Lauri Salminen

PROSESSIEN VÄLISEN KOMMUNIKOINNIN ANALYSOINTI

Opinnäytetyö

Tekniikan ammattikorkeakoulututkinto

Peliohjelmoinnin koulutus

2023

Kaakkois-Suomen ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tutkintonimike	Insinööri (AMK)
Tekijä/Tekijät	Lauri Salminen
Työn nimi	Prosessien välisen kommunikoinnin analysointi
Toimeksiantaja	Oy Veikkaus Ab
Vuosi	2023
Sivut	47 sivua
Työn ohjaaja(t)	Niina Mässeli

TIIVISTELMÄ

Opinnäytetyön tarkoituksena oli kehittää visualisointityökalu yrityksen tietokoneohjelmiston prosessien välisen kommunikoinnin (IPC) analysoimista varten. Yrityksen ohjelmisto koostuu kymmenistä eri prosesseista, jotka kommunikoivat keskenään käyttäen prosessien välistä kommunikointia. Kommunikointia tehdään paljon ja se on suurelta osin näkymätöntä. Työn tavoitteena oli kehittää ohjelmistoon työkalu, jota käyttämällä tämä liikenne saadaan näkyväksi. Valmista visualisointityökalua on tarkoitus käyttää ohjelmiston optimointiin sekä prosessien välisten riippuvuuksien analysointiin.

Visualisointityökalu suunniteltiin koostumaan kahdesta erillisestä ohjelmasta: ohjelmiston IPC-liikenteen tapahtumien keruuohjelmasta sekä keruuohjelmalla kerättyjen tapahtumien visualisointiohjelmasta. Ohjelmiston IPC-liikenteestä poimittu data lähetettiin prosesseilta erilliseen datankeruuohjelmaan. Tätä varten tutkittiin UNIX- sekä Linux-käyttöjärjestelmistä löytyviä IPC-menetelmiä. Tutkituista menetelmistä D-Bus-niminen IPC-järjestelmä valittiin datan lähettämiseen datankeruuohjelmalle. Visualisointiohjelma käyttää graafivisualisointia prosessien välisten riippuvuuksien esittämiseen ja sitä täydentää taulukko yksittäisten tapahtumien esittämiseen. D-Bus-järjestelmää käytettiin Qt Framework -ohjelmistorakenteen D-Bus-kirjastolla ja graafin luomiseen sekä esittämiseen käytettiin cytoscape.js JavaScript -kirjastoa.

Työn tuloksena ohjelmistoon valmistui visualisointityökalu, joka koostuu datankeruuohjelmasta sekä visualisointiohjelmasta. Datan lähettäminen prosesseilta datankeruuohjelmalle D-Bus-järjestelmällä osoittautui hyväksi valinnaksi, ja visualisointiin käytetty graafi sekä taulukko muodostivat hyvin toimivan kokonaisuuden ohjelmiston IPC-tapahtumien sekä riippuvuuksien esittämiseen. Molemmat ohjelmat muodostivat kokonaisuuden, josta on merkittävä apu yrityksen ohjelmiston jatkokehityksessä.

Asiasanat: Linux, IPC, prosessien välinen kommunikaatio, visualisointi, graafi, ohjelmistokehitys, cytoscape, D-Bus, Qt D-Bus

South-Eastern Finland University of Applied Sciences

Degree title	Bachelor of Engineering
Author (authors)	Lauri Salminen
Thesis title	Inter-process communication analysis
Commissioned by	Oy Veikkaus Ab
Time	2023
Pages	47 pages
Supervisor	Niina Mässeli

ABSTRACT

The purpose of the thesis was to develop a visualization tool to analyze inter-process communication (IPC) in the company's software. The software consists of tens of processes that communicate with each other using inter-process communication. There is a large amount of this type of communication, and it is largely invisible. The purpose of this work was to develop a tool that makes this communication visible. The intended purpose of the resulting tools is to optimize the software and analyze dependencies between the processes.

The visualization tool was designed to consist of two separate programs. The first program was a data collector of IPC traffic events. The second program was a visualization program that visualized the collected events. The data from the IPC traffic was sent from the processes to a separate data collector program. This required research on the different IPC methods present in Linux and UNIX operating systems. Based on the research, D-Bus was selected as the method to send data to the data collector. The visualization program uses graph visualization to display dependencies between different processes. It also includes a table to show individual events. The D-Bus system was used with the D-Bus library of the Qt Framework, and cytoscape.js JavaScript library was used for creating and presenting the graph.

As a result of the thesis, a visualization program was developed for the software. The program consists of a data collector program and a visualization program. Sending data from the processes to the data collector with D-Bus proved to be a suitable choice. The combination of graph and table visualizations formed a well-functioning solution to display the IPC events occurring in the software and to show dependencies between the processes. Both programs created an ensemble that can offer significant help with the future development of the company's software.

Keywords: Linux, IPC, inter-process communication, visualization, graph, software development, cytoscape, D-Bus, Qt D-Bus

SISÄLLYS

1 JOHDANTO

Prosessien välisellä kommunikaatiolla (engl. Inter-process communication, IPC) tarkoitetaan eri tietokoneohjelmien välillä tapahtuvaa keskustelua. Keskustelussa prosessit lähettävät toisilleen viestejä, jotka voivat sisältää hyvin vaihtelevaa informaatiota riippuen käyttötarkoituksesta. Isoissa ohjelmistoissa kommunikaatiota voi tapahtua paljon ja sen suurta määrää voi olla vaikea hahmottaa.

Tämän opinnäytetyön tarkoituksena on kehittää datankeruujärjestelmä tilaajayrityksen ohjelmiston IPC-menetelmään ja analysoida sillä kerättyä dataa. Samalla tutkitaan muita IPC-menetelmiä ja työkalun kehittämistä datan esittämiselle. Yrityksen ohjelmisto koostuu kymmenistä eri prosesseista ja niiden välillä tapahtuu paljon kommunikaatiota. Kommunikaatio on kuitenkin suurelta osin näkymätöntä ja sitä tehdään useista eri sijainneista. Tämän takia kommunikaatiota olisi hyvä visualisoida jollain tavalla, jotta viestiliikenteestä saadaan näkyvää ja ihmiselle ymmärrettävää. Aihe valittiin yrityksen tarpeesta kommunikoinnin visualisointityökalulle sekä sen kiinnostavauuden takia.

Työn tilaaja on Veikkaus Oy, joka on Suomessa toimiva rahapeliyhtiö. Veikkaus Oy:llä on yksinoikeus Suomen rahapelimarkkinoihin. Veikkauksen pelivalikoimaan kuuluu niin verkossa pelattavia kasino- ja arpapelejä kuin kasinoilla pelattavia raha-automaatteja ja pelipöytiä. Yritys järjestää yksinoikeudella Suomessa pelattavat lottopelit ja vedonlyönnit. Veikkaus valmistaa myös omia raha-automaatteja Suomen markkinoille.

Työ mahdollistaa yritykselle datan visualisointityökalun kehittämisen, jonka avulla voidaan seurata liikennettä prosessien välillä. Liikenteen analysoinnilla voidaan poistaa turhia riippuvuuksia prosessien väliltä sekä vähentää liikennettä. Työkalu helpottaa myös uusia työntekijöitä tutustumaan ohjelmistoon ja auttaa tapahtuvan liikenteen ymmärtämisessä.

Työ tulee perustumaan suurelta osin prosessien välisen kommunikaation tutkimiseen, joka on laajasti käsitelty aihe tietotekniikassa. Vaikka tietoliikenteen visualisointiin löytyy jo valmiiksi erilaisia ratkaisuja, prosessien välisen kom-

munikaation visualisointiin ei kuitenkaan ole olemassa valmiita työkaluja. Kerätyn sekä reaaliaikaisen datan esittäminen jo olemassa olevaan tietoliikenteen visualisointityökaluun vaatii integraation kehittämisen.

2 TUTKIMUSASETELMA

Työn tarkoituksena on kehittää yrityksen ohjelmistoon prosessien välisen kommunikaation visualisointityökalu. Ohjelmistoon kehitetään datankeruujärjestelmä ja sillä kerätty data tullaan visualisoimaan työkalulla. Tätä dataa hyödyntäen on mahdollista optimoida prosessien välisiä riippuvuuksia ja viestiliikenteen määrää. Tutkimusongelma on ohjelmiston prosessien välisen kommunikaation analysointi, esittäminen sekä optimointi.

Tästä johdetut tutkimuskysymykset ovat:

1. Kuinka IPC-liikennettä voidaan visualisoida?
2. Onko IPC-liikenteen visualisoinnista hyötyä optimoinnin kannalta?
3. Miten dataa voidaan kerätä IPC-liikenteestä?

Työssä käsitellään prosessien välistä kommunikaatiota laajamittaisesti. Tutkimukseen lukeutuu myös Linux-käyttöjärjestelmässä käytössä olevat IPC-menetelmät, niiden erot ja edut. Tutkittavista IPC-menetelmistä tullaan keräämään tietoa ja niitä vertaillaan keskenään. Työn kehitysympäristönä toimii Linux-käyttöjärjestelmä ja työ tehdään sen asettamien rajoitusten mukaisesti.

Kehitettävässä toteutuksessa dataa tullaan keräämään useilta prosesseilta yhteen keskitettyyn paikkaan, josta data siirretään sitä käsittelevään työkaluun. Tässä toteutuksessa joudutaan käyttämään yhtä IPC-menetelmää datan keräämiseen näiltä prosesseilta. Menetelmää voidaan mahdollisesti käyttää myös keskitetyn datan siirtämiseen sitä käsittelevään työkaluun. Tämän takia on tarvetta myös selvittää mikä menetelmä soveltuu näihin tarkoituksiin parhaiten.

Työssä tutkitaan myös erilaisia menetelmiä kerätyn datan esittämiseen ja esitellään eri tapoja näyttää dataa, esimerkiksi graafeja sekä muita tietoverkkojen liikenteen esittämistapoja. Oikeanlaisen esittämistavan valinta on tärkeää ja sen täytyy myös vastata tarvetta. Data täytyy esittää niin, että siitä on hyötyä IPC-liikenteen optimoinnissa ja sen tulisi olla selkeästi luettavaa.

Aihe sivuaa jonkin verran tietotekniikan käsitettä thread safety, koska dataa tullaan keräämään usealta prosessilta saman aikaisesti, mutta prosessit eivät saa kirjoittaa esimerkiksi yhteen tiedostoon samaan aikaan (Ballman 2014). Monet IPC-menetelmät tarjoavat tähän ratkaisuja, mutta yksinkertaisissa ratkaisuissa tämä täytyy huomioida itse.

Työssä kehitetään yrityksen ohjelmistoon kaksi uutta ohjelmaa, jotka muodostavat visualisointityökalun. Tämän takia tutkimusotteeksi valittiin kehittämistutkimus, joka on laadullisen ja määrällisen tutkimuksen yhdistelmä. Kehittämistutkimuksessa tähdätään johonkin muutokseen, esimerkiksi tuotteen kehittämiseen. (Kananen 2019, 81.) Työn alussa tullaan keräämään aineistoa tutkimuskohteista. Aineistoa tietoliikenteen visualisoinnin esittämistavoista kerätään lähinnä verkkolähteistä. Prosessien välisestä kommunikaatiosta aineisto kerätään tutkittavien menetelmien dokumentaatiosta, kirjallisuudesta sekä verkkolähteistä. Kenttämuistiinpanot ovat pääasiallinen aineiston keruun menetelmä. Kehittämistyö ja sen vaiheet kirjataan tarkasti ylös kehityksen aikana. Näiden avulla saadaan tarkka kuva työstä, sen etenemisestä ja saaduista tuloksista.

Työn luotettavuus tulee olemaan lähtökohtaisesti hyvä. Lähteinä käytetään lähinnä työssä käytettävien teknologioiden virallisia dokumentaatioita. Muiden lähteiden sisällöt tarkistetaan myös dokumentaatiosta, jos vain mahdollista. Työssä toteutettua järjestelmää tullaan myös testaamaan sen toiminnan varmistamiseksi.

3 PROSESSIEN VÄLINEN KOMMUNIKAATIO

Prosessien välisellä kommunikaatiolla eli IPC:llä tarkoitetaan useiden prosessien välillä tapahtuvaa koordinointia sekä yhteistyötä. Yleinen esimerkki IPC:n tarpeesta on jaetun resurssin käytön hallinta. Kommunikaatio useiden prosessien välillä on olennaista monissa ohjelmistoissa. Tällä kommunikaatiolla on ollut pitkään keskeinen rooli UNIX-pohjaisissa käyttöjärjestelmissä, kuten Linuxissa. (KDE s.a.)

UNIX on Ken Thompsonin vuonna 1969 kehittämä käyttöjärjestelmä, joka kehitettiin AT&T-yrityksen Bell Laboratories -nimisessä yksikössä (Kerrisk 2010,

6,7). Linux on ilmaisjakelussa oleva klooni UNIX-käyttöjärjestelmästä, jonka kehitys alkoi Linus Torvaldsin toimesta vuonna 1991. Tätä edelsi GNU-projekti, joka tuotti UNIX-tyyppisille käyttöjärjestelmille useita ohjelmia, mutta ei saanut valmiiksi niiden ajamista mahdollistavaa ydintä. (Kerrisk 2010, 6.) Linux-kernel on Linux-käyttöjärjestelmän ydin, joka kehitettiin paikkaamaan tämä puute. Käsitettä Linux käytetään tarkoittamaan ohjelmistoympäristöä Linux-ytimellä, johon kuuluu useita komponentteja. (Siever ym. 2005, 1.) Linuxista on saatavilla useita eri jakeluja, jotka pakkaavat yhteen pakettiin Linux-ytimen ja useita valikoituja sovelluksia. Näihin kuuluu muun muassa vuonna 1993 julkaistu Debian ja siihen pohjautuva vuonna 2004 julkaistu Ubuntu. (Kerrisk 2010, 8.)

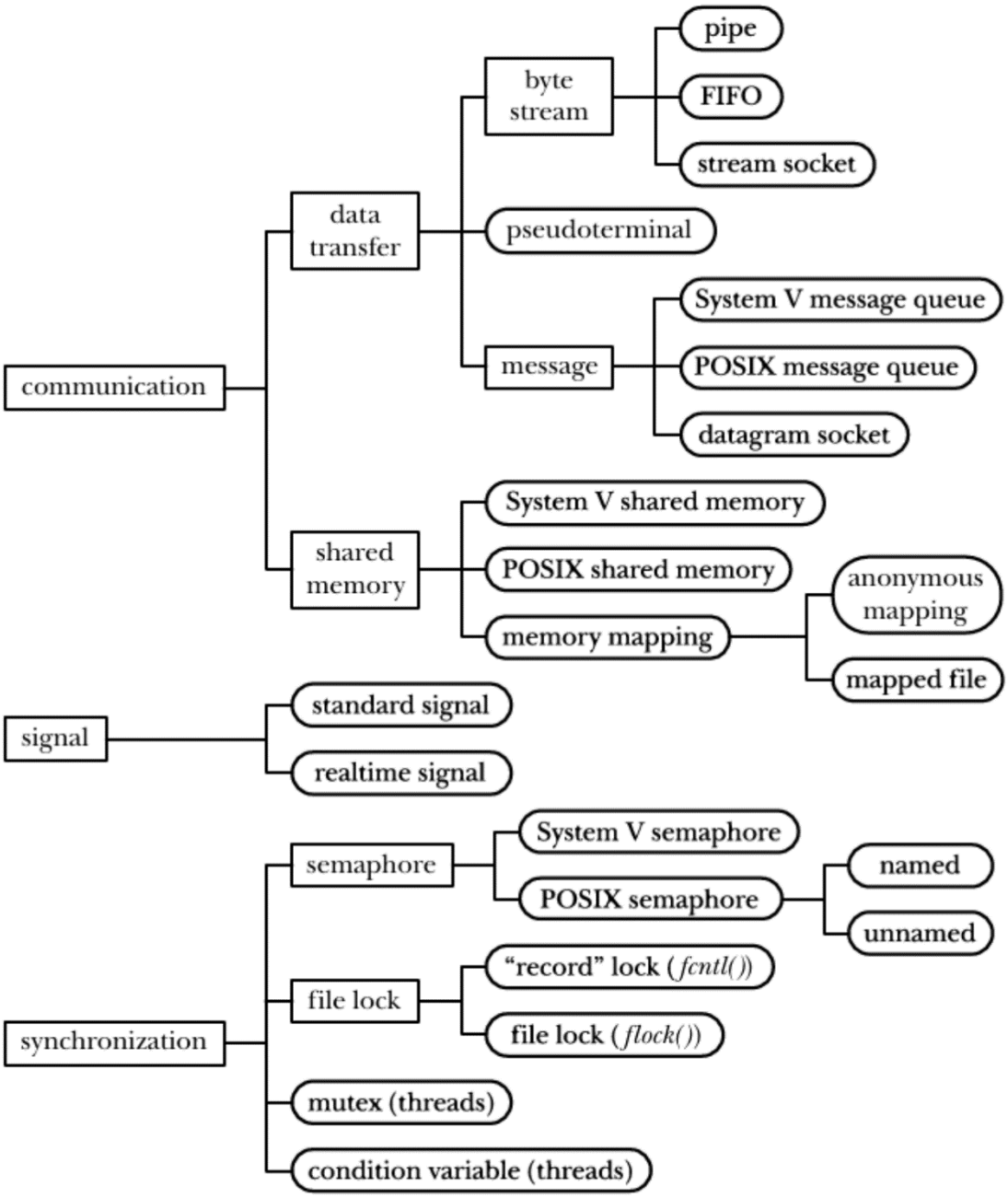
IPC:tä käytetään usein suljettujen järjestelmien kehittämisessä, koska niissä hyödytään useista prosesseista. Moniajosta hyötyvät järjestelmät voivat käyttää IPC-mekanismeja esimerkiksi suorituskyvyn parantamiseen. Suoritustehoa on helpompi jakaa eri prosessien välillä sen sijaan, että ohjelmaa ajettaisiin vain yhdellä prosessilla. (Kalin s.a., 24.) Yksinkertaisimmillaan IPC voi tarkoittaa tietokoneen leikepöytää, kun kopioidaan esimerkiksi tekstiä ohjelmasta ja liitetään se toiseen (Microsoft 2021).

Käsitettä IPC käytetään usein kuvastamaan kolmea eri kategoriaa: kommunikaatio, synkronisaatio sekä signaalit. Kommunikaatio käsittää prosessien välisen tiedon vaihdon ja synkronisaatio prosessien toimintojen synkronoinnin. (Kerrisk 2010, 877.) Signaalien avulla voidaan informoida prosessille, että jokin tapahtuma on tapahtunut. Ne ovat laitteistokeskeytysten kanssa samantyyppisiä, eli niiden avulla voidaan keskeyttää ohjelman normaali toiminta eikä niiden saapumista yleensä voida ennustaa. Niitä usein kutsutaankin ohjelmistokeskeytyksiksi. Esimerkkejä signaaleista on ohjelman keskeytys Ctrl-C-näppäinyhdistelmällä tai ohjelman ikkunan koon muuttaminen. (Kerrisk 2010, 388.) Signaaleja voidaan käyttää myös synkronointiin tai harvinaisemmin, kommunikaatioon (Kerrisk, 877).

Tämän työn yhteydessä käsitellään Linux-käyttöjärjestelmässä käytössä olevia IPC-menetelmiä. Menetelmiin lukeutuvat tulevissa luvuissa käsiteltävät jaettu data tiedostosta, jaettu muisti, stream socket sekä putki. Myös uudempaa

D-Bus IPC -järjestelmää ja sen käyttöä helpottavia kirjastoja käsitellään laajemmin omassa luvussaan.

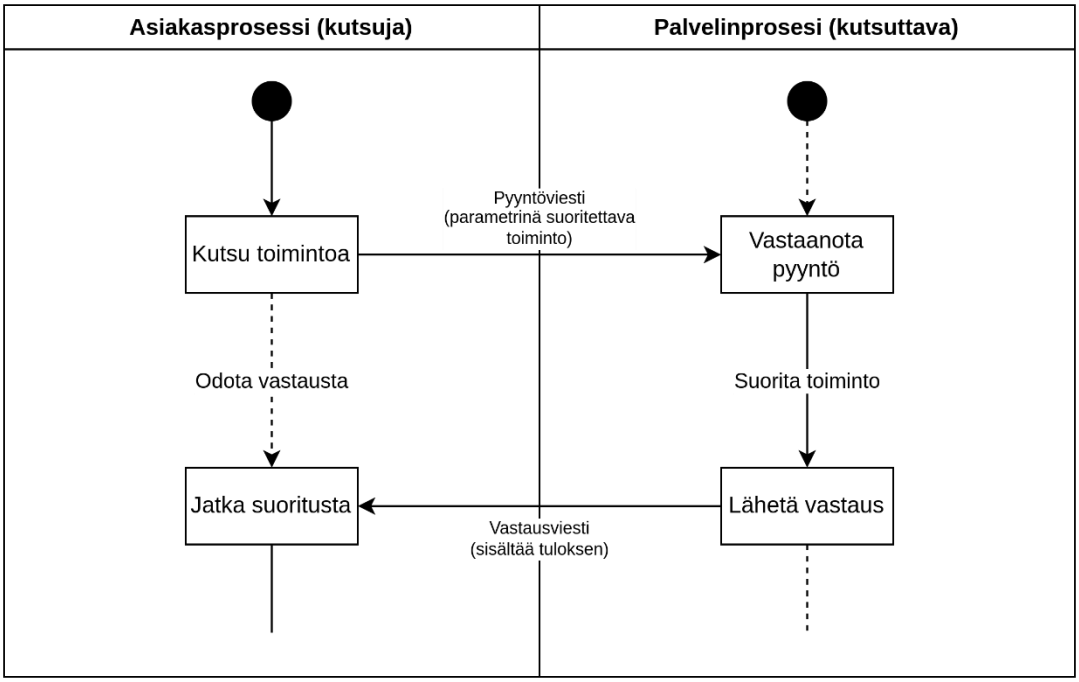
UNIX-käyttöjärjestelmien eri IPC-toimintojen taksonomia on nähtävissä kuvasta 1. Kuvasta on nähtävissä aiemmin esitellyt kolme IPC-kategoriaa: kommunikaatio (communication), signaali (signal) ja synkronisaatio (synchronization). Osassa toiminnoista on päällekkäisyyksiä, esimerkiksi POSIX sekä System V toteuttavat molemmat samoja toimintoja. Tämä johtuu siitä, että uusia toimintoja on kehitetty edellisten toimintojen puutteiden takia. POSIX IPC -toiminnot on kehitetty parannukseksi System V IPC -toiminnoille. Tiedonsiirrosta löytyvät putki (pipe), FIFO ja stream socket ovat kehitetty eri järjestelmille, jotka myöhemmin lisättiin myös muihin UNIX-järjestelmiin. Vaikka nämä kuuluvat samaan kategoriaan, niiden toiminnasta löytyy merkittäviä eroja toistensa kanssa. Esimerkiksi FIFO-menetelmää voidaan käyttää vain samalla laitteella olevien prosessien kanssa, kun taas stream socket -menetelmää voidaan käyttää tietoverkon yli. (Kerrisk 2010, 878,879.)



Kuva 1. UNIX IPC -taksonomia (Kerrisk 2010, 878)

Näiden lisäksi monet Linux-jakelut käyttävät nykyään D-Bus-nimistä IPC- ja RPC-menetelmää. Se kehitettiin korvaamaan nykyiset ja kilpailevat IPC-menetelmät yhdellä yhdistetyllä protokollalla. (Qt D-Bus Overview s.a.) RPC eli remote procedure call -käsitettä tullaan käyttämään hieman D-Bus-järjestelmän käsittelyn yhteydessä. RPC-kutsulla asiakasohjelma suorittaa palvelinohjelmassa toiminnon ja saa siihen tarvittaessa vastauksen. Se toimii kuten paikallisesti suoritettava toiminto, mutta toiminto suoritetaan ulkoisessa palvelimessa tai toisessa prosessissa. (Perterson & Davie 2020, luku 5.3.)

Kuva 2 esittää RPC-mallin toimintaa. Vasemmassa osiossa on asiakasprosessi, joka lähettää pyynnön toiminnon suorittamista varten. Oikeassa osiossa on palvelinprosessi, joka käsittelee pyynnön ja suorittaa toiminnon. Asiakasprosessin lähetettyä pyynnön toiminnon suorittamista varten sen oma suoritus keskeytyy ja se jää odottamaan vastausta palvelimelta. Suoritettuaan toiminnon palvelin palauttaa vastausviestissä tuloksen asiakasprosessille, joka pääsee jatkamaan omaa suoritustaan.



Kuva 2. Esimerkki RPC-mallista

RPC-mallia käytetään paljon hajautetuissa järjestelmissä tietokoneiden sekä palvelimien välillä. Se siirtää suoritettavan toiminnon toiselle prosessille, joka voi sijaita samassa järjestelmässä tai verkon yli käytettävässä ulkoisessa järjestelmässä. (Perterson & Davie 2020, luku 5.3.)

3.1 Jaetun resurssin käyttö

Tietotekniikan käsite atomisuus, joka tunnetaan myös linearisointina, takaa resurssin eristyksen muista samaan aikaan ajettavista operaatioista. Esimerkiksi useat Linux-käyttöjärjestelmän järjestelmäkutsut suoritetaan atomisesti. Ydin takaa, että kaikki suoritettavan operaation vaiheet valmistuvat ilman keskeytyksiä muilta prosesseilta. Atomisuus takaa tiettyjen operaatioiden onnistumi-

sen ja estää race condition -tilanteiden tapahtumisen. Race condition -tilanteessa kahden prosessin tulos jaetulla resurssilla riippuu arvaamattomasti siitä järjestyksestä, jolla ne saavat pääsyn tietokoneen prosessoriin (Kerrisk 2010, 90). Race condition -tilanne tulee esille monien IPC-menetelmien kanssa. Jaettaessa dataa tiedostosta tilanne tulee esille, kun useat prosessit lukevat tai kirjoittavat samaan tiedostoon (Kerrisk 2010, 92). Jaettua muistia käyttävät menetelmät, kuten System V IPC estävät yhtäaikaisen käytön hallinnoimalla muistiin pääsyä (Kerrisk 2010, 965).

Synkronointijärjestelmillä yhtäaikainen pääsy tietueeseen voidaan estää ja siten estää race condition -tilanteiden tapahtuminen. Yksi synkronointijärjestelmä on mutex-lukko (mutual exclusion construct), jolla resurssi saadaan lukittua yhdelle prosessin säikeelle (thread) (Kerrisk 2010, 881). Moderneissa UNIX-käyttöjärjestelmissä yhdellä prosessilla voi olla useita säikeitä, joita se käyttää suoritukseen. Säikeet voi mieltää useina prosesseina, jotka jakavat saman muistialueen muiden attribuuttien lisäksi. Jokainen säie suorittaa samaa ohjelmaa ja jakaa saman muistin. Säikeillä on myös oma paikallinen muisti, mutta säikeiden keskeiseen kommunikointiin ne käyttävät globaalia jaettua dataa. (Kerrisk 2010, 92.) Kun data on lukittu mutex-lukolla, joutuvat toiset säikeet odottamaan lukon avausta ennen sen käyttämistä. Mutex-lukkoa on tietyissä tilanteissa mahdollista käyttää myös prosessien välillä. Tiedostoille on olemassa mutex-lukkojen tapaiset tiedostolukot, jotka toimivat niiden kanssa samankaltaisella periaatteella. Tiedostot ovat mahdollista lukita kirjoittamista tai lukemista varten. (Kerrisk 2010, 881). Tiedostolukoista kerrotaan tarkemmin kappaleessa 3.2.

Jaetun muistin IPC-menetelmissä voidaan käyttää synkronointiin semaforia. Semaforin voi mieltää vuokraamopalveluna, jossa on vain rajallinen määrä vuokrattavia tuotteita. Vuokrattujen tuotteiden määrää nostetaan vuokrattaessa ja palauttamisen jälkeen sitä lasketaan. Jos kaikki tuotteet ovat vuokrattuja, on seuraavan vuokraajan odotettava, kunnes joku palauttaa tuotteen. (Kalin s.a., 6.) Tietotekniikassa semaforin suurin sallittu määrä, olisi jaetun resurssin ilmentymien määrä. Semafori toteutetaan usein binäärisenä, jolloin käytössä ovat kaksi arvoa 0 ja 1 (Kerrisk 2010, 881). Näin käytettäessä semafori toimii samankaltaisesti kuin mutex-lukko, jossa arvo 0 tarkoittaa, että

muisti on varattu prosessin käyttöön. Käytön jälkeen käyttäjä asettaa arvon numeroon 1 ja seuraava prosessi saa pääsyn muistiin. (Kalin s.a, 6.)

3.2 Jaettu data tiedostosta

Tässä yksinkertaisessa IPC-menetelmässä toiselle prosessille tarjotaan tietoa tietokoneelle tallennetun tiedoston avulla. Ensimmäinen prosessi kirjoittaa tiedostoon dataa ja toinen prosessi lukee sen tiedostosta. Tämä on mahdollisesti kaikista alkeellisin IPC-menetelmä. (Kalin s.a., 6.)

Tiedostoja käytettäessä on otettava huomioon niiden mahdolliset ongelmat. Tiedoston olemassaolo on tarkistettava ennen sen käsittelemistä, joka voi olla ongelma sekä kirjoittamis- että lukuvaiheissa. Tiedostossa on oltava myös riittävät käyttöoikeudet tiedoston käyttäjille. Lähtökohtaisesti tässä menetelmässä on kaksi osapuolta: kirjoittaja sekä lukija. Kirjoittajan tehtävänä on tiedoston puuttuessa luoda se ja kirjoittaa siihen dataa, josta toiset lukijaprosessit voivat sen käydä hakemassa. Lukija taas avaa tiedoston lukemista varten ja hyödyntää siitä löytyvää dataa omassa toiminnassaan. (Kalin s.a., 6.)

Tiedostoon kirjoittamisen kanssa tulee nopeasti vastaan ongelmat aiemmin käsitellyn race condition -tilanteen kanssa (3.1). Kahden prosessin kirjoittaessa samaan tiedostoon on mahdollista päätyä tilanteeseen, jossa molemmat yrittävät tehdä sitä saman aikaisesti. Tämä voi johtaa virheisiin ja ohjelman väärään toimintaan. (Kalin s.a., 6.) Tässä menetelmässä kyseisen ongelman voi välttää käyttämällä jotain synkronisaatiomenetelmää, kuten tiedostolukkoa. Lukkojen toiminta on käsitteellisesti helppo ymmärtää. Kun prosessin on tarve kirjoittaa tai lukea tiedostoa, asettaa se siihen lukon. Lukituksen aikana ainoastaan lukon asettanut prosessi voi käyttää tiedostoa. Tiedoston käytön jälkeen prosessi poistaa lukon ja muut prosessit saavat taas siihen pääsyn. (Kerrisk 2010, 1117,1118.)

Linux-käyttöjärjestelmä tarjoaa mekanismit tiedostojen lukitsemiseen flock()- ja fcntl()-järjestelmäkutsujen kautta. Koko tiedosto lukitaan flock-kutsulla ja osittaista lukitsemista tehdään fcntl-kutsulla. Lukot voidaan asettaa joko varoittavaksi tai pakolliseksi. Varoittava lukko ei estä toista prosessia ohittamasta sitä, kun taas pakollista lukkoa ei ole mahdollista ohittaa. (Kerrisk 2010, 1118,

1119.) Pakollista lukitsemista ei suositella nykyään käytettäväksi sen epävakauden takia ja Linux-käyttöjärjestelmän versiosta 4.5 eteenpäin se on ollut valinnainen ominaisuus. Ongelmana on esimerkiksi race condition -tilanteet, kun tiedostosta luetaan tai siihen kirjoitetaan lukon hakemisen aikana. Tulevaisuudessa ominaisuus on tarkoitus poistaa käyttöjärjestelmästä kokonaan. (fcntl(2) — Linux manual page 2021.)

Käytettäessä flock-kutsua tiedoston voi lukita vain varoittavalla lukolla. Tämän kutsun toteutustavat vaihtelevat UNIX-käyttöjärjestelmien välillä. Useat järjestelmät käyttävät sen pohjana fcntl-kutsua koko tiedoston muistialueelle puhtaan flock-kutsun sijasta. Puhdas flock-kutsu ei myöskään pysty toimimaan fcntl-kutsun kanssa. (flock(2) — Linux manual page 2021.) Tiedostojen osittainen lukitseminen fcntl-kutsulla mahdollistaa muiden prosessien käyttää tiedostosta yhtä osaa samalla, kun toinen prosessi käyttää toista. (Kerrisk 2010, 1124.)

Molemmat lukitusmekanismit tarjoavat tiedostojen lukitsemiseen kaksi tapaa: jaetun lukon ja eksklusiivisen lukon. Tiedostoa vain lukevat prosessit käyttävät jaettua lukkoa, jolloin useat eri prosessit voivat lukea tiedostoa samaan aikaan. Eksklusiivista lukkoa taas käyttävät prosessit, jotka kirjoittavat tiedostoon. (Kalin s.a., 8.)

3.3 Uudelleenohjaus UNIX-putken avulla

Putki eli pipe on vanhin ja yksi yleisimmistä UNIX-järjestelmien IPC-menetelmistä. Tätä menetelmää kuvaa "|"-merkki, ja siitä on käyttöjärjestelmän normaalikäytössäkin paljon hyötyä. Putken avulla ohjelman palauttama data voidaan uudelleenohjata toiseen ohjelmaan. (Kerrisk 2010, 890). Eräs yleinen esimerkki on tiedon hakeminen toisen ohjelman palauttamasta datasta. Esimerkiksi komennolla "cat" voidaan tulostaa tekstitiedoston sisältö komentoriviin. Yhdistämällä tämä komento putkella hakukomentoon "grep" voidaan tiedoston sisältä hakea tietty teksti. Tällä tavalla saadaan siirrettyä dataa ohjelmasta "cat" ohjelmaan "grep". Kuva 3 näyttää esimerkin teksti.txt tiedoston sisällön ohjaamisesta grep-ohjelmaan putken avulla.

```
$ cat teksti.txt | grep "haettava teksti"
```

Kuva 3. Tiedon siirto ohjelmasta ”cat” ohjelmaan ”grep” putken avulla

Menetelmää käytettäessä molemmat prosessit ovat yhdistyneet putkeen. Lähettävän prosessin ulostulo putken kirjoituspäähän ja vastaanottavan prosessin sisääntulo putken lukupäähän. Käytännössä kumpikaan prosessi ei ole tietoinen putkesta, vaan prosessit lukevat tai kirjoittavat samalla tavalla kuin tiedostoonkin. Putkeen syötetyllä datalla ei ole koko rajoituksia ja se luetaan samassa järjestyksessä kuin missä se on syötetty. Putkella on kuitenkin rajattu kapasiteetti, jota ytimen muisti ylläpitää. Jos sen kapasiteetti täyttyy, putkeen ei voida kirjoittaa ennen kuin lukija poistaa sieltä dataa. (Kerrisk 2010, 890,891).

UNIX-käyttöjärjestelmistä löytyy paljon putkea muistuttava IPC-menetelmä nimeltä FIFO. Menetelmän merkittävin ero putkeen on se, että se sijaitsee tiedostojärjestelmässä ja se avataan samalla tavalla kuin tiedosto. Tämä mahdollistaa kommunikaation toisistaan erillisten prosessien välillä, esimerkiksi asiakasprosessien sekä palvelinprosessin. (Kerrisk 2010, 906).

3.4 System V IPC

System V on vanha UNIX-käyttöjärjestelmän muoto, joka julkaistiin ensimmäistä kertaa vuonna 1983 BSD-järjestelmän rinnalle. Se syntyi yrityksen AT&T jakautumisen jälkeen kehitetyn System III -järjestelmän seuraajaksi. System V sisällytti itseensä useita BSD-järjestelmän ominaisuuksia ja monet kaupalliset toimijat käyttivät sitä heidän omien UNIX-toteutuksiensa alustana. (Kerrisk 2010, 4.) Tämän takia System V -järjestelmän IPC-menetelmät ovat käytettävissä useimmissa Linux-jakeluissa. System V IPC tarjoaa kolme eri IPC-mekanismia: viestijonot, semaforit sekä jaetun muistin.

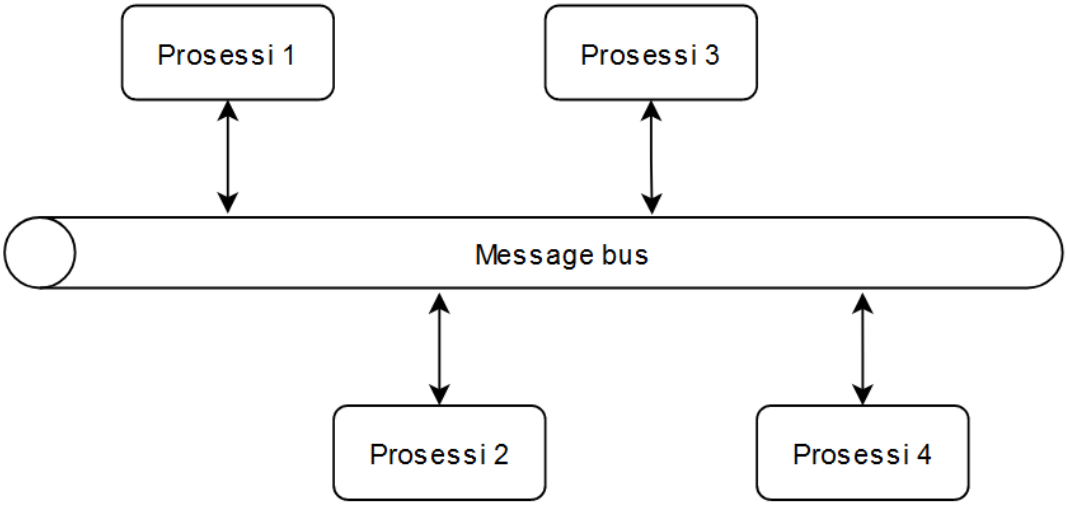
Viestijonoja käytetään viestien välittämiseen prosessien välillä. Ne toimivat putkien kanssa samankaltaisesti, mutta käyttävät rajattuja viesteiksi kutsuttuja yksiköitä, rajattoman tavuvirran sijasta. (Kerrisk 2010, 921.) Viesteille on mahdollista asettaa myös tärkeys. Tämä mahdollistaa viestien toimittamisen eri järjestyksessä kuin missä ne lähetettiin.

Jaettu muisti sallii saman muistialueen käytön useilla prosesseilla. Nopea pääsy muistiin tekee jaetusta muistista yhden nopeimmista IPC-menetelmistä. Prosessit jakavat samaa muistia, joten muistiin tehdyt muutokset ovat heti näkyvissä toisille prosesseille. (Kerrisk 2010, 921.) Koska System V sisältää jaettua muistia käyttävän IPC-menetelmän, on sen hallittava muistin käyttöä. Tämä järjestelmä hyödyntää kappaleessa 3.1 esiteltyä semaforia muistin hallintaan, jotta yhtäaikaista käyttöä ei tapahdu. (Kerrisk 2010, 965).

Nykyään System V IPC on vähäisessä käytössä ja sitä ei suositella käytettäväksi uusien sovellusten kehityksessä (Kerrisk 2010, 937). Sen sijasta on suositeltavaa käyttää esimerkiksi D-Bus IPC-menetelmää. Vuoden 2010 jälkeen käyttöön tullut D-Bus-niminen IPC-menetelmä, on nykyään käytettävissä useissa Linux-jakeluissa. Se kehitettiin korvaamaan lukuisat kilpailevat IPC-rakenteet yhdellä yhdistetyllä protokollalla. (Qt D-Bus Introduction s.a.)

3.5 D-Bus

D-Bus on message bus -järjestelmä, joka tarjoaa yksinkertaisen tavan tietokoneohjelmien väliseen keskusteluun. Se tarjoaa sekä järjestelmä- että käyttäjäkohtaisen taustalla ajettavan prosessin eli daemonin message bus -väylillä tapahtuville tapahtumille. (Freedesktop.com 2022.) Kuva 4 esittää message bus -viestinvälitysmekanismin toimintaa, jota myös D-Bus käyttää. Prosessit voivat esimerkiksi lähettää väylälle viestin, jonka kaikki siitä kiinnostuneet voivat lukea.



Kuva 4. Esimerkki message busista

Kommunikaatio toimii yksinkertaisen objektimallin mukaan, joka tukee RPC- sekä publish-subscribe-mekanismeja (Cocagne 2012). Ohjelmointimallissa publisher-subscriber eri palvelut voivat tilata viestityypin ja sen tapahduttua kaikki aiheen tilaajat vastaanottavat viestin (Pub/Sub Messaging s.a.). Tätä voi hahmottaa kuvan 4 avulla. Prosessit 1 ja 2 tilaavat viestin tyyppiä ”esimerkki”. Prosessi 4 lähettää ”esimerkki”-tyyppisen viestin message bus -väylälle, jonka jälkeen se välittyy sen tilanneille prosesseille 1 ja 2. D-Bus käyttää signaaleja tarjotakseen yhdeltä-monelle publish-subscribe-mekanismin (Cocagne 2012).

Väylälle yhdistäneet ohjelmat voivat hakea väylälle vietyjä objekteja, kutsua niistä metodeja ja pyytää tiedon niiden lähettämistä signaaleista. Objektien metodit ovat jonkin toisen ohjelman toimintoja, jotka yleensä palauttavat vastauksena tuloksen. Yksi väylälle yhdistänyt ohjelma voi tarjota palveluita useiden objektien kautta, joista jokaisella voi olla useita kutsuttavia metodeja. (Cocagne 2012.)

Silloin kun useilla prosesseilla on tarvetta keskustella useiden prosessien kanssa, käytetään D-Bus-väyliä. Tämä tarvitsee keskitetyn palvelimen käynnistämisen, ennen kuin ohjelmat voivat liittyä väylälle. Tämä palvelin pitää kirjaa liittyneistä ohjelmista ja viestien oikeasta välityksestä lähettäjältä vastaanottajalle. D-Bus tarjoaa kaksi ennalta määritettyä väylää: system bus ja session bus. Jos ohjelma tarvitsee esimerkiksi tietoa käyttäjän laitteistosta, se todennäköisesti kommunikoi system bus -väylältä löytyvän palvelun kanssa. Jos ohjelman olisi tarvetta avata esimerkiksi käyttäjän selain, siihen löytyvä palvelu voisi löytyä session bus -väylästä. (Qt D-Bus Overview s.a.)

Tietokoneohjelmat keskustelevat keskenään väylällä lähettelemällä viestejä toisilleen. Viestejä käytetään RPC-kutsujen, sekä niihin liittyvien vastausten ja virheiden välittämiseen. Väylällä voi olla esimerkiksi palvelu, joka tarjoaa sen kutsujalle tietoa jostain tietokoneen komponentista. Tietoa tarvitseva ohjelma lähettää tähän palveluun viestillä RPC-kutsun ja palvelu vastaa omalla viestillä ohjelmalle komponentin tiedot. Näin ohjelma kutsui metodia toisessa ohjelmassa, jonka tämä suoritti ja palautti sitten tuloksen alkuperäiselle ohjelmalle. (Qt D-Bus Overview s.a.)

Viesteille on mahdollista antaa vastaanottaja, kun niitä lähetetään väylän yli. Tämän takia ne välitetään ainoastaan viestistä kiinnostuneille ohjelmille ja siten vähennetään väylän ruuhkautumista. Väylillä on mahdollista käyttää myös viestiä nimeltä signal message. Tällaisilla viesteillä ei ole vastaanottajaa vaan niitä käytetään, kun yhden prosessin on tarvetta lähettää viesti useille prosesseille. Toisten prosessien on ilmoitettava kiinnostuksensa näitä viestejä kohtaan, jotta ne voivat vastaanottaa niitä. (Qt D-Bus Overview s.a.)

Väylällä keskustelevat ohjelmat saavat itselleen nimen, jota ne käyttävät erottuakseen toisistaan, tästä käytetään termiä service name. Näiden nimien avulla viestit välitetään yhdeltä ohjelmalta toiselle. Nämä voidaankin mieltää samankaltaisina kuin tietokoneiden IP-osoitteet ja niihin liitetyt isäntänimet, esimerkiksi ”www.google.com”. Jos väylää ei käytetä, niin nimiäkään ei käytetä, koska vertainen on tunnettu ja sitä ei tarvitse etsiä nimen avulla. Väylällä olevat nimet ovat usein nimetty organisaation verkkotunnuksen mukaan, esimerkiksi freedesktop.org-organisaation D-Bus-palvelu on nimeltään org.freedesktop.DBus. Ohjelmat tarjoavat eri palveluita toisille ohjelmille viemällä väylälle objekteja. Näiden polut muistuttavat tietokoneen tiedostojärjestelmän polkuja, esimerkiksi oheisen D-Bus palvelun metodi nimeltä Hello löytyy osoitteesta /org/freedesktop/DBus. (Qt D-Bus Overview s.a.)

D-Bus-järjestelmää varten on kehitetty useita sen käyttöä helpottavia kirjastoja. D-Bus-järjestelmän virallinen kirjasto on vaikeakäyttöinen ja sitä ei ole tarkoitettu käytettäväksi sellaisenaan. D-Bus-järjestelmän ylläpitäjät ehdottavatkin dokumentaatiossaan käyttämään jotain korkeamman tason kirjastoa, joka on kehitetty tämän kirjaston pohjalta. (Freedesktop.com 2022.) Kirjastoja käytetään D-Bus-järjestelmän ja sen väylien käyttöön.

3.5.1 sdbus-c++

Tämä on C++-ohjelmointikielelle kehitetty korkean tason D-Bus-kirjasto, joka käyttää C++17-ohjelmointikielestä löytyviä ominaisuuksia. Tämä yksinkertaistaa D-Bus-järjestelmän käyttämistä merkittävästi. Kirjasto kehitettiin korvaamaan vanhempi dbus-c++-kirjasto, joka tämän kirjaston ylläpitäjien mukaan

kärsii useista selvittämättömistä virheistä, ohjelmistosuunnitelman monimutkaisuudesta ja rajoituksista. (sdbus-c++ s.a.)

Kirjastoa voidaan käyttää kahdella eri tavalla: käyttämällä joko yksinkertaista mukavuuskerrosta tai C++-kielen mekanismeja käyttävää peruskerrosta. Mukavuuskerros tarjoaa yksinkertaisemman käyttökokemuksen ja piilottaa tarpeettomia yksityiskohtia käyttäjältä. (Using sdbus-c++ library s.a.)

3.5.2 Qt D-Bus

Qt on suomalainen ohjelmistoalan yritys, joka kehittää ohjelmistorakennetta nimeltä Qt Framework (The Future of Digital Experiences s.a.). Tähän rakenteeseen kuuluu eri kirjastoja sekä ohjelmointirajapintoja kuten Qt Core, Qt GUI ja Qt Network (Qt Features, Framework Essentials, Modules, Tools & Add-Ons s.a.). Qt D-Bus on Qt Framework -rakenteen D-Bus-kirjasto. Sitä käytetään D-Bus IPC -menetelmän käytön helpottamiseen. Qt sekä sen sisältämät kirjastot on mahdollista ottaa käyttöön C++-projekteissa CMake-työkalujen avulla. (Get started with CMake 2022.)

CMake-työkaluja käytettäessä C++-kehitykseen projekti määritetään CMakeLists.txt-nimistä tiedostoa käyttäen. Tiedostoon voidaan esimerkiksi määrittää ohjelman tarvitsemat kooditiedostot, kirjastot ja käytettävä C++-versio. Jotta Qt-rakenteen D-Bus-kirjastoa voidaan käyttää kehitettävässä ohjelmassa, on se haettava CMake-työkalujen paketinhakuominaisuudella. Tätä käytetään CMakeLists.txt-tiedostosta ja sille annetaan haettavan kirjaston nimi sekä siitä tarvittavat komponentit. Tässä tapauksessa haetaan Qt-kirjaston D-Bus-komponentti, jotta se saadaan ohjelmassa käyttöön. (Get started with CMake 2022.)

Tätä kirjastoa voidaan käyttää omien luokkien ja niistä löytyvien metodien rekisteröimiseen D-Bus-järjestelmän session- tai system-väylälle. Tässä yhteydessä luokalla tarkoitetaan rakennetta, joka voi sisältää useita metodeja. Metodit taas ovat prosessin eri toimintoja. Kun luokka rekisteröidään väylälle, on toisen prosessin mahdollista kutsua luokkaan määritettyjä metodeja.

Omia luokkia ja metodeja voidaan rekisteröidä väylälle käyttämällä Qt-rakenteen Qt D-Bus Adaptor -luokkaa. Omaa luokkaa käytetään adaptor-luokkana periyttämällä se QObject-nimisestä luokasta, joka antaa sille pääsyn sen tarjoamiin toimintoihin. Tämän lisäksi periytetyn luokan määrityksen alkuun on lisättävä Q_OBJECT-makro. Omasta luokasta löytyvät metodit saadaan väylälle näkyviin määrittämällä ne public slot -tyyppisiksi. Tämä määritys tehdään luokan sisälle. Näiden metodien on seurattava Qt D-Bus Type System -järjestelmän sääntöjä. (Declaring Slots in D-Bus Adaptors 2022.)

Lähdekoodin kääntäminen suoritettavaksi ohjelmaksi vaatii QObject-luokan sisältävän lähdekooditiedoston käsittelemistä moc-työkalulla. Tämä työkalu lukee tiedoston ja etsii sieltä Q_OBJECT-makron. Tästä luodaan uusi C++-lähdekooditiedosto, joka sisältää kaiken tarvittavan sen toimintaa varten. (Using Qt D-Bus Adaptors 2022.) Qt-rakenteen versiossa numero 5 tämä voidaan tehdä automaattisesti lisäämällä CMakeLists.txt-tiedostoon rivi set(CMAKE_AUTOMOC ON). Versiosta 6.4 eteenpäin tiedostoon lisätään rivi qt_standard_project_setup(), joka tekee tämän edellisen rivin lisäksi muita tarvittavia toimintoja. (Get started with CMake 2022.)

Väylälle yhdistäminen ja omien metodien rekisteröinti tapahtuu QDBusConnection-luokasta löytyvillä metodeilla. Esimerkiksi system-väylälle yhdistetään kutsumalla QDBusConnection-luokan systemBus-metodia, joka palauttaa QDBusConnection-olion. Palvelun rekisteröinti väylälle tehdään olion registerService-metodin avulla. Tälle annetaan parametrina palvelun service name. Oma luokka ja sen metodit rekisteröidään metodilla registerObject. Tälle annetaan parametreinä polku, josta metodeja voidaan kutsua sekä luokasta tehty olio, josta metodit löytyvät. (QDBusConnection Class 2022.)

Metodien kutsumiseen löytyy useita tapoja. Jos on tarvetta kutsua vain tiettyä metodia ja sen osoite tiedetään, voidaan viesti lähettää suoraan vastaanottajalle. Lähettäjä yhdistää väylälle ja luo QDBusMessage-viestin. Viesti luodaan QDBusMessage-luokan createMethodCall-metodilla. Tälle annetaan parametreina palvelun service name, polku metodin sijaintiin ja kutsuttavan metodin nimi. (QDBusMessage Class 2022.) Luodulle viestille annetaan sen data ja se lähetetään QDBusConnection-luokan send-metodilla. Tämä metodi lähettää viestin eikä odota paluuarvoa tai vastausta. Viesti on mahdollista lähettää

myös call-metodilla, joka jää odottamaan vastausta ja palauttaa sen. (QDBus-Connection Class 2022.)

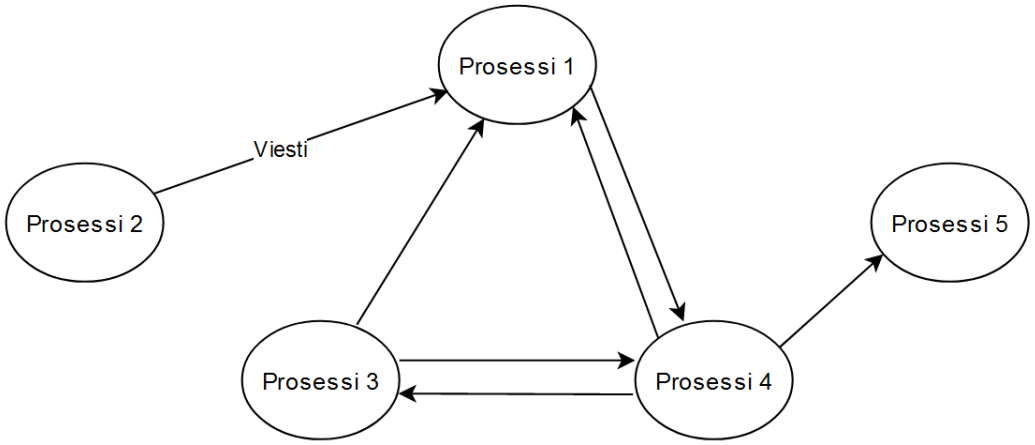
4 VISUALISOINTI

Datan visualisoinnilla tarkoitetaan datan esittämistä grafiikkaa käyttäen. Data voi olla melkein mitä vain ja esitystapoja on lukuisia. Eräitä yleisiä esittämistapoja on diagrammit ja kaaviot, histogrammit sekä puurakenteet. Esitystapa on valittava datan perusteella ja sen tulisi esittää data mahdollisimman selkeällä tavalla. Visualisointi mahdollistaa monimutkaisten tietosuhteiden esittämisen helposti ymmärrettävällä tavalla. Visualisointia voi käyttää datan mukaan moniin eri tarkoituksiin, esimerkiksi organisaation kokoonpanon esittämisestä eri trendien ja käyttäytymismallien selittämiseen. (IBM s.a.)

Tässä työssä kerättävä data on lähellä tietoverkkoa ja siinä kulkevaa liikennettä. Datan avulla on tarkoitus näyttää eri prosessien välisiä riippuvuuksia. Tämän takia työssä keskitytään ratkaisuihin, jotka soveltuvat verkostojen ja tietoverkkojen visualisointiin. Tutkittaviksi esitystavoiksi valittiin graafi- sekä trace-visualisoinnit.

4.1 Graafi

Eräs selkeä tapa etenkin verkostojen visualisointiin on graafi. Tällaisessa visualisoinnissa on verkko, joka koostuu solmuista ja niitä yhdistävistä linkeistä. (Richie & Sigman 2021.) Kuva 5 esittää, kuinka sovellettaessa tähän visualisointiin verkko olisi ohjelmistokokonaisuus, solmut eri prosesseja ja linkit esittäisivät prosessien välistä liikennettä.



Kuva 5. Esimerkki graafista

Graafeja käytetään data-analytiikassa sekä datan visualisoinnissa, kun on tarvetta visualisoida suuria verkostoja sekä verkoston sisäisiä yhteyksiä. Esimerkiksi sosiaalisen median verkostoja sekä ihmisten yhteyksiä toisiin voidaan visualisoida tällä tavoin. Graafien avulla voidaan löytää datasta keskittymiä ja korostaa eri elementtien välisiä yhteyksiä tavoin, joihin numeeriset taulukot eivät kykene. Graafit voivat esittää dataa hyvin erilaisella tavalla kuin ympyrä- ja pylväsdiagrammit. (Richie & Sigman 2021.)

Oikean tyylin määrittäminen graafille on tärkeää, koska se laajalti määrittää graafin kertoman viestin. Eräät tärkeimmät määrittelyt ovat koko, muoto, väri sekä otsikko. Solmujen ja linkkien kokoa voidaan käyttää yhteyksien painojen tai määrien esittämiseen. Muodolla voidaan erottaa erityyppinen data toisesta. Linkin muoto voi myös kertoa jotain kahden solmun suhteesta. Värejä voidaan myös käyttää erityyppisen datan erottamiseen tai jonkin numeerisen datan esittämiseen värialueena. Otsikointia voidaan käyttää moneen eri tarkoitukseen, kuten solmun nimen kertomiseen. (Richie & Sigman 2021.)

Solmujen selkeällä asettelulla on merkittävä rooli graafin ymmärrettävyydessä. Graafista voi helposti tulla vaikeasti luettava verkko satunnaisesti aseteltuja solmuja ja linkkejä. Tärkeimmät periaatteet selkeälle graafin asettelulle ovat seuraavat: jokaisen solmun tulisi olla näkyvissä, jokainen linkki tulisi olla seurattavissa lähteestä määränpäähän ja keskittymät sekä poikkeamat tulisi olla tunnistettavissa. (Dunne & Shneiderman 2009, 2.)

4.2 Graafikirjastot

Graafien laajan data-analytiikkakäytön takia valmiita graafikirjastoja löytyy useita. Näillä kirjastoilla on mahdollista luoda suuriakin graafeja suhteellisen helposti, eikä niiden käyttäjien tarvitsee ymmärtää taustalla tapahtuvasta matematiikasta tai piirtämisestä. Nämä helpottavat etenkin ohjelmointia ja piirtämistä, koska monet ominaisuudet löytyvät valmiina.

4.2.1 Python

Python-ohjelmointikieli on suosittu datan käsittelemiseen ja sille on kehitetty useita kirjastoja graafien luomiseen sekä piirtämiseen. Yksi tapa graafien luomiseen olisi käyttää NetworkX-kirjastoa graafidatan luomiseen ja Pyvis-kirjastoa graafidatan visualisointiin. Vaihtoehtoja näille kirjastoille on olemassa, mutta tämän työn kannalta nämä todettiin parhaiten soveltuviksi.

NetworkX on graafidatan käsittelemiseen kehitetty kirjasto, jonka avulla voidaan luoda graafi sille annetusta datasta. Kirjasto sisältää tietorakenteita sekä algoritmeja graafien luomiseen. (Hagberg ym. 2008, 11.) Tällä luotu graafidata on mahdollista visualisoida suoraan kirjastosta löytyvillä ominaisuuksilla, mutta sitä ei suositella. Kirjaston ylläpitäjät suosittelevat ulkoisen työkalun käyttöä tähän tarkoitukseen, koska sen rajalliset visualisointi ominaisuudet tulevat mahdollisesti poistumaan tulevissa versioissa. (Drawing s.a.) Yksi kirjasto NetworkX-graafidatan visualisointiin on Pyvis.

Pyvis on vis.js JavaScript -kirjaston pohjalta kehitetty graafidatan visualisointikirjasto (Introduction s.a.). Sille on mahdollista antaa NetworkX-kirjastolla tehty graafi ja visualisoida se selaimeen. Kirjasto mahdollistaa myös graafidatan luomisen suoraan kirjaston sisällä, jolloin ulkoista työkalua tämän tekemiseen ei tarvita. Kirjasto sisältää ison osan vis.js-kirjastosta, mutta kaikkia ominaisuuksia ei ole sisällytetty. Kirjaston avulla on mahdollista käsitellä vain staattista dataa, koska sivu jolle graafi piirretään, luodaan aina uudestaan ohjelmaa ajettaessa. Sivua ei ole mahdollista muokata sen luomisen jälkeen ainakaan Pyvis-kirjastoa käyttäen. (Tutorial s.a.)

4.2.2 JavaScript

Myös JavaScript-ohjelmointikielelle on kehitetty lukuisia graafien visualisointikirjastoja. Vahvuutena Python-ohjelmointikielen kirjastoihin löytyvät etenkin grafiikan ja esittämisen puolelta. JavaScript-kirjastojen etuna on se, että ne ovat selainpohjaisia. Selainpohjaisuus takaa niiden toimivuuden useilla eri alustoilla sekä käyttöjärjestelmillä. Tässä työssä tutustuttiin vis.js-kirjastoon, jonka päälle edellisessä luvussa esitelty Pyvis on kehitetty sekä tarkemmin Cytoscape.js-kirjastoon.

Ominaisuuksiltaan vis.js on kattava graafidatan visualisointikirjasto. Se on helppokäyttöinen ja mahdollistaa suurien dynaamisten datamäärien käsittelemisen. (Network s.a.) Aiemmin mainittu Pyvis-kirjasto tarjoaa Python-rajapinnan tämän kirjaston käyttämiseen (4.2.1). Kirjaston käyttäminen suoraan JavaScript-ohjelmointikielellä mahdollistaa kaikkien sen ominaisuuksien hyödyntämisen, kuten graafin päivittämisen sen luonnin jälkeen.

Cytoscape on suosittu ja kattava avoimen lähdekoodin graafien visualisointi sovellus, josta on saatavilla myös itsenäinen JavaScript-kirjasto nimeltä Cytoscape.js. Cytoscape on alun perin kehitetty biologian tutkimukseen, josta kehittyi yleinen alusta verkostoanalyysiin sekä visualisointiin. (What is Cytoscape? s.a.) Kirjasto on helppo ottaa käyttöön ja se on käytettävissä suoraan HTML-tiedostoissa pelkällä skriptin lisäyksellä sivulle.

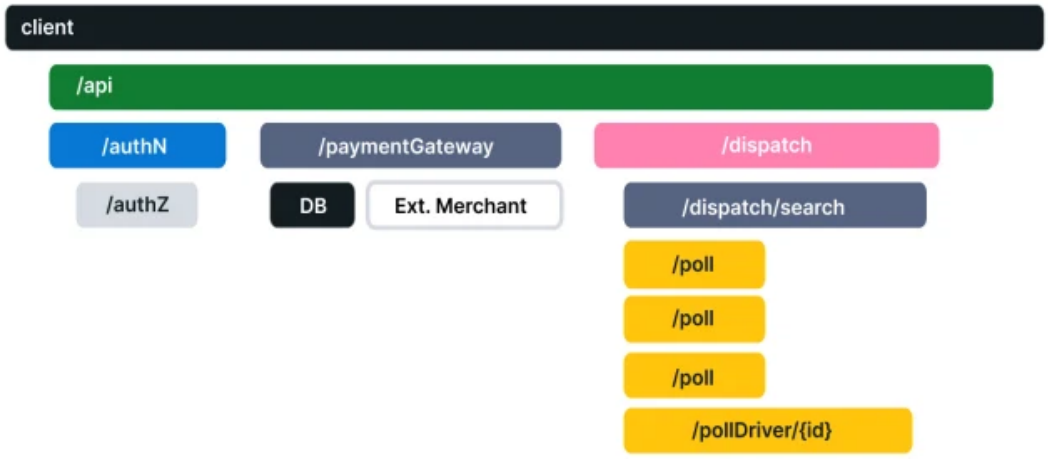
4.3 Trace

Tietoliikenteen visualistointiin sekä analysointiin käytetään usein konseptia nimeltä "trace". Näiden avulla on mahdollista esittää esimerkiksi API-kutsun reittiä sekä eri vaiheiden viemää aikaa. (Distributed Traces s.a.) Trace koostuu useista span-elementeistä. Ylimpänä on niin sanottu root span, joka esittää koko kutsun alusta loppuun. Tämän alta löytyy lisää span-elementtejä, jotka kertovat tarkempaa tietoa kutsun aikana tapahtuneista asioista.

Kutsusta saatu trace voidaan visualisoida esimerkiksi vesiputousmallina. Ylimpänä näytetään root span ja sen alla voi olla useampi lapsi span. Tämä näyt-

tää parent-child-yhteydet span-elementtien välillä. Tämän tyyppinen visualisointi tekee vianhausta helpompaa sekä tekee hajautetuista järjestelmistä helpommin ymmärrettäviä.

Kuvasta 6 näkyy trace, joka on visualisoitu vesiputousmallia käyttäen. Client on tehnyt kutsun ja sen alla näkyy mitä kaikkea kutsu kävi lävitse ennen sen valmistumista. Tästä on helposti nähtävissä, mihin kaikkialle kutsu meni ja kuinka kauan jokainen vaihe kesti.



Kuva 6. Esimerkki vesiputous mallisesta diagrammista (Distributed Traces s.a.)

Tämän tyylinen visualisointi on tullut suosituksi niin sanottujen monoliittisten sovellusten siirryttyä palvelukeskeisiin arkkitehtuureihin. Palvelukeskeisissä sovelluksissa on vaikea hahmottaa, kuinka yksittäiset transaktiot kulkevat eri kerrosten lävitse. Tämä puolestaan teki suorituskyvyn ja viiveiden aiheuttajien löytämisestä haastavaa. Trace-visualisointi teki transaktioiden seuraamisesta merkittävästi helpompaa. (Livens, J 2021.)

5 SUUNNITELMA

Ennen työkalun kehitystyön aloittamista sen tarve kartoitettiin ja tehtiin alustava suunnitelma. Kehitystyölle asetettiin aikataulu, jossa huomioitiin esiselvittelytyöt. Esiselvittelytyössä tutustuttiin erilaisiin datankeruun vaihtoehtoihin (luku 3), visualisointimenetelmiin (luku 4) ja graafikirjastoihin (luvut 4.2.1 ja 4.2.2). Esiselvittelyn perusteella valittiin käyttötarkoitukseen parhaiten soveltuvat menetelmät ja kirjastot.

Datankeruujärjestelmä koostuu kahdesta eri osioista. Ensimmäinen on tapahtumien seuranta ohjelmiston IPC-järjestelmässä ja toinen on tapahtumien keruu tiedostoon tai mahdollisesti suoraan sitä käsittelevään ohjelmaan. Datan keräämiseen tehtävää toteutusta varten tutkittiin sekä testattiin useita menetelmiä. Yksinkertaisin menetelmä olisi sellainen, jossa prosessit itse vastaavat datan keräämisestä. Tämä käytännössä tarkoittaisi sitä, että prosessit kirjoittaisivat datan tiedostoon itse. Muussa tapauksessa prosessit lähettävät datan toiselle ohjelmalle jotain toista IPC-menetelmää käyttäen. Tämän datankeruuohjelman tehtävänä on kirjoittaa vastaanotettu data yhteen tiedostoon. Vahvimmat vaihtoehdot datan lähettämistä varten olivat ohjelmistossa jo käytössä oleva IPC-järjestelmä sekä D-Bus. Puhtaasti UNIX-pohjaiset ratkaisut, kuten putki, FIFO ja stream socket -menetelmät olivat myös harkinnassa. Vaihtoehdot datan lähettämistä varten ovat rajalliset kohdejärjestelmän takia, koska valitun IPC-menetelmän ja sen käyttöä helpottavan kirjaston on oltava saatavilla siinä.

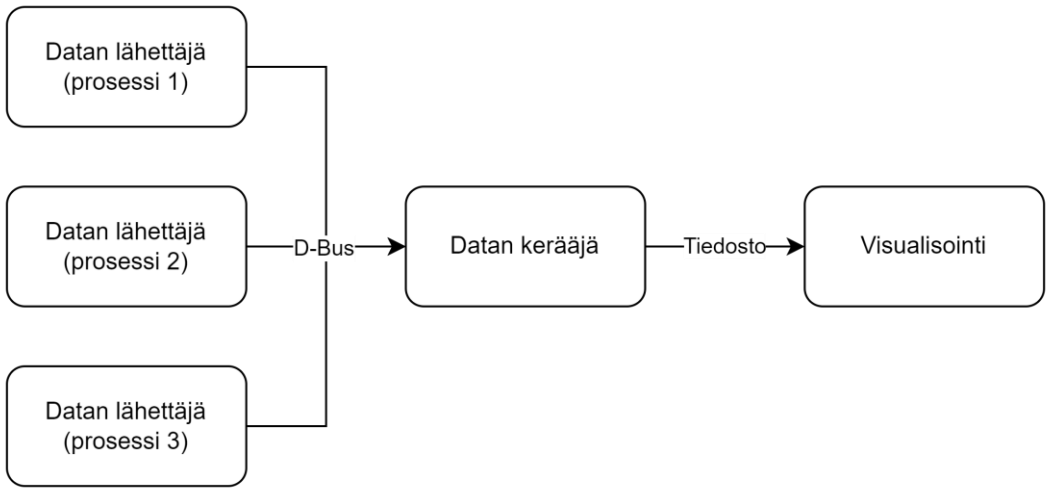
Suunnitteluvaiheessa kehitettiin muutama testiversio eri menetelmistä, jotta parhaiten soveltuva menetelmä löytyisi. Testattiin myös, onko tämän tyyppinen datankeruujärjestelmä mahdollista lisätä ohjelmistoon. Datan kerääminen aloitettiin kaikista yksinkertaisimmalla ratkaisulla, eli tiedostoon kirjoittamisella. Tässä toteutuksessa jokainen prosessi vastaa datan keräämisestä sekä tallentamisesta itse. Tällä ratkaisulla tulee vastaan selkeä ongelma race condition -tilanteen kanssa, koska prosessit eivät voi kirjoittaa yhteen tiedostoon samaan aikaan ilman synkronointijärjestelmän, kuten tiedostolukon, käyttämistä. Ratkaisulla varmistetaan myös, että datankeruujärjestelmän lisäys ohjelmistoon on mahdollinen.

Datan lähettämistä varten ohjelmiston oman IPC-järjestelmän käyttö karsittiin nopeasti pois. Vaikka järjestelmä olisi helppo ottaa käyttöön, se tulisi tuplaamaan sen kautta kulkevan IPC-liikenteen. Koska työkalun tarkoituksena on seurata tämän järjestelmän liikennettä, olisi liikenteen lisääminen tällä tavalla häiritsevää. Ongelmaksi voi myös tulla IPC-järjestelmän käyttäminen ohjelmiston käynnistyessä, jos järjestelmä ei olisi vielä käytettävissä.

D-Bus valittiin toteutuksessa käytettäväksi IPC-menetelmäksi prosessien ja datankeruuohjelman väliseen kommunikaatioon. Se valittiin, koska sitä suositaan nykyään Linux-ohjelmistokehityksessä ja kirjoittajalla oli omaa kiinnostusta tätä menetelmää kohtaan. Koska D-Bus-järjestelmän omaa kirjastoa ei suositella käytettäväksi sellaisenaan, oli valittava sopiva kirjasto kehittämistä varten. Vaihtoehdoiksi valittiin kirjastot sdbus-c++ ja Qt D-Bus, koska molempia niistä ylläpidetään, ne ovat moderneja ja D-Bus-järjestelmän ylläpitäjät suosittelevat niitä.

Molemmille vaihtoehdoille tehtiin soveltuvuusselvitys, joka sisälsi yksinkertaisen asiakas- ja palvelinohjelman tekemisen. Palvelinohjelma kuuntelee D-Bus-väylältä viestejä, jotka ovat osoitettu sille ja tulostaa ne komentoriville. Asiakasohjelma lähettää tulostettavan tekstin sisältävän viestin palvelinohjelman osoitteeseen. Tämän selvityksen perusteella molemmat kirjastot todettiin toimiviksi ja käyttökelpoisiksi. Työn käyttötarkoitukseen kirjastoista paremmaksi osoittautui sdbus-c++ sen helppokäyttöisyyden takia. Tästä huolimatta Qt D-Bus -kirjasto valittiin lopulta käytettäväksi ratkaisuksi, koska se oli helpompi ottaa kohdejärjestelmässä käyttöön. Vaikka sdbus-c++-kirjasto olisi ollut myös lisättävissä järjestelmään, sen lisääminen vain tämän ohjelman käyttöön todettiin tarpeettomaksi.

Kuva 7 esittää kokonaiskuvaa työkalun lopullisesta suunnitelmasta ja sen eri osista. Vasemmalla näkyvät eri prosessit, jotka lähettävät dataa D-Bus-järjestelmää käyttäen datan kerääjäprosessille. Näiden prosessien määrää ei ole rajattu ja niitä voi olla jopa kymmeniä. Keskellä oleva datan kerääjä tallentaa kaiken sille lähetetyn datan yhteen tiedostoon. Kerääjä on erillinen prosessi, jolla ei ole tietoa sille dataa lähettäneiden prosessien olemassaolosta. Visualisointiohjelmalle syötetään tiedosto, johon data on kerätty. Tämä ohjelma käy tiedoston rivi kerrallaan läpi ja muuntaa sen ymmärrettävämpään muotoon. Ohjelma voi myös seurata muutoksia tiedostoon ja päivittää niitä lähes reaaliajassa.



Kuva 7. Suunnitelma työkalusta

Muutamia eri visualisointimenetelmiä ja niiden soveltuvuutta tämän tyyppiseen dataan tutkittiin ja arvioitiin. Testattavaksi menetelmiksi valittiin graafi-, trace- ja tekstipohjainen visualisointi. Trace-visualisointia varten tehtiin testiohjelma soveltuvuuden testaukseen. Testien ja menetelmän tutkimisen perusteella todettiin, että trace-visualisointi ei soveltunut tämän tyyppiselle liikenteelle. Trace-visualisoinnissa seurataan viestin liikennettä eri palveluiden välillä ja mitataan niiden suoritusaikoja. Käytettävässä datassa viesti usein lähetetään yhdelle vastaanottajalle ja sitä ei välitetä eteenpäin. Suoritusajat ovat myös suuressa osassa viesteissä vaikeita määrittää, koska viestien paluuta ei aina jäädä odottamaan.

Ensisijaiseksi visualisointimenetelmäksi valittiin graafi. Graafit ovat hyvin muokattavissa tarpeen mukaan ja yksittäiset relaatiot komponenttien välillä ovat helppoja esittää. Graafin solmut määritettiin eri prosesseiksi ja niiden väliset linkit prosessien väliseksi liikenteeksi. Koska pelkkä graafi ei riitä esittämään erillisiä viestejä ja tapahtumia, suunniteltiin graafin rinnalle myös tekstipohjainen visualisointi. Tämän tarkoituksena on esittää yksittäiset viestit ja tapahtumat helpommin luettavassa muodossa.

Graafivisualisoinnille määritettiin sen tarve ja tarvittavat ominaisuudet. Työkalun ensisijainen tarkoitus on esittää ohjelmiston IPC-liikenteellä syntyvät riippuvuudet. Tätä liikennettä tulisi pystyä suodattamaan eri prosessien ja viestien perusteella. Datan reaaliaikainen esitys tulisi myös olla mahdollista, joko tiedostosta lukemalla tai suoraan kerääjäprosessilta.

6 DATAN KERÄÄMINEN LIIKENTEESTÄ

Toteutus datan keräämistä varten aloitettiin lisäämällä ohjelmiston IPC-järjestelmään toiminto datan talteen ottamista varten. Tämän toiminnon avulla jokainen järjestelmän läpi kulkeva tapahtuma muunnetaan JSON-muotoon. JSON-tietue sisältää oleelliset tiedot tapahtumasta kuten ajan, lähettäjän ja vastaanottajan. JSON-tietueen luonnin jälkeen se käsitellään ja kerätään tiedostoon.

Tapahtumien käsittely ja kerääminen aloitettiin suoralla tiedostoon tallennuksella, jossa jokaisella prosessilla on oma tiedosto. Race condition -tilanteiden välttämiseksi jokaisen tiedoston nimeen lisättiin kirjoittajan säikeen tunniste siltä varalta, että yhdellä prosessilla olisi useita säikeitä. Tämän menetelmän avulla pystyttiin toteamaan datan talteen ottamisen toimivuus, mutta lopullista ratkaisua siitä ei lähdetty kehittämään. Tiedostoja luotiin kymmeniä ja kirjalla pysyminen niistä olisi ollut vaikeaa. Tiedostojen käsittely visualisointiohjelmassa tulisi myös olemaan haastavaa, jos käsiteltäviä tiedostoja olisi useita. Tähtäimenä oli päästä yhteen tiedostoon, johon kaikki kerätty data tallennetaan. Tämän tiedoston voi sitten syöttää suoraan visualisointiohjelmaan, joka käy sen rivi kerrallaan läpi. Yksi tiedosto mahdollistaa myös raakadatan helpomman läpikäymisen.

6.1 Datankeruuohjelma

Datan keräämistä varten kehitettiin oma ohjelma, jonka tehtävänä on kuunnella sille lähetettyä dataa D-Bus-väylän yli ja tallentaa se tiedostoon. Datan lähettämiseen kerääjälle käytettiin Qt-rakenteesta löytyvää D-Bus-kirjastoa, joka otettiin käyttöön ohjelmiston IPC-järjestelmän CMakeLists.txt-tiedoston avulla. Kirjastoa hyödyntäen kehitettiin menetelmä datan lähettämiseen prosesseilta datankeruuohjelmaan. Tarkemmat selitykset toteutuksessa käytettävistä metodeista löytyy luvusta 3.5.2.

Datankeruuohjelma on toiminnoiltaan yksinkertainen ja se käyttää datan keräämiseen ohjelmaan luotua luokkaa, jonka sisällä on yksi metodi. Metodia kutsumalla voidaan kirjoittaa sille parametrina annettu teksti tiedostoon. Ohjelman luokka ja sen sisältävä metodi rekisteröidään D-Bus-järjestelmän session

bus -väylälle, jotta metodia voidaan kutsua toisista prosesseista väylää käyttäen. Väylänä käytetään session bus -väylää system bus -väylän sijasta, koska se mahdollistaa ohjelman suorittamisen ilman pääkäyttäjän oikeuksia.

Kuva 8 näyttää esimerkin Printer-luokasta ja sen print-metodista, jotka voidaan rekisteröidä D-Bus-väylälle. Jotta luokka saadaan rekisteröityä väylälle, se periytetään QObject-luokasta ja sen määrittelyn alkuun lisätään Q_OBJECT-makro. Tämän jälkeen määritetään public slots -kohdan alapuolelle metodit, jotka ovat kutsuttavissa väylällä. Tässä tapauksessa metodi nimeltä print, jota kutsumalla ohjelma tulostaa komentokehotteeseen sille parametrinä annetun tekstin.

```
class Printer : public QObject
{
    Q_OBJECT
public slots:
    void print(const QString &arg)
    {
        std::string stringToBePrinted = arg.toStdString();
        std::cout << stringToBePrinted << std::endl;
    }
};
```

Kuva 8. Esimerkki tulostusmetodista

Yhteys session bus -väylälle haetaan QDBusConnection-luokan sessionBus-metodilla ja sen palauttama yhteys tarkistetaan luokan isConnected-metodilla. Jos yhteys onnistuu, rekisteröidään sille palvelu käyttäen registerService-metodia. Tälle metodille annetaan parametrina palvelun nimi, joka voisi olla esimerkiksi org.organization.Printer. Kun palvelu on rekisteröity, voidaan siihen rekisteröidä ohjelmaan luodun luokan metodi. Tämä tehdään kutsumalla yhteyden registerObject-metodia.

Kuva 9 näyttää esimerkkiluokan Printer ja miten siitä löytyvät metodit rekisteröidään system bus -väylälle. Ensin Printer-luokasta luodaan uusi instanssi nimeltä printer. Tämän jälkeen haetaan system bus -väylä ja kutsutaan sen registerObject-metodia. Tätä edelsi yhteyden tarkistus ja palvelun rekisteröinti. Metodille registerObject annetaan ensimmäisenä parametrinä suhteellinen polku, josta metodit ovat löydettävissä. Tässä metodi asetetaan löytymään

aiemmin rekisteröidyn org.organization.Printer-palvelun polusta ”/”. Toisena parametrinä annetaan viittaus aikaisemmin luotuun Printer-instanssiin, josta metodi löytyy. Lopuksi kerrotaan, että kaikki slot-metodit Printer-luokasta on tarkoitus viedä väylälle. Tässä tapauksessa väylälle viedään metodi ”print”.

```
if (!QDBusConnection::systemBus().isConnected())
{
    std::cout << "Can't connect to D-Bus session bus\n";
    return 1;
}

if (!QDBusConnection::systemBus().registerService(
        "org.organization.Printer"))
{
    std::cout << "error registering service\n";
    return 1;
}

Printer printer;
QDBusConnection::systemBus().registerObject(
    "/", &printer, QDBusConnection::ExportAllSlots);

std::cout << "server start!\n";
```

Kuva 9. Tulostusmetodin rekisteröinti väylälle

Työhön luodussa toteutuksessa ohjelma kirjoittaa tekstin tiedostoon komentorivin sijasta. Tiedostoon kirjoittaminen vaatii tiettyjen ominaisuuksien lisäämistä ohjelmaan. Ohjelman täytyy olla ajettavissa ilman pääkäyttäjän oikeuksia, joten tiedosto on kirjoitettava sijaintiin, johon kaikilla käyttäjillä on pääsy ja tiedostolle asetetaan lukemisen sekä kirjoittamisen sallivat käyttöoikeudet. Tiedoston ja sen sisältävän kansion olemassaolo on tarkistettava ennen kirjoitusta. Kansiot sekä tiedosto luodaan niiden puuttuessa ja kirjoitettava data lisätään tiedostoon. Tiedostolle on asetettu maksimikoko, jonka ylittyessä se tyhjennetään. Tätä ennen luodaan varmuuskopio, jotta dataa ei menetetä. Tiedostolla on maksimikoko, koska ilman sitä se voisi kasvaa loputtomiin ja suuria datamääriä on vaikeampi käsitellä.

Ohjelma käännetään suoritettavaksi sovellukseksi ja se ajetaan komentoriviltä. Komentorivin kautta voidaan asettaa polku hakemistoon, jonne tapahtumat tallennetaan. Ohjelma kertoo käyttäjälle tallennettavan tiedoston polun, jonka jälkeen se jää odottamaan tulevia tapahtumia. Väylälle vietyä metodia

kutsutaan objektin kautta ja sille annetaan parametrinä tulostettava teksti. Ohjelman ajamista varten luotiin systemd-järjestelmänhallintatyökalulle service-tiedosto, joka mahdollistaa ohjelman ajamisen tietokoneen käynnistyksen yhteydessä.

6.2 Datan lähetys prosesseilta

Ohjelmiston IPC-järjestelmää käyttävät prosessit lähettävät jokaisesta IPC-tapahtumasta tiedon datankeruuohjelmalle. Tähän IPC-järjestelmään lisättiin toiminnot datan lähettämiseen. Jokainen IPC-tapahtuma on kirjattava ja siitä on kerättävä kaikki tarpeellinen tieto. Erilaisia tapahtumia on esimerkiksi viestin lähetys, viestiin vastaaminen ja viestien tilaus (subscribe) broadcast-tapahtumien kuuntelua varten. Tapahtumien kirjausta varten luotiin oma luokka, joka sisältää kaikille tapahtumille oman kirjaus metodin. Nämä metodit luovat tapahtuman tiedot sisältävän JSON-tyyppisen tietueen. Kuva 10 näyttää esimerkin JSON-datasta, jonka viestin lähetys voisi tallentaa.

```
{
  "timestamp": 1678641177,
  "sender": "process1",
  "receiver": "process2",
  "message": "message type"
}
```

Kuva 10. Esimerkki JSON-datasta

Luotu JSON-tietue lähetetään datankeruuohjelmalle D-Bus-järjestelmän session bus -väylää pitkin. Väylälle yhdistäminen ja yhteyden tarkastus tehdään samalla tavalla kuin datankeruuohjelmassa. Viestien lähettämiseen on useita tapoja ja tässä toteutuksessa käytettiin niistä yksinkertaisinta. Uusi viesti luodaan QDBusMessage-luokan createMethodCall-metodilla ja sen parametreihin annetaan vastaanottavan palvelun nimi, metodin sijainti ja kutsuttavan metodin nimi.

Kuva 11 esittää kuinka aikaisemman Printer-luokka esimerkin kanssa palvelun nimi olisi org.organization.Printer, metodin sijainti "/" ja kutsuttavan metodin nimi "print". Viestin luomisen jälkeen sille syötetään data, joka tulee menemään kutsuttavan metodin parametreihin. Printer-esimerkissä syötetyn datan

tulisi olla string-tyyppistä. Luotu viesti lähetetään D-Bus-yhteyden send-metodilla, joka on yksinkertaisin tapa lähettää viesti Qt-rakenteen D-Bus-kirjastolla. Se ottaa parametrina luodun viestin ja lähettää sen siihen asetettujen tietojen perusteella vastaanottajalle. Tämä metodi ei jää odottamaan vastausta tai tietoa lähetyksen onnistumisesta.

```
QDBusMessage busMsg = QDBusMessage::createMethodCall(
    "org.organization.Printer", "/", "", "print");

busMsg << "string to be printed";

if (!QDBusConnection::systemBus().send(busMsg))
{
    fprintf(stderr, "Queue unsuccessful\n");
    return;
}
```

Kuva 11. Datan lähetys tulostusohjelmalle

Tässä toteutuksessa send-metodi todettiin parhaaksi vaihtoehto, koska viestejä lähetetään paljon ja ne tulisi saada vastaanottajalle mahdollisimman nopeasti. Paluuarvoilla ei ole merkitystä ja vaikka tieto viestin vastaanottamisesta olisi kiinnostavaa, se ei ole oleellista ohjelman toiminnan kannalta. Dataa voidaan lähettää useilta prosesseilta nopeaan tahtiin myös saman aikaisesti ilman ongelmia. Järjestelmä jonouttaa viestit tarvittaessa ja datankeruuohjelma vastaanottaa ja kirjoittaa ne tiedostoon yksi kerrallaan.

Tiedostoon tallennettuja tapahtumia käsiteltäessä on huomioitava viestien saapumisjärjestys, koska se ei välttämättä ole niiden tapahtumajärjestys. Kahden tapahtuman tapahtuminen lähes samaan aikaan voi johtaa myöhemmän tapahtuman päätymiseen tiedostoon ennen aikaisempaa tapahtumaa. Tämä pitää ottaa huomioon dataa käsiteltäessä. Pääsääntöisesti tämä ei koidu ongelmaksi, ellei viestien saapumisjärjestystä ole tarvetta selvittää.

7 DATAN VISUALISOINTI

IPC-liikenteestä kerätty data visualisoidaan verkostograafien avulla. Kerätty data käsitellään ja siitä luodaan graafi. Visualisointiohjelman kehitys aloitettiin

Python-ohjelmointikieltä käyttämällä. Python-ohjelmointikielellä kehitettiin datan käsittelyä varten komentorivityökalu, jonka avulla raakadata voidaan kääntää helpommin luettavaksi. Esimerkiksi UNIX-aika käännetään UTC-ajaksi ja viestien numeraaliset ID-kentät niitä vastaaviksi teksteiksi. Tapahtumia voidaan myös suodattaa viestityypin, lähettäjän että vastaanottajan mukaan. Käsitelty data voidaan tulostaa joko komentoriville tai tiedostoon. Python-ratkaisun graafi luodaan myös komentorivityökalulla.

Graafin määrittäminen tehtiin NetworkX-kirjaston avulla. Kirjastolla luotavaan graafidataan määritetään kaikki graafin tiedot ja ne määritettiin seuraavanlaiseksi: solmu on yksi prosessi ja linkki esittää solmujen välistä viestiä. Solmuille määritetään myös tieto kaikista solmun naapureista. Linkkien sisälle tallennetaan jokainen sen kautta kulkenut viesti. Näistä viesteistä muodostetaan teksti, josta on luettavissa kunkin viestin lukumäärä.

Alustava visualisointi toteutettiin VisJS JavaScript -kirjaston pohjalta tehdyllä Pyvis-kirjastolla. Tähän kirjastoon voidaan suoraan syöttää NetworkX-kirjastolla luotu graafidata ja Pyvis luo siitä HTML-tiedoston, joka voidaan avata internetselaimessa. Kirjaston avulla saatiin luotua yksinkertainen visualisointi kaikista prosessien välisistä riippuvuuksista sekä visualisointi broadcast-tapahtumien kautta tulevista riippuvuuksista. Tätä dataa on mahdollista rajoittaa lähettävän sekä vastaanottavan prosessin mukaan, sekä linkissä kulkeneiden viestien mukaan. Yksittäisiä viestejä ei tällä kirjastolla saatu esille vaan pystyttiin esittämään vain mitä viestejä on kulkenut ja kuinka monta.

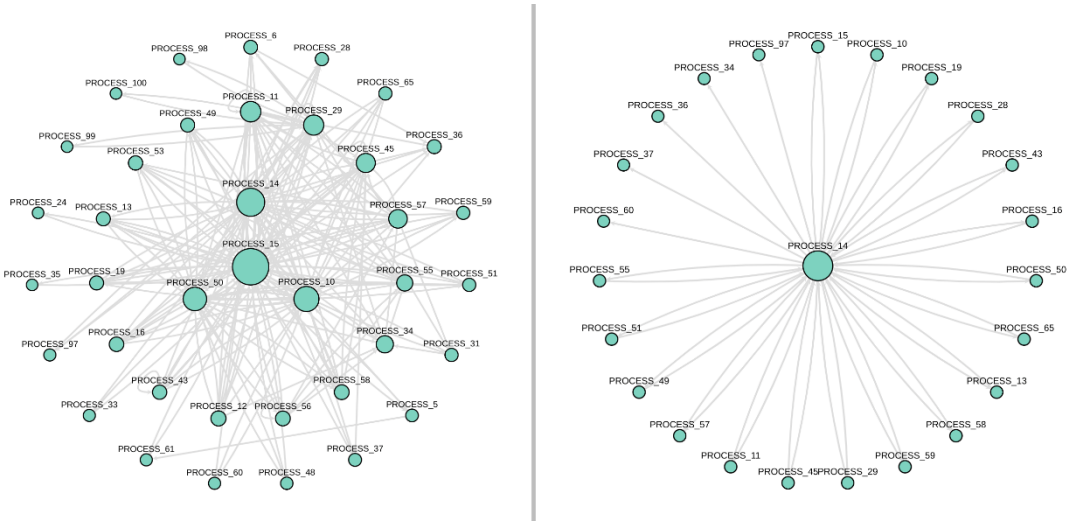
Python-kirjaston rajallisten ominaisuuksien vuoksi visualisoinnin jatkokehitys tehtiin JavaScript-ohjelmointikielellä siihen saatavilla olevien kirjastojen laajempien ominaisuuksien takia. Tämä on täysin selainpohjainen ratkaisu, joka käyttää Cytoscape-kirjastoa sekä graafin luomiseen että piirtämiseen. Lopullinen Cytoscape-kirjastolla tehtävä visualisointiohjelma pyrittiin ensiksi saamaan samalle tasolle kuin Python-ohjelmointikielellä tehty. Ohjelman toteutukseen tulisi lisätä useita parannuksia, jotka eivät olleet mahdollisia Python-toteutuksessa. Toteutus mahdollistaisi myös laajemmat jatkokehitysmahdollisuudet tulevaisuudessa kirjaston ominaisuuksien sekä muokattavuuden takia. Graafivisualisoinnin rinnalle kehitetään myös tekstipohjainen visualisointi, jonka avulla on mahdollista listata yksittäisiä viestejä ja muita tapahtumia.

Kehitys aloitettiin puhtaassa selainympäristössä, jossa otettiin käyttöön Cytoscape-kirjasto. Työkalulle lisättiin tarvittavat resurssit, joita sen käyttö edellytti. Resursseihin kuului tiedosto, johon datankeruuohjelmalla oli kerätty ohjelmistosta dataa ja ID-määritykset viestien ja prosessien nimien kääntämiseen tekstiksi. Ensimmäinen tavoite oli luoda täysi riippuvuuspuu prosesseista.

Tapahtumat sisältävä tiedosto luetaan rivi kerrallaan ja jokainen rivi käsitellään erikseen. Rivistä luetaan tapahtuman tyyppi ja se käsitellään sen edellyttämällä tavalla. Käsiteltäviä tapahtumia ovat muun muassa prosessin aloitus, viestin lähetys ja viestin tilaus. Aloitustapahtumassa luodaan uusi solmu graafiin, lähetystapahtumassa lisätään sen puuttuessa kahden solmun välille linkki ja tilaustapahtumassa lisätään solmun tietoihin tilauksen kohde. Linkin luonnin yhteydessä voidaan myös luoda solmu, jos viestin lähettäjää tai vastaanottajaa ei ole vielä lisätty. Jos linkki on jo luotu viestiä lähetettäessä, viestin tiedot lisätään linkin tietoihin. Jokaisen solmun tietoihin lisätään myös sen lähettämät ja vastaanottamat viestit. Jos viesti on lähetetty broadcast-tapahtumana, se siirtyy jatkokäsittelyyn. Jatkokäsittelyssä graafin jokaisesta solmusta tarkistetaan solmun tilauslista ja jos viesti löytyy listalta, lähetetään viesti solmulle.

Graafin solmuille kokeiltiin useita erilaisia asetteluja ja niistä valittiin kaikista toimivimmat, jotka sisällytettiin ohjelmaan. Asetteluiksi valittiin kolme yksinkertaista ja yksi fysiikkaan perustuva. Yksinkertaiset asettelut ovat ympyrä, jossa solmut muodostavat ympyrän, neliö, jossa solmut asetetaan tiiviisti suorakulmion sisälle ja samankeskinen, jossa solmut asetetaan ympyröiksi missä solmut, joilla on useita linkkejä ovat keskemmillä ympyröillä. Fysiikkaan perustuva asettelu on nimeltään force-directed, jossa solmut asettuvat annettujen parametrien mukaan fysiikkaa hyödyntäen. Ohjelmasta on mahdollista valita eri asettelujen välillä sen käyttöliittymän painikkeista.

Samankeskinen asettelu sovellettuna ohjelmistoon on nähtävissä kuvasta 12. Vasemmanpuoleinen osio esittää täyttä riippuvuuspuuta, eli kaikkien ohjelmiston prosessien riippuvuuksia toisiinsa. Oikeanpuoleinen osio esittää yhden prosessin riippuvuudet toisiin prosesseihin.



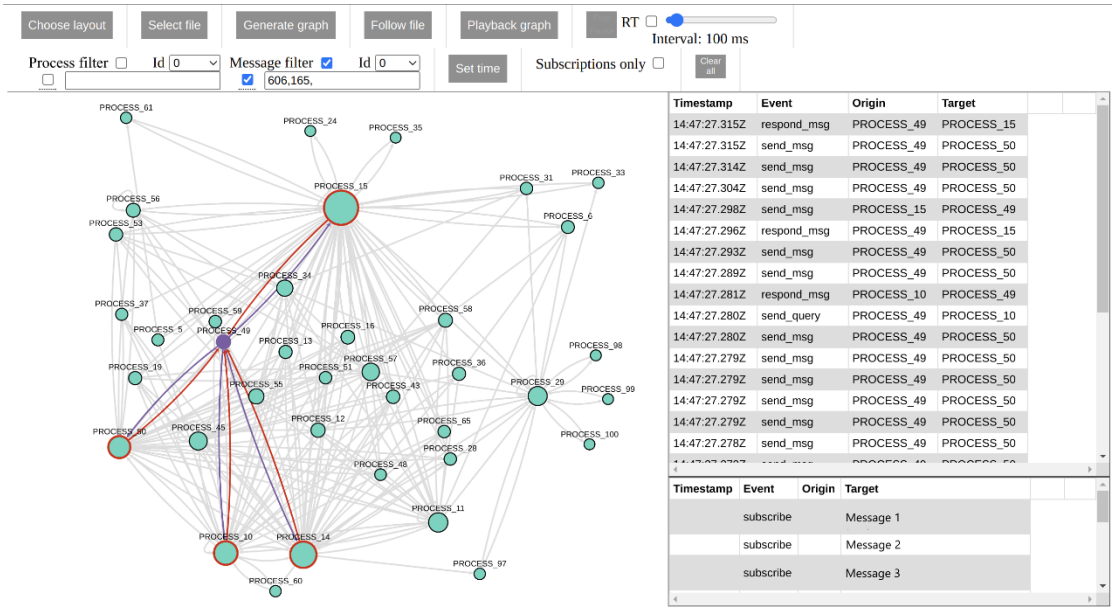
Kuva 12. Täysi riippuvuuspuu sekä yhden prosessin riippuvuudet, samankeskinen asettelu

Tekstipohjaisena visualisointina toimii graafin oikealla puolella oleva taulukko, jonka yksi rivi esittää yhtä tapahtumaa. Kun graafi luodaan, kaikki sen tapahtumat listataan taulukkoon. Rivi kirjoitetaan aina, kun uusi tapahtuma käsitellään. Graafin ja taulukon toimintoja yhdistettiin, jotta työkalusta saatiin hyödyllisempi. Kun graafista valitaan solmu tai linkki, listataan kaikki valitun kohteen tiedoista löytyvät viestit taulukkoon. Tämä toimii myös toiseen suuntaan, kun taulukosta valitaan tapahtuma, väritetään graafista tapahtumaan liittyvä kohde.

Työkalussa on viestien sekä prosessien suodatustoimintoja. Molempia voidaan suodattaa joko poistamalla valitut tai sisällyttämällä ainoastaan valitut visualisointiin. Ohjelma jättää kokonaan käsittelemättä tai käsittelee ainoastaan tapahtumat, jotka suodatetaan. Ohjelma voi myös näyttää pelkästään broadcast-tapahtumat, jolloin se suodattaa pois kaikki viestit, joiden vastaanottaja ei ole broadcast.

Kuva 13 näyttää kuvituskuvan valmiista visualisointiohjelmasta ja sen käyttöliittymästä. Ohjelman yläreunassa on työkalun ensisijainen käyttöliittymä. Solmujen asettelu valitaan Choose layout -painikkeesta. Datankeruuohjelman luoma tiedosto, josta graafi luodaan, valitaan Select file -painikkeesta. Generate graph -painike luo graafin valitusta tiedostosta. Follow file -painike aloittaa tiedoston seuraamisen sille annettuun polkuun ja päivittää sinne tulevia tapahtumia graafiin. Osioista Process filter ja Message filter voidaan asettaa suoda-

tus joko viesteille tai prosesseille. Suodattimia voidaan asettaa alasvetovalikosta valitsemalla tai syöttämällä ne suoraan tekstikenttään. Tekstikentän vasemmalta puolelta valitaan suodatustapa, joka voi olla sisällytys tai suodatus. Jos Subscriptions only -valinta asetetaan, graafi näyttää vain broadcast-tapahtumista syntyneet viestit. Jos viesti lähetettin broadcast-tapahtumana, välitetään se kaikille viestin tilaajille. Tämän valinnan avulla voidaan esittää viestitilausten kautta syntyneet riippuvuudet. Kuvasta on myös nähtävissä painike aikavälisuodatuksen asettamiseen sekä toistosäätimet tiedoston seuraus ja uusintatoisto ominaisuuksia varten.



Kuva 13. Kuvituskuva valmiista työkalusta, force-directed-asettelu

Työkalun vasen osio kattaa graafivisualisoinnin ja oikea osio tekstipohjaisen visualisoinnin. Graafista valitun solmun esitystapa on nähtävissä kuvan 13 graafiosiosta. Jos solmu valitaan, korostetaan se violetin väriseksi. Valittuun solmuun tulevat linkit värjätään punaisiksi, kun taas solmusta lähtevät linkit värjätään violeteiksi. Prosessien solmut, joiden välille on syntynyt riippuvuuksia valitun solmun kanssa, värjätään reunuksesta punaisiksi. Graafin oikealla puolella näkyvä tekstipohjainen visualisointi koostuu kahdesta taulukosta, joista ylempään listataan tapahtumat ja alempaan listataan tilaukset. Tilaukset listataan vain, jos valintana on solmu. Solmun koko määräytyy siihen liittyneiden linkkien lukumäärän mukaan. Tässä datassa solmun koko auttaa hahmottamaan prosessien riippuvuuksien lukumäärää. Suodatusta on mahdollista tehdä myös graafin kautta kaksoisnapsauttamalla solmua. Tässä tapauksessa ainoastaan napsautettu solmu ja siihen yhdistyneet solmut jäävät näkyviin.

Täysin selainpohjainen ratkaisu pystyy lukemaan sille annetun tiedoston vain kerran yhtenä kokonaisuutena. Se ei pysty lukemaan tiedostoja työkalun hakemiston ulkopuolelta pelkästään annetun polun avulla. Se ei myöskään pysty lukemaan uusia tiedostoon saapuvia tapahtumia reaaliajassa. Näiden puutteiden takia ohjelma siirrettiin ajettavaksi Electron-nimiselle alustalle. Tämä mahdollistaa ohjelman ajamisen selaimen ulkopuolella ja sallii laajemman pääsyn tietokoneen tiedostojärjestelmään.

Muutoksen jälkeen ohjelmaan lisättiin ominaisuus tapahtumien reaaliaikaiseen seurantaan. Ominaisuus lukee tiedostoon saapuneet uudet tapahtumat tietyin väliajoin ja päivittää ne graafiin sekä taulukkoon. Sen avulla voidaan seurata tiedostoa jo ennen sen luomista, millä saadaan seurattua tiedostoa sen luonnista asti. Seuranta hyödyntää tapahtumapuskuria jonne tapahtumat lisätään odottamaan käsittelemistä. Puskuri mahdollistaa sinne lisättyjen tapahtumien läpikäymisen hidastettuna tai sen voi pysäyttää kokonaan.

8 TULOKSET JA JOHTOPÄÄTÖKSET

Molemmat osat visualisointityökalua saatiin onnistuneesti valmiiksi. Datankeruuohjelma toimii hyvin, eikä sen kanssa tullut vastaan suuria ongelmia. Iso osa kehitystyön loppuvaiheesta käytettiin visualisointityökalun testaamiseen ja siitä löytyneiden virheiden korjaamiseen. Työn tekeminen edellytti monien uusien eri teknologioiden ja työkalujen opettelemista. Tämä tuki kirjoittajan omaa kehittymistä etenkin Linux- sekä web-kehityksen puolella. Lopullinen visualisointityökalu koostui datankeruuohjelmasta sekä visualisointiohjelmasta.

8.1 Visualisointityökalu

Datankeruuohjelma vastaanottaa JSON-tietueeksi muotoiltua tekstiä D-Bus-väylän yli suoritettavalla metodikutsulla. Vastaanotettu data tallennetaan keskitetysti yhteen tiedostoon. Ohjelmalle on mahdollista antaa polku, jonne tiedosto tallennetaan. Se myös rajoittaa tiedoston kokoa ja tallentaa siitä varmuuskopion tiedostokoon noustua yli asetetun rajan.

Visualisointiohjelma lukee datankeruuohjelmalla luodun tiedoston ja muodostaa siitä selaimessa esitettävän graafin cytoscape.js JavaScript -kirjaston

avulla. Ohjelma suoritetaan Electron-nimisellä alustalla käyttäjän oman selaimen sijasta. Ohjelmaan sisältyy useita ominaisuuksia datan suodattamiseen sekä esittämiseen. Datan suodattamista voidaan tehdä joko prosessien tai viestien perusteella. Suodatusta käyttämällä voidaan esittää vain määritetyt elementit graafissa tai poistaa ne siitä. Suodatusta voidaan tehdä myös aikavälin mukaan, jolloin ainoastaan asetetulla aikavälillä tapahtuneet tapahtumat näytetään. Pelkkien viestitilausten kautta kulkevat viestit voidaan esittää omana graafina työkalussa olevan ominaisuuden avulla. Graafille on määritetty neljä eri asettelua, joiden vaihtaminen onnistuu työkalun käyttöliittymästä. Asettelut ovat neliö, ympyrä, samankeskinen sekä force-directed.

Visualisointiohjelman avulla tiedostoon tulevia muutoksia voi seurata reaaliajassa tai hidastettuna. Tiedoston tapahtumat ovat myös mahdollista toistaa uusintana, joko yhtenäisellä tapahtumien välisellä viiveellä tai reaaliaikaa simuloivalla tapahtuma-ajasta lasketulla erotuksella. Erotuksella laskettu viive vaatii jatkokehitystä, koska JavaScript-ohjelmointikielen viiveominaisuus ei tue tarpeeksi pieniä viiveitä. Tähän ominaisuuteen voitaisiin tutkia jotain toista toteutustapaa tai poistaa se kokonaan.

8.2 Tutkimusongelma

Tutkimusongelma selvitettiin ohjelmistoon kehitetyllä työkalulla. Työkalu saatiin onnistuneesti valmiiksi ja se vastaa sille asetettua tarvetta. Työkalun avulla voidaan analysoida ohjelmiston IPC-liikennettä, joko aikaisemmin kerätystä datasta tai reaaliajassa ohjelmiston ajamisen yhteydessä. Datan esitys saatiin selkeäksi graafin sekä eri suodatustoimintojen avulla. Näitä käyttämällä voidaan luoda selkeitä kuvia eri prosessien välisistä riippuvuuksista, sekä havainnoida ohjelmistossa kulkevaa IPC-liikennettä. Työkalun kehitystyön yhteydessä todettiin sen hyödyllisyys myös optimoinnin kannalta. Työkalun käyttö mahdollistaa turhan ja virheellisen liikenteen löytämisen ohjelmistosta.

Työssä tutkittiin eri menetelmiä IPC-liikenteen visualisointiin ja siihen löydettiin tämän työn kannalta parhaiten soveltuva vaihtoehto. Graafi todettiin parhaaksi vaihtoehdoksi, kun on tarvetta esittää prosessien välisiä riippuvuuksia. Liikenteen sekä yksittäisten viestien esittäminen tällä tavalla oli haastavaa, mutta

niidenkin esittämisessä onnistuttiin käyttämällä animaatioita sekä tekstipohjaista visualisointia graafin rinnalla. Muita visualisointitapoja olisi myös voinut liittää työkaluun eri datan esittämiseen, mutta niitä ei työn aikataulun takia tutkittu.

IPC-liikenteen visualisoinnin hyödyistä optimoinnin kannalta saatiin osittaisia vastauksia. Vastaukset tähän tulivat kehitystyön aikana tehdyistä havainnoista, koska työkalua ei vielä työn kehityksen aikana otettu käyttöön. Näistä havainnoista oli kuitenkin mahdollista todeta, että työkalu soveltuu optimointiin. Työkalun avulla voidaan seurata eri prosessien lähettämiä sekä vastaanottamia viestejä. Tätä liikennettä analysoimalla on mahdollista selvittää eri ongelmakohtia, kuten turhia viestejä. Työkalun tarpeellisuus sekä toimivuus todennetaan, kun työkalua käytetään ensimmäistä kertaa erään ohjelmiston komponentin uudelleenkirjoitustyön kanssa. Tästä komponentista tullaan keräämään sen lähettämiä sekä vastaanottamia viestejä ja niitä analysoimalla selvitetään komponentin oleellinen liikenne. Komponentista luodaan myös kuva esittämään sen riippuvuuksia.

IPC-liikenteen datan keräämisen eri tapojen selvittäminen ja valinta kohdistui työssä tehtyyn datankeruuohjelmaan. Tätä kysymystä tutkittiin työn aikana paljon, mutta siitä huolimatta lisätutkimukselle jäi tarvetta. Itse datan poimiminen IPC-liikenteestä riippuu paljon kohdejärjestelmästä ja siellä käytettävästä IPC-menetelmästä. Tässä työssä päädyttiin keräämään ohjelmiston IPC-järjestelmästä tapahtumia JSON-tyyppiseksi dataksi muotoiltuna, joka lähetettiin toiselle prosessille käsiteltäväksi. Tämä menetelmä todettiin parhaaksi, koska se siirtää vastuun datan keräämisestä pois IPC-järjestelmästä sekä sitä käyttäviltä prosesseilta. Menetelmä mahdollisti myös datan keskittämisen yhteen sijaintiin. Eri tapoja datan keräämiseen myös tutkittiin ja datan siirtämiseen arvioitiin eri IPC-menetelmiä. Valittua D-Bus-menetelmää tutkittiin ja sen käyttöön valittiin tähän työhän parhaiten soveltuva kirjasto.

8.3 Haasteet ja jatkokehitys

Datankeruuohjelman kehityksessä tuli vastaan haasteita D-Bus-järjestelmän ymmärtämisen ja toimintaan saannin kanssa. Etenkin Qt D-Bus -kirjaston dokumentaatio oli osittain rajallista sekä vaikeata ymmärtää. Tietoa oli useilla eri

sivuilla, jotka eivät selkeästi linkittyneet toisiinsa. Dokumentaatio ei näyttänyt suoria esimerkkejä kirjaston ominaisuuksien käyttämisestä, vaan ainoat esimerkit löytyivät muutaman esimerkkiohjelman toteutuksista. Näitä esimerkkiohjelmia soveltamalla saatiin kehitettyä toimiva ohjelma soveltuvuustestaukseen. Kirjastosta löytyi useita tapoja D-Bus-väylien käyttämiseen ja näitä sovellettiin kirjaston dokumentaation luokkamäärittelyjen avulla.

Visualisointiohjelma oli paljon laajempi kokonaisuus, jonka kanssa tuli enemmän haasteita vastaan. Oikean graafikirjaston sekä käytettävien teknologioiden valinta aiheutti vaikeuksia. Useita eri kirjastoja tutkittiin ja testattiin Python- sekä JavaScript-ohjelmointikielellä. Testauksen jälkeen valittiin cytoscape.js-kirjasto lopullista toteutusta varten ja se osoittautui hyväksi valinnaksi. Ohjelman kehityksen aikana kohdattiin useita eri virheitä sen toiminnan kanssa. Esimerkiksi viestien puuttuminen oli usein vaikeata havaita suuren datamäärän seasta. Virheitä tuotti myös ohjelman toteutus. Aluksi ohjelman ominaisuudet olivat rajalliset, joten ohjelmasta tehtiin yksinkertainen. Kuitenkin ominaisuuksien määrän kasvaessa alkuperäisen toteutuksen rajoitteet tulivat esille. Tämän olisi voinut välttää suunnittelemalla ohjelma ja sen lähdekoodi paremmin alusta lähtien mahdollisten uusien toimintojen varalle. Tätä jouduttiin korjaamaan jälkeenpäin monien osien uudelleenkirjoituksella.

Kaikki suunnitellut toiminnot saatiin ainakin perustasolla sisällytettyä ohjelmaan opinnäytetyön tekemisen aikana. Vaikka mitään suunniteltua ei jäänyt työn toteutuksesta puuttumaan, voidaan jatkokehitystä tehdä useastakin kohdasta. Datankeruuohjelman luoman tiedoston kanssa tulee eteen ongelma, jos se ei sisällä tapahtumia ohjelmiston käynnistämisestä lähtien. Näiden tapahtumien puute johtaa siihen, että viestit eivät välity prosesseille oikein visualisointiohjelmassa. Tämä on ratkaistavissa käynnistämällä datankeruuohjelma tietokoneen käynnistyksen yhteydessä, joka jäi tämän työn aikana vailla lopullista ratkaisua. Tekstipohjaiseen visualisointiin käytetty taulukkoratkaisu hidastaa ohjelmaa merkittävästi, etenkin suurilla datamäärillä. Tämä olisi hyvä vaihtaa johonkin toiseen ratkaisuun, kuten valmiiseen taulukkokirjastoon. Muun tilastollisen datan kerääminen tapahtumista olisi myös hyvä jatkokehitysaihe. Tilastollisen datan esittämiseen voisi tutkia eri menetelmiä ja mahdollisuuksia.

9 POHDINTA

D-Bus-järjestelmä toimi IPC-menetelmänä hyvin ja sitä oli yksinkertaista käyttää koodissa. Selvittämättä jäi, oliko D-Bus paras vaihtoehto tämän tyyppiselle liikenteelle sekä ohjelmalle. Yksinkertaisemmat IPC-menetelmät, kuten stream socket ja FIFO olisivat mahdollisesti olleet myös riittäviä. Näiden menetelmien tarkempi tutkiminen ja testaus jäi kuitenkin tästä työstä pois tutkimus- sekä kehitystyön aikarajojen takia. D-Bus valittiin, koska se on pitkälle kehitetty järjestelmä ja suhteellisen moderni. Siinä on kuitenkin paljon ylimääräistä monimutkaisuutta, joka mahdollisesti vaikeutti sen käyttämistä tietyissä tilanteissa. Työssä kehitetty työkalu kehitettiin helpottamaan kohdeyrityksen työntekoa sekä vianhakua ja työ tehtiin sovitun aikataulun sekä rajoitusten sisällä. Tämän lisäksi kirjoittaja kehitti työkalun yksin, joka rajoitti työn laajuutta entisestään.

Datankeruuohjelman avaaminen tietokoneen käynnistyksen yhteydessä tulee vaatimaan lisäselvitystä, koska sitä ei saatu tämän työn aikarajojen sisällä täysin ratkaistua. Ohjelma olisi hyvä pitää alusta asti käynnissä, koska prosessien viestitilaukset tehdään suurelta osin ohjelmiston käynnistysvaiheessa. Ohjelman käynnistämiseen käytettiin systemd-järjestelmänhallintatyökalua, joka mahdollistaa ohjelmien avaamisen tietokoneen käynnistyessä. Ongelmia tuli systemd-työkalun vaatiman konfiguraatiotiedoston lisäyksen sekä session bus -väylään yhdistämisen kanssa. Nämä ongelmat ovat kierrettävissä käyttämällä systemd-työkalua käyttäjätilassa, joka mahdollistaa session bus -väylään yhdistämisen tai vaihtamalla session bus -väylä system bus -väyläksi. Nämä vaihtoehdot eivät kuitenkaan toimi ongelmitta kaikissa ympäristöissä, jossa ohjelmistoa ajetaan. Esimerkiksi system bus -väylän käyttäminen tarvitsee pääkäyttäjän oikeudet, jotka eivät aina ole saatavilla. Käyttäjätila ei taas ole helposti käytettävissä, jos käyttäjänä on pääkäyttäjä. Molemmat ongelmat ovat ratkaistavissa, mutta vaativat jatkokehitystä.

Visualisointityökalulle jäi useita jatkokehitysmahdollisuuksia ja parannustarpeita. Suurimmaksi tarpeeksi jäi tekstipohjaisen visualisoinnin parantaminen. Tällä hetkellä työkalu käyttää HTML-taulukkoa datan esittämiseen, joka on suurilla rivimäärillä hidas. Taulukkoa voitaisiin parantaa käyttämällä ulkoista

taulukkokirjastoa. HTML-taulukon hitaus näkyy erityisesti silloin, kun käsitellään suuria datamääriä. Hitaus vaikuttaa myös työkalun käytettävyyteen negatiivisesti. Datan suodattamiseen ja muokkaamiseen olisi voitu käyttää Cytoscape-kirjaston ominaisuuksia sen sijasta, että graafi luodaan kokonaan uusiksi muutosten jälkeen. Tämä hidastaa työkalun käyttöä ja voi aiheuttaa tietyissä tilanteissa ongelmia.

Graafivisualisoinnin rinnalle olisi voitu kehittää myös muita esitystapoja. Esimerkiksi viesti-, komponentti- sekä riippuvuusmääriä olisi voitu esittää graafin lisäksi toisellakin tavalla. Erinäköistä statistiikkaa olisi mahdollista luoda datan pohjalta, esimerkiksi millä prosesseilla on eniten riippuvuuksia, mitkä lähettävät sekä vastaanottavat eniten viestejä, millä on eniten tilauksia yms. Tämän tyyppisen datan esittämiseen voisi käyttää pylväs- ja ympyrädiagrammeja. Datan voisi myös taulukoida, jolloin dataa voi järjestää halutulla tavalla.

Työn alkuperäisenä tarkoituksena oli keskittyä juuri datankeruuratkaisuun, mutta sen nopean valmistumisen takia visualisointiohjelman kehittämiseen saatiin merkittävästi enemmän aikaa. Kehitystyö visualisointiohjelmalle aloitettiin käyttäen työkaluja, jotka eivät soveltuneet lopulliseen ratkaisuun. Tämän takia ohjelma tehtiin kokonaan uusiksi siihen paremmin soveltuvalla tekniikalla. Tekniikan vaihto vei hieman aikaa pois kehitystyöltä, mutta se osoittautui oikeaksi ratkaisuksi.

Kokonaisuudessaan työn tavoitteissa onnistuttiin hyvin eikä projektin toteutukselle tullut suuria esteitä vastaan. Datan visualisointi saatiin myös kehitettyä hyödylliselle asteelle ja alustavassa aikataulussa pysyttiin. Jatkokehitystä sekä virheiden korjausta tehtiin myös aikataulun ulkopuolella, työkalun testaamisen ja siitä saadun palautteen jälkeen.

LÄHTEET

Ballman, A. 2014. Thread Safety Analysis in C and C++. WWW-dokumentti. Päivitetty 13.10.2022. Saatavissa: https://insights.sei.cmu.edu/blog/thread-safety-analysis-in-c-and-c/ [viitattu 14.11.2022]

Cocagne, T. 2012. DBus Overview. WWW-dokumentti. Päivitetty 01.08.2012. Saatavissa: https://pythonhosted.org/txdbus/dbus_overview.html [viitattu 29.03.2023]

Davie, P & Peterson, L. 2020. Computer Networks: A Systems Approach. Amsterdam: Elsevier. E-kirja. Saatavissa: https://book.systemsapproach.org/ [viitattu 28.03.2023]

Declaring Slots in D-Bus Adaptors. 2022. The Qt Company. WWW-dokumentti. Saatavissa: https://doc.qt.io/qt-5/qdbusdeclaringslots.html [viitattu 27.2.2023]

Distributed Traces s.a. OpenTelemetry. WWW-dokumentti. Päivitetty 10.02.2023. Saatavissa: https://opentelemetry.io/docs/concepts/observability-primer/#distributed-traces [viitattu 17.02.2023]

Drawing s.a. NetworkX. WWW-dokumentti. Saatavissa: https://networkx.org/documentation/stable/reference/drawing.html [viitattu 5.2.2023]

Dunne, C. & Shneiderman, B. 2009. Improving graph drawing readability by incorporating readability metrics: A software tool for network analysts. PDF-dokumentti. Saatavissa: https://www.cs.umd.edu/sites/default/files/scholarly_papers/CodyDunneUpdate_1.pdf [viitattu 17.03.2023]

fcntl(2) — Linux manual page. 2021. The Linux man-pages project. WWW-dokumentti. Päivitetty 22.03.2021. Saatavissa: https://man7.org/linux/man-pages/man2/fcntl.2.html [viitattu 28.03.2023]

flock(2) — Linux manual page. 2021. The Linux man-pages project. WWW-dokumentti. Päivitetty 22.03.2021. Saatavissa: https://man7.org/linux/man-pages/man2/flock.2.html [viitattu 29.03.2023]

Pub/Sub Messaging. s.a. Amazon. WWW-dokumentti. Saatavissa: https://aws.amazon.com/pub-sub-messaging/ [viitattu 29.03.2023]

Get started with CMake. 2022. The Qt Company. WWW-dokumentti. Saatavissa: https://doc.qt.io/qt-5/cmake-get-started.html [viitattu 27.2.2023]

Interprocess Communications. 2021. Microsoft. WWW-dokumentti. Päivitetty 01.07.2021. Saatavissa: https://learn.microsoft.com/en-us/windows/win32/ipc/interprocess-communications [viitattu 14.11.2022]

Introduction to D-Bus s.a. KDE. WWW-dokumentti. Saatavissa: https://develop.kde.org/docs/features/d-bus/introduction_to_dbus/ [viitattu 14.11.2022]

Hagberg, A., Schult, D. & Swart, P. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. PDF-dokumentti. Saatavissa: https://conference.scipy.org/proceedings/SciPy2008/paper_2/full_text.pdf [viitattu 5.2.2023]

Introduction s.a. Pyvis. WWW-dokumentti. Saatavissa: https://pyvis.readthedocs.io/en/latest/introduction.html [viitattu 5.2.2023]

Kalin, M. s.a. A guide to inter-process communication in Linux. Opensource.com. PDF-dokumentti. Saatavissa: https://opensource.com/sites/default/files/gated-content/inter-process_communication_in_linux.pdf [viitattu 14.11.2022]

Kananen, J. 2019. Opinnäytetyön ja pro gradun pikaopas: Avain opinnäytetyön ja pro gradun kirjoittamiseen. Jyväskylä: Jyväskylän ammattikorkeakoulu.

Kerrisk, M. 2010. The Linux Programming Interface: A Linux and UNIX System Programming Handbook. San Francisco: No Starch Press.

Network s.a. vis.js. WWW-dokumentti. Saatavissa: https://visjs.github.io/vis-network/docs/network/ [viitattu 10.3.2023]

QDBusConnection Class. 2022. The Qt Company. WWW-dokumentti. Saatavissa: https://doc.qt.io/qt-5/qdbusconnection.html [viitattu 27.2.2023]

QDBusMessage Class. 2022. The Qt Company. WWW-dokumentti. Saatavissa: https://doc.qt.io/qt-5/qdbusmessage.html [viitattu 27.2.2023]

Qt D-Bus Overview s.a. The Qt Company. WWW-dokumentti. Saatavissa: https://doc.qt.io/qt-6/qtdbus-overview.html [viitattu 5.2.2023]

Qt Features, Framework Essentials, Modules, Tools & Add-Ons. The Qt Company s.a. WWW-dokumentti. Saatavissa: https://www.qt.io/product/features?hsLang=en [viitattu 5.2.2023]

Ritchie, S & Sigman, J. 2021. Network Visualization: What and How. Issues in Science and Technology Librarianship. WWW-dokumentti. Saatavissa: https://journals.library.ualberta.ca/istl/index.php/istl/article/view/2600/2615 [viitattu 17.03.2023]

sdbus-c++ s.a. Kistler Group. WWW-dokumentti. Päivitetty: 2.2.2023. Saatavissa: https://github.com/Kistler-Group/sdbus-cpp [viitattu 5.2.2023]

Siever, E., Weber, A., Figgins, S., Love, R. & Robbins, A. 2005. Linux in a Nutshell. 5. painos. Sebastopol: O'Reilly Media.

systemd System and Service Manager. 2022. Freedesktop.org. WWW-dokumentti. Päivitetty 12.01.2022. Saatavissa: https://www.freedesktop.org/wiki/Software/systemd [viitattu 5.2.2023]

The Future of Digital Experiences s.a. The Qt Company. WWW-dokumentti. Saatavissa: https://www.qt.io/group [viitattu 5.2.2023]

Tutorial s.a. Pyvis. WWW-dokumentti. Saatavissa: https://pyvis.readthedocs.io/en/latest/tutorial.html [viitattu 5.2.2023]

Using Qt D-Bus Adaptors. 2022. The Qt Company. WWW-dokumentti. Saatavissa: https://doc.qt.io/qt-5/usingadaptors.html [viitattu 27.2.2023]

Using sdbus-c++ library s.a. Kistler Group. WWW-dokumentti. Saatavissa: https://github.com/Kistler-Group/sdbus-cpp/blob/master/docs/using-sdbus-c++.md [viitattu 17.03.2023]

What is Cytoscape? s.a. Cytoscape Consortium. WWW-dokumentti. Saatavissa: https://cytoscape.org/what_is_cytoscape.html [viitattu 17.02.2023]

What is D-Bus? 2022. Freedesktop.org. WWW-dokumentti. Päivitetty 28.02.2022. Saatavissa: https://www.freedesktop.org/wiki/Software/dbus/ [viitattu 14.11.2022]

What is data visualization? s.a. IBM. WWW-dokumentti. Saatavissa: https://www.ibm.com/topics/data-visualization [viitattu 14.11.2022]

What is distributed tracing and why does it matter? 2021. Dynatrace. WWW-dokumentti. Päivitetty 27.09.2021. Saatavissa: https://www.dynatrace.com/news/blog/what-is-distributed-tracing/ [viitattu 17.03.2023]