi sisältää järjestelmän määrittelemät geneeriset metodit, joita pystyttiin käyttämään riippumatta siitä, oliko käytössä tämä vai jonkun muun paketti. Aiempi toteutus perustui siihen, että tuoteajureita saattoi olla useita erityyppisiä, ja tämä järjestelmä haluttiin säilyttää.

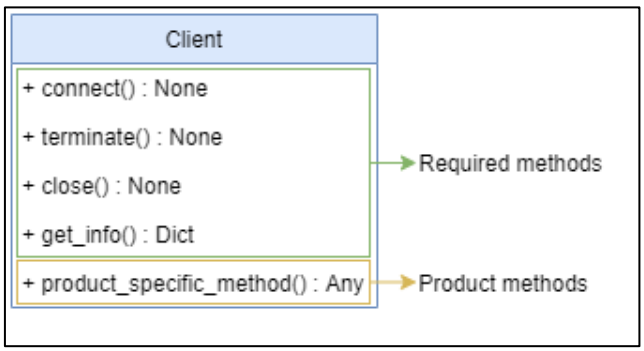
Tavoitteena oli säilyttää paketin sisäinen toiminta mahdollisimman yksinkertaisena. Sitä varten luokkien kehityksessä pyrittiin käyttämään SOLID-periaatetta. Periaatetta käyttäen erilaiset loogiset yksiköt koodissa eritellään omiin luokkiinsa, jotka pyrkivät toteuttamaan vain yksittäisen tehtävän, joka yksinkertaistaa koodin rakennetta ja luettavuutta. SOLID-periaatteen tarkoitus ohjelmistokehitysprojekteissa on hyödyntää olio-ohjelmointiparadigmaa oikein, jotta säästyttäisiin yleisiltä ohjelmointiongelmilta kuten koodin yksiselitteisyydeltä, toistolta ja ylläpidon vaikeudelta (4).

2.1 Pääluokan rajapinta

Yksi vaatimuksista kehitettävälle paketille oli se, että paketilla on oltava yksi pääluokka. Luokan tehtävänä on tarjota yksinkertainen rajapinta pakettia käyttävälle järjestelmälle, joka mahdollistaa yksinkertaisen tuotteen ohjauksen ja sisältää tarpeelliset hallinta metodit.

Luokan rajapinnan suunnitteleminen tapahtui suhteellisen helposti, koska projektia varten oli olemassa testijärjestelmän spesifikaatiodokumentti, joka kertoi tarkalleen, minkälaisia operaatioita asiakkaan tuoterajapinnan pitäisi tukea.

Ohjelmistotiimi määritteli kehitettävän pääluokan metodipohjan (kuva 3) sitä varten, että paketti pystyy kommunikoimaan kehitetyn testijärjestelmänsä kanssa oikein.



KUVA 3. Paketin Client-pääluokan perusrakenne

Client-luokan toteutuksessa oli otettava huomioon sitä käyttävän järjestelmän vaatimukset, jotka asettivat kuvan 3 mukaiset vaatimukset metodeille. Testausjärjestelmä kutsuu näitä riippumatta siitä, onko käytössä tämä paketti vai jokin muun rajapinnan toteuttava luokka.

Asiakkaan tuotteen tarjoamia ominaisuuksia en tässä opinnäytetyössä voi paljastaa, joten kaikki viittaukset asiakkaan tuotteen ominaisuuksista ovat esimerkin omaisesti esitettyjä.

Tärkein tehtävä luokalla oli tarjota kattava yhdistysmetodi, joka osasi selvittää, onko järjestelmään kytketty jokin asiakkaan tuote käyttämällä jotakin tunnetuista protokollista. Tämä toteutus suunniteltiin tapahtuvan kahden erillisen luokan sisässä, jossa ne pystyvät kokeilemaan yhteyden muodostamista asiakkaan rajapintaan käyttäen omaa protokollaansa, jotka olivat tässä tapauksessa RS485 ja socket.

Socket-yhteyden muodostamisen jälkeen oli turvallista olettaa, että palvelin oli valmis vastaamaan, koska muuten yhteyttä ei olisi voitu muodostaa alun perinkään.

Sarjaliikenneväylän tilanne oli hieman monimutkaisempi ja järjestelmän oli pystyttävä lähettämään asiakkaan protokollan mukaisia viestejä, joihin väylän kautta oli myös saatava vastaus, jotta voitiin luottaa yhteyden olevan muodostettu.

Luokan oli pystyttävä myös tarjoamaan metodit yhteyden terminoimiselle ja katkaisemiselle hallitusti. Yhteydensulkemismetodin tehtävä oli informoida asiakkaan rajapinnalle, että yhteys voidaan sulkea sekä järjestelmän fyysinen yhteys voidaan katkaista (socketin kuuntelu tai sarjaliikenneväylän lukeminen). Luokan täytyi myös vapauttaa käytössä olevat resurssit uutta yhteydenottoa varten. Terminointimetodi jäi vain tyhjäksi toteutukseksi, koska tälle rajapinnalle pelkkä yhteyden muodostaminen ja sulkeminen riitti, eikä terminointia tarvittu toteuttaa.

2.2 Tuoterajapinta

Pääluokan tehtävä oli tarjota pakettia käyttävälle järjestelmälle rajapinta tuoteominaisuuksiin. Sen vuoksi suunniteltiin tehdä kaksi rajapintaluokkaa, joiden tehtävänä oli hallita joko RS485- tai socket-rajapinnan toteutusta omatoimisesti.

2.2.1 RS485-rajapinta

Tietyt asiakkaan tuotteet käyttivät RS485-sarjaliikenneväylää kommunikoimiseen käyttäen asiakkaan kehittämää rajapintakuvausta. Kyseinen rajapintakuvaus sisälsi erilaisia ominaisuuksia, jotka eivät olleet välttämättömiä rajapintaluokan toimintaan. Tästä syystä asiakkaan dokumentaatiosta piti seuloa vain tärkeät ja tarpeelliset ominaisuudet implementoitavaksi itse rajapintaluokkaan.

Asiakkaan rajapinta oli suunniteltu siten, että järjestelmässä on yksi isäntä, joka ohjaa mahdollisesti usean sarjaliikenneväylässä olevan alaisen toimintoja. Jokaisella laitteella on oma yksilöllinen tunniste, joihin isäntä pystyy viittaamaan ja pyytää heitä suorittamaan jonkin tyyppisen operaation, johon alaisen täytyy antaa tunnetun tyyppinen vastaus pyydetyn sisällön mukaan.

Fyysisesti tuotteen RS485-rajapintaan oli tarkoitus kytkeytyä käyttäen TCP/IP-muunninta, joka muuntaa saamansa datan RS485-sarjaliikenneväylään

sopivaksi. Aikaisessa vaiheessa siirryttiin kuitenkin käyttämään asiakkaan ehdottamaa USB-muunninta.

Rajapintakuvaus määrittelee, että sarjaliikenneväylässä kulkevan datan täytyy olla asiakkaan haluamassa formaatissa, jotta alaiset voivat ymmärtää lähetetyt komennot oikein. Data kulkee sarjaliikenneväylässä yhtenä kokonaisena sarjana tavuja (kuva 4), jossa tietyt merkit kertovat mistä viesti alkaa, mihin se loppuu ja muita olennaisia tietoja.

KUVA 4. RS485-rajapinnan käyttämä viestiformaatti yksinkertaistettuna

Asiakkaan ehdottama USB-muuntaja toimi käyttäen ajureita, jotka listasivat sen tietokoneen sarjaliikenneväylissä, joten tämä mahdollisti minulle jo ennestään tutun PySerial-kirjaston (5) käyttämisen kommunikointia varten. Kirjasto mahdollistaa sarjaliikennettä käyttävien laitteiden kommunikoimisen tarjoamalla yksinkertaiset ominaisuudet datan lähettämiseen ja vastaanottamiseen tavupohjaisesti, mikä sopi täydellisesti asiakkaan rajapinnan tarpeisiin.

Rajapintakuvauksessa kerrottiin, että jotta järjestelmä toimisi oikein, täytyi isännän suorittaa tietyin väliajoin alaisten etsintä- ja kiertokyselyjä. Tämän avulla pystyttiin määrittelemään, onko yhteys alaisiin kunnossa. Tätä toimintoa varten suunniteltiin toimintojen suorittamista erillisissä säikeissä, jotta rajapinnan toimintoja voidaan kutsua riippumatta siitä, onko se suorittamassa jotain rutiinioperaatiota tuotteella.

Säikeiden käyttö Pythonissa onnistuu helposti käyttäen peruskirjaston tarjoamaa Threading-pakettia (6). Rajapinnan toimintaa varten alustava suunnitelma oli toteuttaa oma säie rajapinnan vaatiman rutiinin suoritukseen ja toinen sarjaliikenneväylän lukuun sekä kirjoitukseen. Lopullinen toteutus poikkesi hieman tästä mallista, jota kuvaillaan osiossa 3.1.2.

2.2.2 Socket-rajapinta

Loput asiakkaan tuotteista käyttivät socketia kommunikoimiseen. Projektin alkuvaiheessa tästä rajapinnasta ei ollut muuta informaatiota saatavilla kuin se, että kommunikaatio tapahtuu käyttäen sockettia.

Socketeja voidaan käyttää Pythonin peruskirjastosta löytyvällä Socket-paketilla (7), jonka avulla voidaan lähettää ja vastaanottaa dataa socket-palvelimelta käyttäen TCP/IP:tä. Sitä hyödyntäen kommunikointi asiakkaan rajapinnan kanssa on yksinkertaista ja helppo kehittää.

Rajapinnassa liikkuvan datan formaatista ja viestirakenteista ei ollut minkäänlaista tietoa saatavilla, joten sen suunnittelu piti siirtää myöhemmäksi.

2.3 Testaus

Pakettia suunnitellessa yksi käsiteltävä aihe oli, miten asiakkaan rajapinnan eri ominaisuuksia pitäisi testata käytännössä. Koodin testaus voidaan toteuttaa käyttäen integraatio- ja yksikkötestejä, joiden avulla voidaan tarkistaa ohjelman yksittäisten osien toiminta ja varmistaa, että ohjelman eri komponentit keskustelevat oikein keskenään (7).

Vaihtoehtoja testaukseen Pythonin tapauksessa on monia, mutta helpointa on käyttää valmiita testauskirjastoja, jotka mahdollistavat koodin automaattisen testauksen ja kohdistettujen testien ajamisen selkeitten tulosten kanssa. Yleisiä kirjastoja, joita voidaan käyttää testauksen automatisointiin, ovat esimerkiksi unittest, nose ja pytest (8).

Aiempien kokemuksien mukaan Unittest-kirjaston käytössä ilmeni testien kirjoituksen ja ajon aikana monesti epämääräisiä ongelmia, joten pytest valittiin käytettäväksi kirjastoksi siinä toivossa, että se toimisi paremmin.

2.4 Integraatio

Paketin integroiminen Pythonilla on erittäin suoraviivaista. Käytännössä pakettien asennus tapahtuu käyttäen pip-paketinasennustyökalua (9). Paketti asennetaan

sieltä löytyvän setup.py-konfiguraation perusteella ja asentanut järjestelmä voi sen jälkeen käyttää paketin tarjoamia ominaisuuksia omassa koodissaan.

Yleensä Python-ohjelmoinnissa on tapana luoda oma virtuaaliympäristö, joka tarkoittaa käytännössä jonkin Python-version käyttämistä omana eristettynä pakettina. Siihen asennetaan vain suoritukseen tarvittavat paketit. Erilliset projektit voivat käyttää jokainen omaa virtuaaliympäristöä, joka helpottaa ohjelmaympäristön ylläpitämistä ja asennettujen pakettien hallitsemista (10).

Pääasiassa paketin ominaisuuksia tullaan käyttämään testerijärjestelmän määrittelemissä testiskripteissä, joiden on tarkoitus kuvata yksittäisen tuotteen ominaisuuden testausta. Näiden skriptien kirjoitus on tarkoitus tapahtua, kun paketti on saatu sellaiselle tasolle, että sillä voidaan jo konkreettisesti testata joitakin tuotteen ominaisuuksia.

3 TOTEUTUS

Paketin kehitys lähti käyntiin suunnitelman mukaisesti. Ensimmäinen kehitettävä osa oli pääluokan yleinen metodistruktuuri ilman funktionaalisuutta, jonka päälle voitiin rakentaa muita tarvittavia komponentteja ja hiljalleen tuoda sisältöä, kun sitä voitiin toteuttaa.

Asiakkaan testirajapinnan kuvaukseen ei projektin alussa ollut muuta materiaalia, kuin miten RS485-kommunikaatioprotokolla toimi ilman testirajapinnan funktioita. Sen kehittäminen oli ensimmäinen tehtävä paketin kehityksessä.

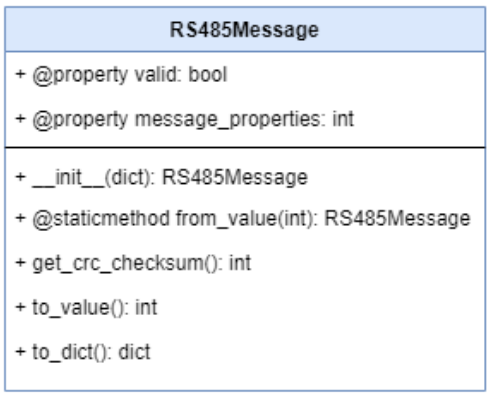
3.1 RS485-rajapinta

Luokan ohjelmointi lähti vauhdikkaasti liikkeelle, koska asiakkaan rajapinnan toteutus oli hyvin dokumentoitu ja pyserial-kirjastoa oli helppo käyttää.

3.1.1 Viestirakenne

Asiakas oli dokumentoinut selkeästi sarjaliikenneväylässä kulkevien viestien formaatin, joten ensimmäinen askel oli luoda luokka, joka kuvasi viestien rakennetta.

Kuten kuvasta 5 voidaan nähdä, luokan perusideana oli tarjota kaikki viestissä käytettävät ominaisuudet julkisesti, siten että ne on helppo lukea ja antaa niille uusia arvoja, joiden varmistetaan olevan protokollan mukaisia asetuksen yhteydessä.



KUVA 5. Asiakasrajapinnan viestiformaatin luokkakuvaus

Tämä oli helppo toteuttaa käyttäen Pythonin tarjoamaa property-koristajaa. Sen avulla jokainen luokan viestiominaisuus voidaan lukea määritellystä luokan muuttujasta, jonka arvon vaihtaminen on sidonnainen johonkin ensin ajettavaan funktioon. Muuttujan määrittelyssä voidaan asettaa ehtoja sille, minkä tyyppisiä parametrejä kyseinen muuttuja hyväksyy (kuva 6).

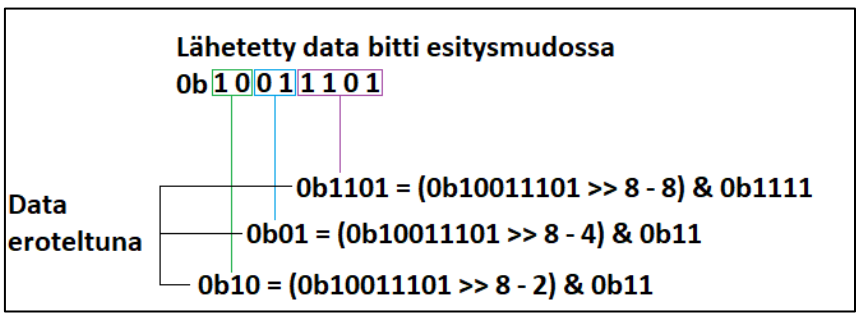
```python
class Example:
    _member = 0

    @property
    def member(self) -> int:
        return self._member

    @member.setter
    def member(self, value: int) -> int:
        if not isinstance(value, int):
            raise ValueError('Invalid property value')
        if 0 <= value <= 10:
            self._member = value
        else:
            raise ValueError('Invalid property value')
```

KUVA 6. Esimerkki property-koristajan käytöstä

Tämän lisäksi luokassa on metodeja, jotka helpottavat luokan instanssien luomista ja kääntämistä sarjaliikenneväylässä ymmärrettävään data formaattiin.

Viestien purkaminen ja instantoiminen

Staattinen metodi *from_value* mahdollistaa luokan instanssien luomisen suoraan kokonaisluvusta. Asiakkaan dokumentti määrittelee lähetetyn datan bitti- ja tavujärjestyksen, joten kokonaisluku voidaan erotella osiin käyttäen loogisia vertauksia sekä bittien siirtoa (kuva 7).



KUVA 7. Esimerkki kokonaisluvun erottelusta osiin

Käyttäen yllä mainittua erottelumenetelmää, luokan instansseja voidaan kuvailla kokonaislukumuodossa, joka pystytään helposti lähettämään sarjaliikenneväylässä yksittäisinä tavuina.

Prosessi on muuten täysin sama, mutta viestin ensimmäistä arvoa lähdetään bittisiirtämään vastakkaiseen suuntaan kuin edellisessä kuvassa.

Viestien virheentarkistus

Asiakkaan RS485-rajapinta kuvauksessa kerrotaan, että asiakkaan tuotteet vaativat viesteihin lisättävän 16-bittisen CRC-koodin, joka on generoitu viestin muusta sisällöstä.

CRC on tehokas tapa tarkistaa, onko siirron aikana lähetetylle datalle tapahtunut viestinsiirtohäiriötä, jonka vuoksi viestistä on esimerkiksi voinut vaihtua tai kadota dataa. Virheentarkistus CRC:llä perustuu generoituun koodiin, joka on tulos siitä, kun alkuperäiseen dataan on lisätty jokin määrä nollabittejä, ja sitten se on jaettu polynomigeneraattorin kahden jakojäännöksellä (11).

CRC-koodin generoimiseen oli yksinkertaisempaa valita valmis kirjasto, koska muuten virheentarkistusalgoritmi pitäisi kirjoittaa itse. CRC:n toimintaa testattiin siten, että asiakas antoi esimerkin toimivasta viestistä ja sen perusteella generoitiin CRC-koodi, jota sitten verrattiin viestin lopussa määriteltyyn koodiin.

Kokeilin käyttää pycrc- ja crcmod-kirjastoja (12, 13), koska niiden dokumentaation mukaan ne sisälsivät 16-bittisen CRC:n laskemisen. En kuitenkaan saanut kummankaan kirjaston generointimetodeja palauttamaan oikeaa CRC-koodia viestiin verrattuna. Yksi ongelmista oli se, että CRC:n generoinnissa on monia muuttujia kuten, invertoidaanko jakotulos, mitkä ovat aloitus ja polynomi arvo ja viedäänkö lopputulosta XOR-filtterin läpi, joita nämä kaksi aiemmin mainittua kirjastoa eivät näyttäneet tarjoavan.

Lopuksi kuitenkin kokeilin kolmatta kirjastoa nimeltä libscrc (14), joka tarjosi monipuolisemmat metodit CRC:n generoimiseen vapaasti asetettavien parametrien avulla. Kirjaston avulla generoitu koodi vastasi esimerkki viestin CRC:tä.

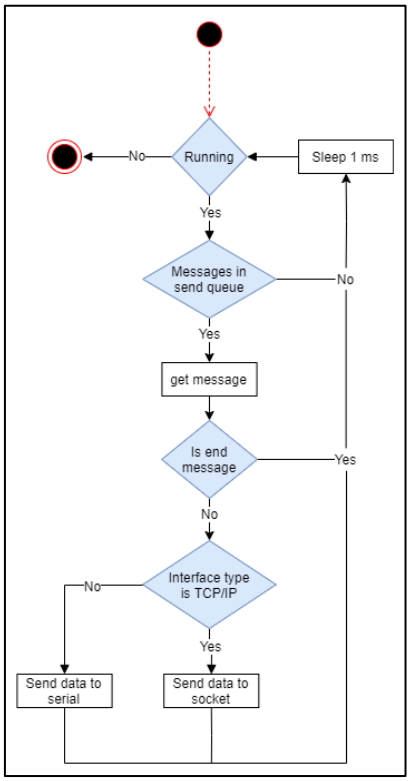
Datan lähetyksen ja vastaanoton yhteydessä tarkistetaan siis, onko viesti virheellinen vai ei, käyttäen tätä tekniikkaa.

3.1.2 Lähetettävän ja vastaanotettavan datan käsittely

Kuten suunnitteluosiossa kerrottiin, kommunikaatio päätettiin toteuttaa säikeissä. Toteutuksen aikana todettiin, että helpoin tapa sarjaliikenteen hallintaan oli luoda oma säie datan lähetykselle ja vastaanotolle, koska siten säikeiden toteutusta olisi helpompi ylläpitää.

Lähetyssäie lukee luokassa määriteltyä lähetysjonoa ja kun se huomaa uuden viestin olevan lisätty, se purkaa viestin tavuesitys muotoon kokonaisluvusta ja lähettää koko viestin sarjaliikenneväylään.

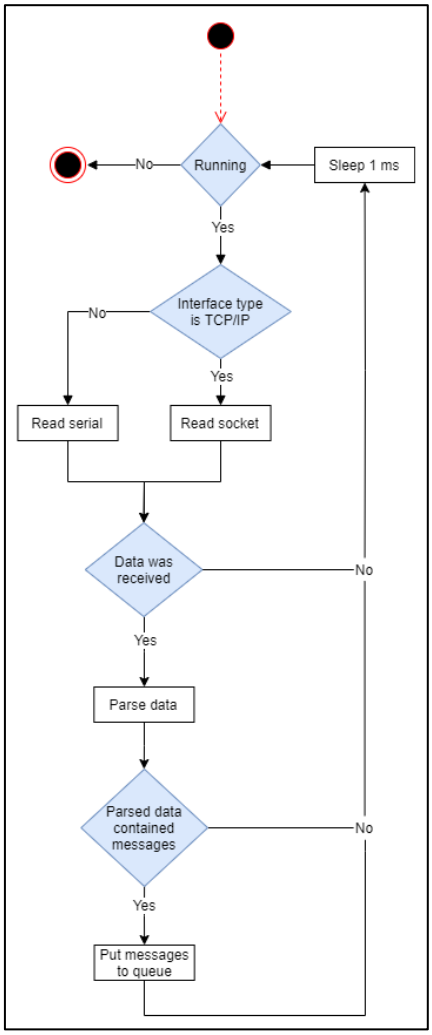
Kuten kuvassa 8 voidaan nähdä, ennen datan lähetystä on vaihe, jossa tarkistetaan minkä tyyppinen rajapinta on käytössä.



KUVA 8. Lähetyssäikeen vuokaavio

Luokkarakenne muuttui jonkin verran suunnitellusta mallista, jossa RS485- ja socket-rajapinta olivat erillään, koska niiden toteutus loppujen lopuksi vastasi lähes kokonaan toisiaan. Tästä kerron lisää osiossa 3.2.

Vastaanottosäikeen toiminta vastaa pitkälti lähetyssäikeen toimintaa, mutta sen datan käsittely osio on hieman monipuolisempi (kuva 9).



KUVA 9. Vastaanottosäikeen vuokaavio

Ensimmäiseksi säikeessä yritetään lukea dataa käyttäen socketia tai sarjaliikenneväylää.

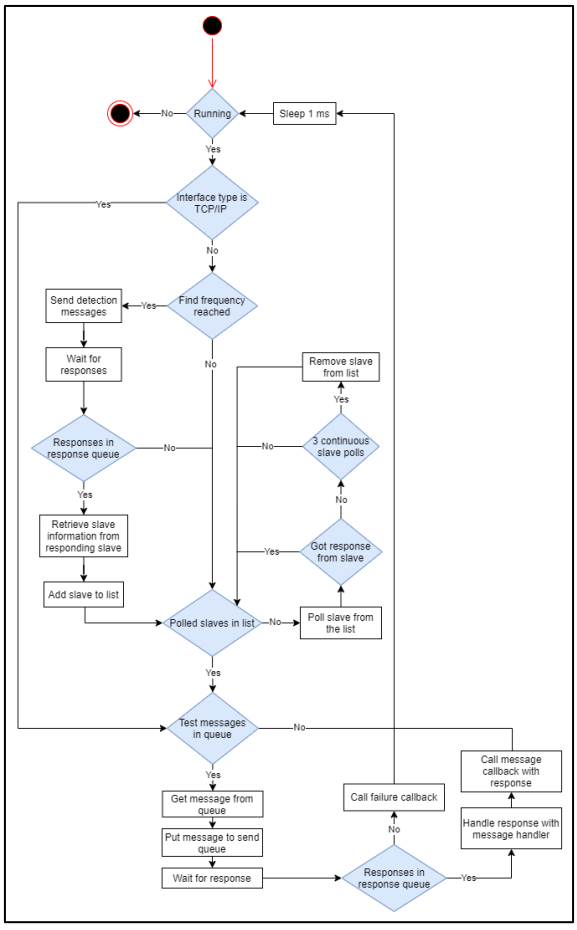
Mikäli dataa pystyttiin lukemaan, se käsiteltiin siten, että jos aiemman datan ja uuden datan yhdistyksestä saadaan muodostettua kokonaisia viestejä, ne siirretään vastaanotettujen viestien jonoon. Viestiä edeltävä data hävitetään ja sen jälkeinen data säilytetään uutta lukua varten.

Datan käsittelyssä ilmeni ongelmia muutaman kerran kehityksen aikana. Yksi huomionarvoinen seikka oli se, että data käsiteltiin perustuen aloitus- ja lopetustavuihin. Tämä aiheutti ongelmia tilanteissa, jossa nämä kyseiset tavut ilmestyivät itse viestin sisällössä joko yhtenä viestin ominaisuutena tai datassa.

Ongelma korjattiin asettamalla datan pilkkomiseen tietty viestin minimipituus ja tarkistamalla viestistä löytyvän datan pituus parametrin arvoa. Mikäli lopetustavu löytyi ennen haluttua viestin pituutta, se jätettiin käsittelemättä ja siirryttiin datankäsittelyssä eteenpäin.

3.1.3 Kommunikaatiosekvenssi

Kun itse lähetettävien ja vastaanotettavien viestienhallinta oli toteutettu omilla säikeillä, pystyin aloittamaan kommunikaatiosekvenssin suunnittelemisen, johon kuului uusien alaisten etsiminen, löydettyjen alaisten kiertokyselyn suorittaminen ja testaus operaatioiden suorittaminen (kuva 10).



KUVA 10. Kommunikointi sekvenssin vuokaavio

Ensimmäiseksi sekvenssissä tarkistetaan, onko käytössä RS485- vai socket-rajapinta.

Kun käytössä on RS485-rajapinta, niin ensimmäisenä suoritetaan operaatio, jonka tavoitteena on löytää alaisia linjasta, joiden kanssa voidaan kommunikoida. Jos alaisia löytyi, näille täytyi suorittaa kiertokysely, jolla voitiin varmistua siitä, että ne ovat valmiita toimimaan. Mikäli jokin tietty alainen ei vastannut kiertokyselyyn kolmeen kertaan, se poistettiin listasta, koska sen kanssa ei voitu kommunikoida.

Lopuksi käytiin läpi viestijono, johon sisältyi lähetettävät viestit, sen argumentit ja epäonnistuneen sekä onnistuneen viestin takaisinkutsufunktiot. Jos viestejä löytyi, jonosta luettiin yksi viesti ja se lisättiin viestijonoon lähetettäväksi sekä sen vastausta jäätiin odottamaan. Mikäli odotuksen jälkeen löytyi vastaus, sen sisältö käsiteltiin viestin sisältämällä käsittelijäfunktiolla ja sen tulos palautettiin takaisinkutsufunktion kautta. Muussa tapauksessa kutsuttiin virheitä käsittelevää takaisinkutsufunktiota virheen merkiksi.

3.2 Socket-rajapinta

Suunnitelman mukaisesti aluksi kehitettiin erillistä luokkaa kummallekin rajapinnalle, mutta rajapinta kuvauksen saatavuus pakotti kehittämään RS485-rajapintaa eteenpäin ensin.

Socket-rajapinnan tarkemman toteutuksen selvittyä oli selvää, että luokan toteutus on käytännössä sama kuin RS485:n tapauksessa. Olennainen ero toteutuksessa oli se, ettei kommunikaatioprosessissa tarvinnut erikseen etsiä alaisia.

Sockettia käytettäessä oletuksena on, että yhteyden muodostamisen jälkeen palvelin ja asiakas ovat valmiita vastaanottamaan ja lähettämään dataa toisilleen. Tämän vuoksi asiakas ei ollut kehittänyt vastaavaa kommunikaatiokerrosta, joka oli RS485:ssä käytössä.

Tämä johti siihen, että luokan toiminnot yhdistettiin kaikilta muilta osin paitsi datan lukemiselta ja kirjoittamiselta, jotka tapahtuivat eri tavalla riippuen, mikä on konfiguroitu rajapinta sillä hetkellä.

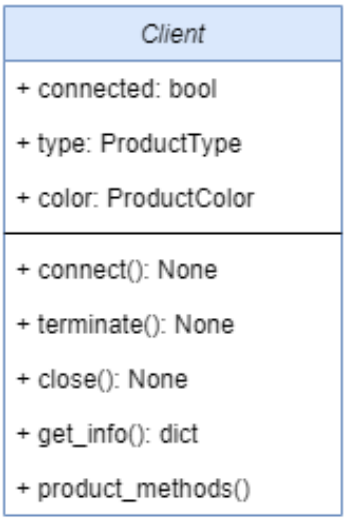
3.3 Data välityksen erot

Data formaatin ja viestien sisällön puolesta viestit olivat kummallakin rajapinnalla samanlaiset. Tästä huolimatta eroja toteutusten välillä oli muutama.

Esimerkiksi RS485-rajapinnassa kaikki välitetty data oli binäärimuodossa. Socket-rajapinnassa osa viestin sisällöstä oli kuvattu ASCII-numeroformaatissa, jonka vuoksi viestit täytyi käsitellä hieman eri tavalla, vaikka viestien sisältö oli olennaisesti sama.

3.4 Pääluokan toteutus

Kun tuoterajapintaluokan toteutus lähestyi valmistumista, aloitettiin pääluokan toteutus. Käytännössä luokan toteutus koostui rajapintaluokan kautta lähetetyistä kutsuista tuotteelle asiakkaan rajapintakuvauksen mukaisesti, mutta itse rajapintakutsujen lisäksi luokan piti tarjota logiikkaa, jolla käsiteltiin rajapinnasta haettuja tietoja. Lisäksi täytyi pystyä selvittämään, voidaanko johonkin tuotteeseen yhdistäytyä vai ei. Kuvasta 11 nähdään lopullinen pääluokan metodi- ja jäsenrakenne.



KUVA 11. Pääluokan kuvaus lisättyjen ominaisuuksien kanssa

3.4.1 Yleiset ominaisuudet

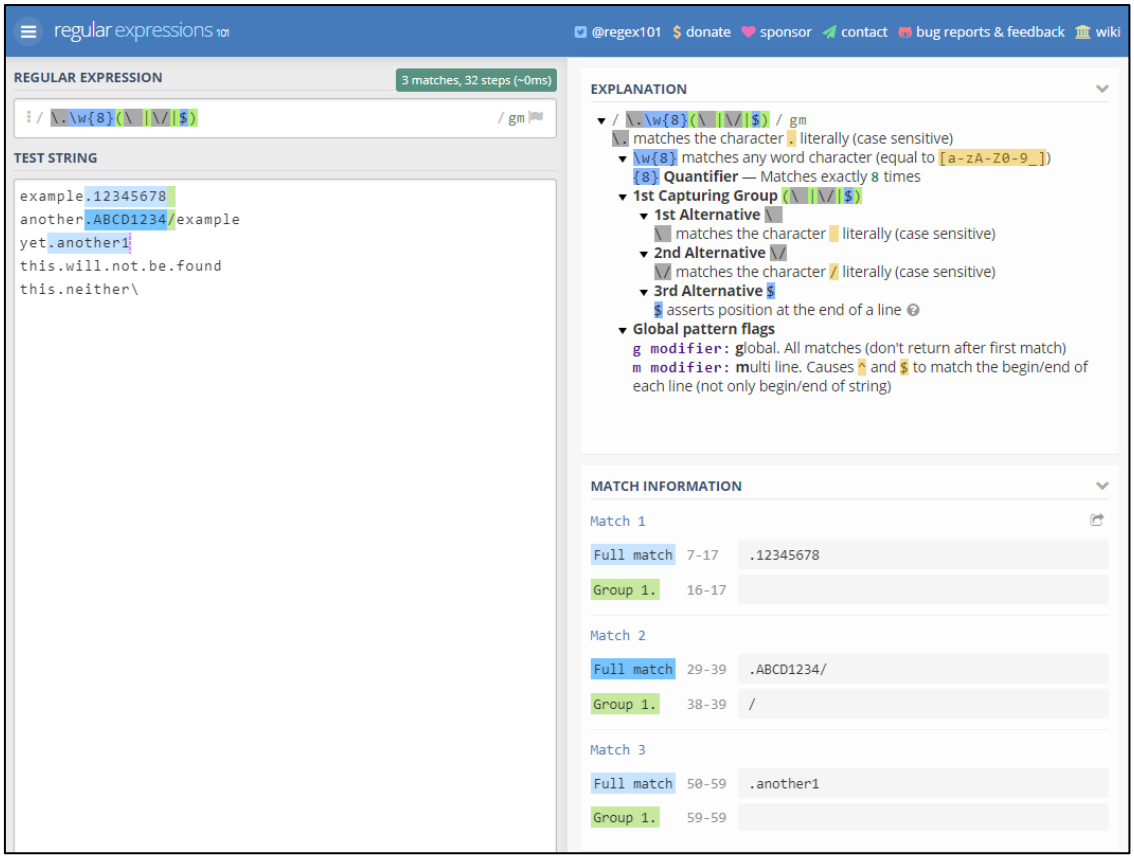
Yhdistysmetodin toteutuksen täytyi toimia siten, että sisäisesti järjestelmä yrittää yhteyden muodostamista sekä socket- ja RS485-rajapinnan kautta, kunnes yhteys saadaan muodostettua tai aikaa on kulunut määritetyn ajan verran, jolloin metodi nostaa poikkeuksen virheen merkiksi.

RS485:n tapauksessa yhteys kommunikointiväylään on aina auki, joten yhteys on muodostettu siinä vaiheessa, kun protokollan mukaisesti yksi tai enemmän alaisia on vastannut etsintäpyyntöihin. Socketin tapauksessa yhteyden muodostus riittää varmistukseksi, koska palvelimen täytyy olla auki, jotta socketti voidaan yhdistää.

Yhdistyksessä olennainen asia oli saada selville minkä tyyppinen tuote on kytketty järjestelmään, jotta testatessa ajetaan oikeat testikomennot. Tämä tapahtui kätevästi pyytämällä asiakkaan rajapinnasta tuotetunnus, josta pystyttiin selvittämään halutut tiedot vertaamalla niitä asiakkaan jakamiin tunnuksiin eri malleista.

Tuotetunnuksen erottelu haluttuihin arvoihin tapahtui pääluokassa käyttäen säännöllisiä lausekkeita, joilla voidaan muodostaa hakukaavoja, jotka mahdollistavat halutun tyyppisten tekstikohteiden löytämisen (15). Muiden ominaisuuksien haku tapahtui vertaamalla tunnusta ennalta tunnettuihin teksteihin.

Regular expressions 101 -sivusto (16) oli erittäin tehokas työkalu näiden säännöllisten lauseiden suunnittelemiseen, jonka näkee hyvin kuvasta 12.



KUVA 12.Esimerkki säännöllisesti lausekkeesta, joka hakee kahdeksan kirjaimen tai numeron yhdistelmää, joka alkaa pisteellä ja päättyy joko välilyöntiin, kauttaviivaan tai tyhjään käyttäen Regular expressions 101-sivuston editoria

Yhteyden sulkeminen vaatii, että järjestelmä lähettää sulkemispyynnön rajapintaluokalle, jossa kommunikointi tapahtuu säikeellä. Säikeet pysyivät hengissä niin kauan, kuin luokassa määritellyn elossaolomuuttujan arvo oli tosi. Kun muuttujan arvo vaihtua epätodeksi, säikeen lopussa varmistettiin, että fyysinen yhteys katkaistiin nykyiseen rajapintaan. Käytössä olleiden muuttujien arvot palautettiin niiden lähtöarvoihin, jotta uuden yhteyden muodostaminen olisi mahdollista.

Luokan tiedonhakumetodin tarkoitus oli hakea mahdollisimman paljon informaatiota yhteen kokoelmaan, jotta yksittäisiä metodeja varten ei tarvitse tehdä useita kutsuja. Metodista päätettiin palauttaa sanakirjaolio, johon pystyi tallentamaan erilaisia tietoja avain-arvopareina. Sisäisesti metodi kutsuu

asiakasrajapinnasta useita eri tiedonhakumetodeja ja tallentaa niiden tuloksen yllä mainittuun sanakirjaolioon.

3.4.2 Tuotekohtaiset ominaisuudet

Tässä työssä ei tulla käymään tuotekohtaisia metodeja tarkkaan läpi, vaan käytössä on esimerkin omaisia metodeja.

Kuvassa 13 näkyy esimerkki rajapintametodin käytöstä ja siitä voidaan todeta, että lähetettävien viestien syntaksi on yksinkertaista ja siinä vaihtuu ainoastaan halutun testikategorian koodi sekä siihen liittyvä viesti.

```
@require_connected
def perform_product_specific_operation(self) -> str:
    self._interface.send_test_request(
        0x1, # test command category
        'data_string',
        self._active_slave,
        self._test_request_callback,
        self._test_fail_callback
    )
    return self._wait_for_callback_result()
```

KUVA 13. Koodiesimerkki rajapintaluokan käytöstä halutun rajapintametodin ajamiseksi.

Muutama asia, mitkä on hyvä ottaa huomioon tästä esimerkistä, ovat koristajien käyttö halutun tyyppisen ohjelmarajoitusten asettamiseksi ja yksiselitteinen lähetys sekä luku syntaksi.

Connected-koristaja varmistaa, että tuotekohtaisia tai muita rajapintaluokkaan sidottuja komentoja ei voida suorittaa ilman, että järjestelmä on yhteydessä asiakkaan tuotteeseen. Koristajan toiminta perustuu olion ominaisuuden vertaamiseen tosiarvoon ja mikäli ehto ei täyty, nostetaan määritelty poikkeus virheen merkiksi.

Tuotekohtaisten metodien tärkeä osa on käsitellä vastauksien data halutulla tavalla. Lähetys metodi käsittelee vastauksen normaalisti siten, että se palauttaa vastauksen sisällön tekstimuodossa ja se täytyy jatko käsitellä saadakseen halutun vastauksen.

Mikäli datan sisältöä ei voida käsitellä tekstimuodossa, lähetysmetodille voidaan myös antaa vaihtoehtoinen vastauksen käsittelijä metodi, jolla voidaan palauttaa data ei tekstimuotoisessa formaatissa, mikäli se on tarpeen.

Yleisesti tuotekohtaisten metodien lopussa tapahtuu jokin yksinkertainen käännös tekstimuodosta joksikin halutun tyyppiseksi vastaukseksi. Esimerkiksi monet vastaukset sisältävät totuusarvon kokonaisluku muodossa (1 tarkoittaa tosiarvoa ja 0 epätosiarvoa).

4 INTEGROINTI

Ennen kuin paketti oli saatu valmiiksi, testijärjestelmää varten täytyi kehittää testiskriptipohja toisia kehittäjiä varten. Ensimmäinen askel oli tutkia vanhojen projektien skriptejä ja ottaa niistä mallia, miten pakettia kuuluisi niissä käyttää.

4.1 Testiskriptien rakenne

Skripteissä määriteltiin omia testiluokkia, jotka perivät valmiin testiluokkapohjan. Testiluokkapohjan avulla käytettävät ajurit pystyttiin tarjoamaan skriptin eri osiin, jotta niitä voitiin käyttää ajamaan halutun tyyppisiä testioperaatioita. Projektin dokumentaatiossa oli kuvattu, minkä typpisiä testioperaatioita tuotteille piti suorittaa, joten niistä sai hyvän idean, minkä tyyppisiä testiskriptejä täytyi luoda.

Skriptipohjissa käytettiin kehitetyn paketin tuotekohtaisia metodeja riippuen siitä, mitä kyseissä skriptissä testattiin. Itse paketin metodien käytön lisäksi useisiin kohtiin jätettiin kommentteja, että tässä pitäisi ajaa jokin eri operaatio käyttäen toista ajuria, joka oli kuvattu projektidokumentaatiossa. Esimerkki skriptipohjasta alla olevassa kuvassa 14.

```
import time
from tester import TestCase


class ValidationTest(TestCase):
    name = "Validation test"

    def runTest(self, params):
        self.logger.debug('Executing %s', self.name)
        nameplate = self.dut.read_nameplate()
        # TODO > Compare nameplate with expected value
        sw_version = self.dut.read_software_version()
        os_version = self.dut.read_os_version()
        # TODO > Compare os and sw version to expected versions
```

KUVA 14. Koodiesimerkki rajapintaluokan käytöstä testijärjestelmän testiskriptissä

Itse testiskriptipohja jäi suhteellisen pieneksi toteutukseksi, koska tuotemetodit olivat yksinkertaisia ja yksiselitteisiä, mutta skripteissä isompi osa oli lopun järjestelmän komentaminen ja tuloksien analysoiminen.

4.2 Testaus ja korjaus

Kun pohja oli saatu valmiiksi, projektin ohjelmistotiimi otti sen käyttöön oikean toteutuksen luomiseksi sekä ajurien testaamiseksi.

Skriptejä testatessa suurin osa paketin ominaisuuksista toimi odotetulla tavalla ja niitä ei tarvinnut muuttaa, mutta jotkin tietyt operaatiot eivät joko toimineet tai niiden toteutus haluttiin muuttaa erilaiseksi.

Osa muutoksista ei vaatinut muuta kuin paketin toteutuksen muuttamista halutunlaiseksi. Tietyt ominaisuudet olivat sidottuja asiakkaan rajapinnan toteutukseen, joita varten täytyi kommunikoida asiakkaan kehittäjille, että jonkin ominaisuuden toiminta täytyi muuttaa.

Tämä aiheutti monessa tapauksessa paljon viivettä kehityksessä ja satunnaisesti myös väärinymmärryksiä toiminnon muutoksesta.

5 YHTEENVETO

Opinnäytetyössä lähdettiin kehittämään Python-pakettia, jonka avulla pystyttiin suorittamaan testirutiineja ja hakemaan tietoja asiakkaan tuotteilta käyttäen RS485- tai socket-rajapintaa. Kehitettävä paketti saatiin valmiiksi ja sen avulla pystyttiin suorittamaan haluttuja rutiineja käyttäen asiakkaan tuotteita.

Suurin ongelma paketin kehityksessä oli sen aikataulutus. Paketin kehitys viivästyi paljon, koska suurin osa asiakasrajapinnan ominaisuuksista ei projektin alkuvaiheessa ollut saatavilla ja kommunikaatio kehittäjien kanssa oli usein melko hidasta. Tämä aiheutti myös viivettä testijärjestelmän skriptien jatkokehityksessä lopulla ohjelmistotiimillä, koska jos ajurien ominaisuudet eivät toimineet, ei pystytty testaamaan testien toimintaakaan. Tästä syystä projektin valmistuminen viivästyi.

Kehitetyn paketin ohjelmarakenne toimi mielestäni erittäin hyvin ja järjestelmän hajauttaminen rajapintaluokkaan ja pääluokkaan mahdollisti yksinkertaisen rajapinnan luomisen.

Ongelmiakin paketin kehityksessä oli, mutta ne keskittyivät suurimmaksi osaksi siihen, miten asiakkaan tuotteen kanssa kommunikoitiin. Datan käsittelyssä tapahtui aluksi paljon virheitä, mutta paketin kehittyessä se kehittyi tasapainoisemmaksi ja alkoi toimia pääasiassa ilman virheitä.

Yksi isoimmista ongelmista RS485-rajapinnassa oli se, että alkuvaiheessa ei huomattu, että viestin datassa pystyi olemaan mukana lopetustavu. Tämän takia osaa vastaanotetuista viesteistä ei tunnistettu kokonaiseksi aikaisen päättymisen vuoksi.

Socket-rajapinnan kehityksessä alku oli haastavin, koska näytti siltä, että tuote ei vastaanota lähetettyjä viestejä, vaikka itse socketti näyttäisi toimivan. Lopulta ongelma ratkesi sillä, kun alettiin keskustelemaan viestien terminoimisesta asiakkaan kanssa. Sen seurauksena todettiin, että heidän dokumentaationsa oli puutteellinen ja siinä ei mainittu tarvittavien loppumerkkien lähettämistä viestien perään.

Tämän puutteen korjauksen jälkeen itse kommunikoinnissa ei ilmennyt paljon ongelmia, mutta lähetettyjen viestien formaattia vaihdettiin kerran. Tämä siksi, että voitaisiin varmistua datan perille tulemisesta paremmin, koska socket-rajapinnan viestiformaatti ei sisältänyt vastaavia aloitus- ja lopetusmerkkejä tai viestin pituutta kuin RS485-rajapinta.

Yksittäisiä korjauksia pakettiin jouduttiin tekemään ohjelmavirheiden vuoksi tai uusien ominaisuuksien lisäämiseksi, mutta mitään suurta struktuurimuutosta ei tarvinnut tehdä.

Kaiken kaikkiaan paketin kehityksessä onnistuttiin haluttujen funktioiden luomisessa. Paketti osasi yhdistyä automaattisesti asiakkaan tuotteeseen, suorittaa haluttuja tuotekohtaisia metodeja sekä siivota jälkensä uutta yhteydenottoa varten käytön jälkeen. Paketti otettiin testausjärjestelmässä käyttöön ja sillä ajetaan edelleen testausoperaatioita asiakkaan tuotteilla.

Tämä projekti oli erittäin mielenkiintoinen ja se tarjosi hyviä oppimismahdollisuuksia, joita voi hyödyntää tulevaisuudessa vastaavia järjestelmiä suunnitellessa.

LÄHTEET

1. Stiller, Bart 2015. RS-485 basics: Introduction. Texas Instruments. Saatavissa: https://e2e.ti.com/blogs_/b/industrial_strength/archive/2015/04/28/rs-485-basics-introduction. Hakupäivä 5.11.2020.

2. Network socket. 2020. Wikipedia. Saatavissa: https://en.wikipedia.org/wiki/Network_socket. Hakupäivä 5.11.2020

3. Optofidelity 2020. OptoFidelity FUSION. Saatavissa: https://www.optofidelity.com/offering/products/fusion. Hakupäivä 5.10.2020.

4. Azevedo, Mariana 2019. S.O.L.I.D principles: what are they and why projects should use them. Medium. Saatavissa: https://medium.com/@mari_azevedo/s-o-l-i-d-principles-what-are-they-and-why-projects-should-use-them-50b85e4aa8b6. Hakupäivä 7.10.2020.

5. Liechti, Chris 2020. Short introduction. Pyserial. Saatavissa: https://pyserial.readthedocs.io/en/latest/shortintro.html. Hakupäivä 8.10.2020.

6. Python Software Foundation 2020. Threading – Thread based parallelism. Saatavissa: https://docs.python.org/3.7/library/threading.html. Hakupäivä 9.10.2020.

7. Python Software Foundation 2020. Socket – Low-Level networking interface. Saatavissa: https://docs.python.org/3.7/library/socket.html. Hakupäivä 9.10.2020.

8. Shaw, Anthony 2018. Getting Started With Testing In Python. Real Python. Saatavissa: https://realpython.com/python-testing/. Hakupäivä 9.10.2020.

9. Python Software Foundation 2020. pip 20.2.3. PyPI. Saatavissa: https://pypi.org/project/pip/. Hakupäivä 13.10.2020.

10. Python Software Foundation 2020. Virtual Environments and Packages. Saatavissa: https://docs.python.org/3/tutorial/venv.html. Hakupäivä 27.10.2020.

11. Marchi, Mirco de 2020. What is CRC? Medium. Saatavissa: https://medium.com/@mircodemarchi/what-is-crc-8e96f3f7e2ef. Hakupäivä 14.10.2020.

12. Python Software Foundation 2020. pycrc 0.9.2. PyPI. Saatavissa: https://pypi.org/project/pycrc/. Hakupäivä 14.10.2020.

13. Python Software Foundation 2020. crcmod 1.7. PyPI. Saatavissa: https://pypi.org/project/crcmod/. Hakupäivä 14.10.2020.

14. Python Software Foundation 2020. libscrc 1.5. PyPI. Saatavissa: https://pypi.org/project/libscrc/. Hakupäivä 15.10.2020.

15. Sturtz, John 2020. Regular Expressions: Regexes in Python (Part 1). Real Python. Saatavissa: https://realpython.com/regex-python/. Hakupäivä 26.10.2020.

16. Dib, Firas 2020. Regular expressions 101. Saatavissa: https://regex101.com/. Hakupäivä 27.10.2020.