Tampereen yliopisto

Lassi Ollinen

HAJAUTETUN TESTAUSJÄRJESTELMÄN INTEGRAATION TOTEUTTAMISVAIHTOEHDOT

Diplomityö
Tekniikan ja luonnontieteiden tiedekunta
Mikko Salmenperä
Matti Vilkko
19.4.2023

TIIVISTELMÄ

Lassi Ollinen: Hajautetun testausjärjestelmän integraation toteuttamisvaihtoehdot
Diplomityö
Tampereen yliopisto
Teknistentieteiden DI-ohjelma
Huhtikuu 2023

Diplomityössä tutkittiin, millä eri tavoilla voidaan toteuttaa hajautetun testausjärjestelmän integrointi, toteutettiin yksi vaihtoehto ja tutkittiin kuinka se vastasi työssä asetettuihin vaatimuksiin. Työn käytännönosuudessa Sandvikille® toteutettiin jatkuvan integraation palvelu Jenkinsin® ja hajautetun Creanex-simulaattoritestausympäristön® välinen integraatio. Jenkins ja Creanex-simulaattorit yhdistettiin yhdeksi jatkuvan integraation putkeksi, jossa Jenkinsissä tapahtuva yksikkötestaus yhdistettiin simulaattoreilla tapahtuvaan integraatio- ja järjestelmätestaukseen. Keskeisimmät tutkimuskysymykset olivat: millaisilla suunnittelumalleilla hajautettu järjestelmä saadaan integroitua, ja kuinka toteutettu järjestelmäintegraatio täyttää sille asetetut vaatimukset.

Työn teoriaosuudessa käsiteltiin yleisesti ohjelmistokehitystä ja testausta eri malleilla sekä millä erilaisilla järjestelmillä testausta voidaan toteuttaa. Teoriassa järjestelmäintegraation suunnittelumalleja tarkasteltiin siitä näkökulmasta, kuinka hyvin niillä pystytään toteuttamaan hajautetun järjestelmän integraatio, sekä mitä hyötyjä ja haasteita niillä on toisiinsa verrattuna. Teoriassa järjestelmäintegraatiolle välttämätöntä viestintää tarkasteltiin paitsi järjestelmäarkkitehtuurin myös yleisesti eri viestintätekniikoiden näkökulmasta. Teoriassa lisäksi käsiteltiin relaatio- ja oliotietokantarakenteita sekä mitä hyötyjä ja haasteita niillä on toisiinsa verrattuna hajautetun järjestelmän tietokantaratkaisun näkökulmasta.

Työn käytännön osuudessa toteutettiin teorian pohjalta Sandvikin hajautetun testausympäristön integraatio. Käytännön osiossa vertailtiin, kuinka pipeline-, reititin- ja palvelupohjaisen arkkitehtuurin voisi toteuttaa Sandvikin testausympäristössä. Lopulta integraatio toteutettiin työssä käsitellyllä hub-and-spoke-pohjaisella reititinarkkitehtuurilla, johon yhdistettiin REST-arkkitehtuurin mukainen ohjelmistorajapinta. Reititinarkkitehtuurin tietokantaratkaisuna käytettiin MongoDB®-oliotietokantaa.

Työn lopputuloksena saatiin teorian pohjalta tutkittua, kuinka toteuttaa hajautetun järjestelmän integraatio. Käytännön osuuden lopputuloksena saatiin toteutettua Sandvikin testausympäristön vaatimukset täyttävä hajautetun järjestelmän integraatio, joka vastasi työn teoriassa esitettyjä väittämiä. Työn aikana pystyttiin vastaamaan tutkimuskysymyksiin siitä, millä eri tavoilla voidaan toteuttaa hajautetun järjestelmän integraatio ja kuinka toteutettu järjestelmäintegraatio täyttää sille asetetut vaatimukset.

Avainsanat: Jenkins, relaatiotietokanta, SQL, oliotietokanta, NoSQL, integraatio, hajautettu järjestelmä, MongoDB, REST, Sandvik, Creanex, hub-and-spoke, reititin, testaus

ABSTRACT

Lassi Ollinen: System integration options for distributed testing systems
Master of Science Thesis
Tampere University
Master's Degree Programme in Automation Technology
April 2023

In this master thesis, it was investigated in which different ways the integration of a distributed system can be implemented, one option was implemented, and it was investigated how it met the requirements set in the work. In the practical part of the work, the integration between the continuous integration service Jenkins® and the distributed Creanex® simulator testing environment was implemented for Sandvik®. Jenkins and Creanex simulators were combined into one continuous integration pipeline where unit testing in Jenkins was combined with integration and system testing using simulators. The main research questions were: what kind of design models can be used to integrate a distributed system, and how the implemented system integration meets the requirements set for it.

In the theory part of the work software development and testing with different models were discussed in general as well as with which different systems testing can be carried out. In theory, system integration design models were examined from the point of view of how well they can implement distributed system integration as well as what benefits and challenges they have compared to each other. Communication, which is necessary for system integration, was examined in theory not only from the perspective of the system architecture but also from the point of view of different communication technologies in general. The theory also discussed relational and object database structures and what benefits and challenges they have compared to each other from the perspective of a distributed system database solution.

In the practical part of the work, the integration of Sandvik's distributed testing environment was carried out based on theory. The practical section compared how a pipeline, router and service-oriented architecture could be implemented in Sandvik's testing environment. In the end, the integration was implemented with the hub-and-spoke-based router architecture discussed in the work which was combined with a software interface according to the REST architecture. The MongoDB® object database was used as the database solution for the router architecture.

As a result of the work, it was possible to study based on theory how to implement the integration of a distributed system. The result of the practical part was the implementation of a distributed system integration that met the requirements of Sandvik's testing environment which corresponded to the claims made in the theory of the work. During the work, it was possible to answer the research questions about the different ways in which the integration of a distributed system can be implemented and how the implemented system integration meets the requirements set for it.

Keywords: Jenkins, relational database, SQL, object database, NoSQL, integration, distributed system, MongoDB, REST, Sandvik, Creanex, hub-and-spoke, router, testing

ALKUSANAT

Työskennellessäni viime kesänä Sandvikilla ilmeni tarve kehittää ratkaisu Jenkinsin ja simulaattoritestauksen järjestelmä integraatioon. Tähän ei ollut valmista ratkaisua, joten se päädyttiin toteuttamaan osana tulevaa diplomityönä. Automaation tietotekniikan pääaineopintojen ja ohjelmistotekniikan sivuaineopintojen aikana olin oppinut useita eri tapoja toteuttaa järjestelmä integraatio teoriassa. Sen takia eri vaihtoehtojen tutkiminen osana diplomityötä tuli pitkälti vaivattomasti.

Sandvikilta haluan kiittää kaikkia tämän diplomityön aikana minua auttaneita. Erityisesti haluan kiittää Sandvikilta Jani Hännistä, joka Sandvikin osalta valvoi diplomityön etenemistä. Lisäksi haluan kiittää Sandvikilta Teuvo Elorantaa, jonka kanssa käydyt säännölliset perjantaipalaverit ovat antaneet tukea käytännön ohjelmakoodin toteuttamiseen.

Yliopistolta haluan kiittää diplomityön valvojia Mikko Salmenperää ja Matti Vilkkoa. Heidän ja Jani Hännisen kanssa käydyistä kuukausittaisista palavereista on ollut paljon apua diplomityön tekstin rakennetta ja sisältöä hahmotellessa.

Kiitokset myös perheeni jäsenille ja tyttöystävälleni, jotka parhaansa mukaan auttoivat minua oikolukemaan tätä työtä. Tiedostan että tekstin korkealentoisuuden vuoksi tämän työn lukeminen ei ole varmasti ollut mukavinta illan vietettä.

Tampereella, 19 huhtikuuta 2023

Lassi Ollinen

SISÄLLYSLUETTELO

KUVALUETTELO

LYHENTEET JA MERKINNÄT

CAN	engl. Controller area network
CI	engl. Continious integration, jatkuva integraatio
CSS	engl. Cascading style sheet
ESB	engl. Enterprise service busin
HATEOAS	engl, Hypermedia as the engine of application state
HIL	engl. Hadrware-in-the-loop, komponenttipohjainen testaus
HTTP	engl. Hypertext transfer protocol, Hypertekstin siirtoprotokolla
IP	engl. Internet protocol, Internet protokolla
JSON	engl. Javascript object notation
MIL	engl. Model-in-the-loop, mallipohjainen testaus
NoSQL	engl. Not Only Structured Query Language
RDBMS	engl. Relational Database Management System
REST	engl. Representational State Transfer
SDLC	engl. Software Development Life Cycle, Ohjelmistokehityksen elinkaari
SIL	engl. Simulator-in-the-loop, simulaattoripohjainen testaus
SOA	engl. Service-Oriented-Architecture, Palvelupohjainen arkkitehtuuri
SOAP	engl. Simple Object Access Protocol
SQL	engl. Structured Query Language
SSH	engl. Secure Shell
URI	engl. Uniform resource identifier
URL	engl. Uniform recource locator
VIL	engl. Vehicle-in-the-loop, laitepohjainen testaus
VNC	engl. Virtual network computing
XML	engl. Extensible markup language

.

1. JOHDANTO

Järjestelmäintegraatiot ovat yleinen tapa kehittää järjestelmiä tehokkuuden lisäämiseksi. Kahden tai useamman järjestelmän integroiminen toisiinsa lisää tehokkuutta ja mahdollistaa järjestelmien toiminnan tehostamisen. Samalla sivutuotteena voi syntyä uusia synergiaetuja esimerkiksi tuotteita ja innovaatioita. Insinööri kohtaa usein systeemiarkkitehtuuria suunnitellessa ongelman siitä, minkälaisilla suunnittelumalleilla kaksi tai useampi järjestelmä saadaan integroitua toisiinsa. Tähän on tarjolla erilaisia systeemiarkkitehtuurisia ratkaisuja, joita avuksi käyttäen voi kehittää parhaan ratkaisun erilaisiin järjestelmäintegraatio-ongelmiin.

Kahden tai useamman olemassa olevan järjestelmän integraatiota vaikeuttaa usein molempien järjestelmien eriävät tarpeet. Järjestelmien erilaiset rajapinnat voivat myös aiheuttaa ongelmia. Lisäksi ongelman keskiössä on se, kuinka järjestelmäintegraation saa toteutettua mahdollisimman kevyesti ja helposti ylläpidettävällä tavalla, joka ei tulevaisuudessa estä järjestelmien kehitystä yhdessä eikä erikseen.

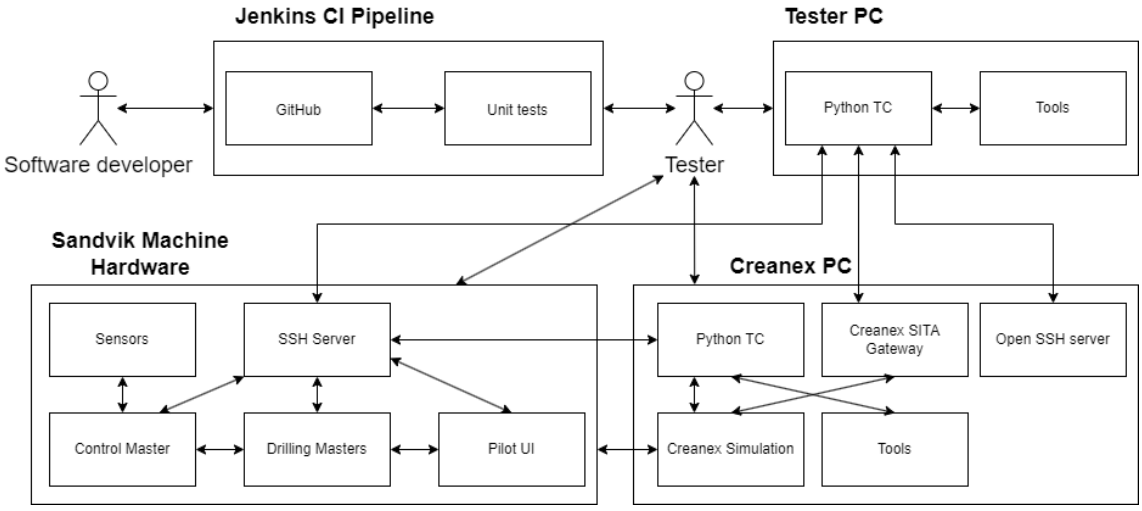
Tässä diplomityössä tutkitaan millä eri tavoilla voidaan toteuttaa hajautetun järjestelmän integraatio, ja kuinka voidaan toteuttaa Sandvikin hajautetun testausympäristön integraatio. Tämän työn taustaa käsitellään enemmän luvun ensimmäisessä alaluvussa. Toisessa alaluvussa kerrotaan työn tavoitteet. Kolmannessa alaluvussa kerrotaan työn tutkimusmetodeista ja neljännessä alaluvussa työn rakenteesta.

1.1 Työn tausta

Tutkimuksen taustana ja käytännön osiona toimii Sandvikin testiympäristö, joka koostuu useista järjestelmistä. Tässä diplomityössä toteutetaan näistä kahden järjestelmäintegraatio. Ensimmäinen integroitavista järjestelmistä on ohjelmistokehityksessä käytettävä jatkuvan integraation (CI, engl. Continuous integration) ohjelmisto Jenkins, jota käytetään eri ohjelmistokomponenttien yhdistämiseen yhdeksi käännöspaketiksi, jolle ajetaan säännöllisesti yksikkötestejä osana Jenkins-pipelinea [1]. Toinen integroitavista järjestelmistä on testiympäristössä toimiva automatisoitu integraatio- ja järjestelmätestausympäristö, joka koostuu useista eri Creanexin valmistamista simulaattoreista. Näihin simulaattoreihin on integroitu

kaivoslaitteen ajotietokoneet ”carrier master” (CM), ”drilling master” (DM), ajonäyttö eli ”pilotti” ja erilaiset anturit. Näissä testattavan kaivoslaitteen alustan ohjelmistot ovat ajossa ja niitä manipuloidaan Creanexin valmistaman simulaattorin ja sen tarjoaman ohjelmistorajapinnan kautta [2].

Integraatio- ja järjestelmätestausympäristössä python pohjaiset testiskriptit käyttävät Creanexin simulaattori PC:llä ajettavaa ohjelmistoa, joka simuloi aitoa kaivoslaitteen laitteistoa. Näin testataan kaivoslaitteiden ohjelmistoversiot ennen varsinaista laitetestausta ja hyväksyntätestausta, joko manuaalitestauksen tai automatisoitujen testien avulla. Alla olevassa kuvassa 1 esitellään Sandvikin testausympäristö nykyisellään käyttäen UML-rakennekaaviota.



Kuva 1. Sandvikin testausympäristö nykyisellään.

Edellä olevasta kuvasta voidaan huomata, että Sandvikin testiympäristö eli Creanex PC-simulaattori ja sitä ajavat testiskriptit ovat täysin erillisiä jatkuvan integraation palvelusta Jenkinsistä. Tämän diplomityön käytännön osiossa pyritään tutkimaan, millä eri tavoilla nämä kaksi erillistä järjestelmää saadaan integroitua toisiinsa.

1.2 Työn tavoitteet

Diplomityön tavoitteena on tutkia, millä eri tavoilla voidaan toteuttaa hajautetun järjestelmän integraatio siten, että lopputuloksena saadaan helposti ylläpidettävä ja laajennettava järjestelmä. Tämä toteutetaan tutkimalla, mitä systeemiarkkitehtuurisia vaihtoehtoja integraation toteuttamiseen on. Diplomityössä keskitytään etenkin siihen, miten kahden tai useamman erilaisen järjestelmän integraatio onnistuu hajautetussa järjestelmässä. Työssä tutkittavan hajautetun järjestelmän tulee myös tarjota keskitetty ratkaisu järjestelmän laajentamiseen, järjestelmän tilan tarkasteluun ja järjestelmäosien toiminnan vertailuun.

Diplomityön systeemiarkkitehtuurisen tutkimuksen pohjalta toteutetaan ohjelmistokehityksen yksikkötestauksen ja simulaattoreilla toteutettavan hajautetun integraatio- ja järjestelmätestausympäristöjen integraatio. Työn tavoitteena on tutkia erilaisia vaihtoehtoja hajautetun järjestelmän integraatioon sekä toteuttaa niistä yksi. Tavoitteena on myös tutkia sitä, vastasiko toteutettu järjestelmä sille asetettuja vaatimuksia vai olisiko parempi lopputulos ollut saavutettavissa muilla vaihtoehtoisilla ratkaisuilla.

Työn tavoitteet voidaan tiivistää seuraaviin tutkimuskysymyksiin:

- Millaisilla suunnittelumalleilla hajautettu järjestelmä saadaan integroitua?
- Kuinka toteutettu järjestelmäintegraatio täyttää sille asetetut vaatimukset?

Tutkimuskysymyksien lisäksi hajautetun järjestelmän integraatiolla pyritään saavuttamaan mahdollisimman helposti ylläpidettävä ja laajennettava järjestelmä. Integraatio mahdollistaisi yksikkötestauksen sekä integraatio- ja järjestelmätestauksen liittämisen toisiinsa osaksi yhteistä jatkuvan integraation putkea. Tämä mahdollistaisi uusia ja tehokkaampia tapoja järjestelmän käyttämiseksi.

1.3 Tutkimusmetodit

Tutkimus toteutetaan kahdessa osassa. Teoreettisessa osassa tarkastellaan erilaisia systeemiarkkitehtuurisia tapoja ratkaista kahden tai useamman järjestelmän integroiminen toisiinsa sekä mitä erilaisia etuja ja haittoja niillä on toisiinsa verrattuna. Teoriassa keskitytään erityisesti siihen, mitä erilaisia vaihtoehtoja on toteuttaa hajautetun järjestelmän integraatio ja mitä erilaisia rajoitteita hajautettu järjestelmä asettaa erilaisten vaihtoehtojen suhteen. Teoriaosuudessa erilaisia ratkaisuvaihtoehtoja tarkastellaan myös sen pohjalta, kuinka säilyttää integroitujen järjestelmien laajennettavuus niin yhdessä kuin erikseen, sekä kuinka hallita järjestelmää kokonaisuutena. Tutkimuskysymyksiin vastatessa vastaukset pohjautuvat työssä viitattavaan systeemiarkkitehtuurin tutkimuksiin.

Tutkimuksen käytännön osiossa toteutetaan Sandvikin testiympäristössä kahden testiautomaatiojärjestelmän integrointi keskenään. Käytännön osuuden systeemiarkkitehtuuria suunnitellessa haasteen aiheuttaa etenkin se, että testiympäristö on useasta erilaisesta simulaattorista koostuva hajautettu järjestelmä. Tämä aiheuttaa sen, että jokaiselle tulee asentaa Jenkins käännöspaketista riippuen eri ohjelmistot ennen integraatio- ja järjestelmätestien ajamista. Lisäksi testitulosten seuranta ja vertailu

tulisi mahdollistaa niin laitteiden välillä, kuin saman laitteen eri ohjelmistoversioiden välillä. Tällä pyritään löytämään ohjelmistovirheet mahdollisimman aikaisessa vaiheessa niiden korjaamisesta aiheutuvien kulujen minimoimiseksi.

1.4 Työn rakenne

Työn toisessa luvussa käsitellään työn käytännön osuuden kontekstia teorian kautta. Luvussa käsitellään yleisesti ohjelmistokehitystä ja testausta sen osana. Luvussa myös kerrotaan ohjelmistokehityksen elinkaaresta ja siihen liittyvistä syklisistä kehitysmalleista. Luvussa myös kerrotaan erilaisista mallipohjaisen testauksen tavoista sekä testausjärjestelmistä, jotka ovat oleellisia tämän työn kontekstissa. Lisäksi luvussa myös kerrotaan testaamisen hyödyistä ja haasteista.

Kolmannessa luvussa tarkastellaan järjestelmäintegraatiota yleisesti, mitä hyötyjä ja haasteita järjestelmäintegraatiossa on ja miksi siihen yleisesti päädytään. Lisäksi tarkastellaan järjestelmäintegraation teoriaa tieteellisiin lähteisiin pohjaten ja käsitellään erilaisia vaihtoehtoja kommunikointiin hajautetun järjestelmän kanssa. Tässä tarkastelussa keskitytään kommunikointiin verkkoarkkitehtuurin sekä systeemiarkkitehtuurin jo olemassa olevien rakennemallien avulla. Lisäksi kolmannessa luvussa käsitellään, mitä erilaisia etuja ja haittoja jokaisessa teoreettisessa mallissa on hajautettujen järjestelmien laajennettavuuden, seurannan ja hallinnan kannalta.

Neljännessä luvussa kuvataan työn käytännön osuuden Sandvikin testiympäristö toisen luvun teorian pohjalta. Luvussa kerrotaan testiympäristön kahdesta erillisestä järjestelmästä yksikkötestauksen sekä integraatio- ja järjestelmätestauksen osalta, keskittyen yksikkötestauksen osalta ohjelmistokehityksen yhteydessä käytettävään Jenkins CI -putkeen sekä sen toimintaan. Luvussa myös kerrotaan testiympäristön kehitystarpeet sekä työlle asetetut tavoitteet ja toiveet.

Viidennessä luvussa käsitellään erilaisia tapoja ratkaista käytännön integraatio-ongelma Sandvikin testiympäristössä ja mitä tavoitteita lopputuloksella on. Pohjalla toimii arkkitehtuurimallien teoria, jota käsitellään toisessa ja kolmannessa luvussa. Viidennessä luvussa keskitytään etenkin erilaisiin arkkitehtuurisiin ratkaisuihin UML-malleja avuksi käyttäen.

Kuudennessa luvussa käsitellään työn tuloksia ja sitä, saavutettiinko päätetyllä arkkitehtuurilla integraatiolle asetetut tavoitteet. Luvussa tarkastellaan lopputulosta vertailemalla uutta ratkaisua aiempaan ratkaisuun, vertailemalla toteutuksen teoriaa käytäntöön ja pohtimalla olisiko parempi lopputulos saavutettu erilaisilla ratkaisuilla. Tällä vertailulla pyritään myös tukemaan teorian ajatusta siitä millä erilaisilla ratkaisuilla

hajautetun järjestelmän integraatio voidaan toteuttaa ja vastasiko työn lopputulos sille asetettuihin tavoitteisiin ja toiveisiin. Lisäksi kuudennessa luvussa käsitellään mahdollisia jatkokehitysideoita.

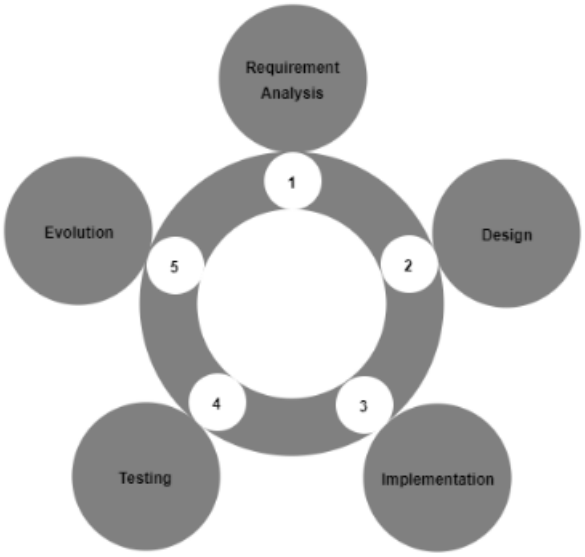
Seitsemännessä luvussa kerrotaan yhteenvetona työn lopputulos ja mitä tämän työn lopputuloksena saavutettiin. Vastattiinko työssä työn alussa määriteltyihin tutkimuskysymyksiin sekä millaisen lopputuloksen teoriaosuuden pohjalta valittu arkkitehtuurimalli mahdollisti.

2. OHJELMISTOKEHITYS JA TESTAAMINEN

Tässä luvussa käsitellään yleisesti ohjelmistokehitystä ja kautta testaamista eri mallien ja testausjärjestelmien teorian näkökulmasta. Tämä on oleellista, sillä diplomityön käytännön osuudessa integroidaan hajautettu testiympäristö. Luku koostuu neljästä alaluvusta, jossa ensimmäisessä käsitellään yleisesti ohjelmistokehityksen sykliä ja sen pohjalta kehitettyjä ohjelmistokehityksen malleja. Toisessa alaluvussa kerrotaan testauksen mallipohjaisesta suunnittelusta, testauksen vaiheista ja testiautomaation hyödyntämisestä eri vaiheissa. Kolmannessa alaluvussa käsitellään työn käytännön osion kannalta oleelliset testausjärjestelmät, ja kuinka ne hyödyntävät edellä käsiteltyä ohjelmistokehityksen ja testauksen teoriaa. Neljännessä alaluvussa kerrotaan yleisesti testaamisen hyödyistä ja haasteista.

2.1 Ohjelmistokehityksen elinkaari

Ohjelmistokehityksen elinkaari SDLC (engl. Software Development Life Cycle) tai ohjelmistokehityksen sykli pitää sisällään ohjelmiston suunnittelun, kehittämisen, testauksen ja evoluution (Kuva 2). Sitä toteuttavat tiimit seuraavat näitä vaiheita siten, että aina edellisen vaiheen lopputulos vaikuttaa seuraavaan vaiheeseen ja sen tulokseen. [17]



Kuva 2. Ohjelmistokehityksen elinkaari (Software Development Life cycle) [17]

Ohjelmistokehityksen elinkaari alkaa tarvemäärittelyllä (engl. Requirement analysis), jossa määritellään kehitettävän järjestelmän vaatimukset. Tämä toteutetaan tekemällä

saatavilla olevan tiedon pohjalta analyysiä järjestelmän vaatimuksista. Tässä vaiheessa myös määritellään kehityksen budjetti sekä tavoitteet. [17]

Tarvemäärittelyvaiheen jälkeen siirrytään suunnitteluvaiheeseen (engl. Design), jossa määritellään järjestelmän arkkitehtuuri suuressa kuvassa. Tavoitteena on määritellä millä ominaisuuksilla ja ratkaisuilla tarvemäärittelyssä määritellyt tavoitteet saadaan toteutettua. Suunnittelu voi sisältää erilaisia prosessikaavioita, käyttöliittymäsuunnittelua ja luokkakaavioita. Tässä vaiheessa myös syntyy suuret määrät erilaisia dokumentteja, joissa dokumentoidaan suunnitteluratkaisut ja niiden perusteet. [17] Tarvemäärittely on vaikein kaikista syklin vaiheista, sillä sen onnistuminen määrittelee koko projektin onnistumismahdollisuudet, laajuuden, hinnan ja käyttäjät [8].

Suunnitteluvaiheen jälkeen syklissä siirrytään toteutusvaiheeseen (engl. implementation), jossa suunniteltu ohjelmistorakenne ja niiden tavoitteet jaetaan tehtäviksi suunnittelijoiden kesken. Tämä vaihe voi kestää aina kuukausista vuosiin riippuen projektin laajuudesta. [17]

Toteutusvaiheen jälkeen siirrytään testaukseen (engl. testing), jossa toteutettu järjestelmä testataan testaustiimin toimesta. Jokainen toteutettu ohjelmistomoduuli ja yksikkö testataan ja testeistä saatu tieto kerätään. Tällä pyritään mahdollistamaan virheiden nopea korjaaminen. [17]

Kun kattava testaus on suoritettu, voidaan ohjelmisto siirtää tuotannon kautta evoluutiovaiheeseen (engl. evolution). Evoluutiovaiheessa toteutettua ohjelmistoa ylläpidetään ja jatkokehitetään käyttäjien palautteen pohjalta. Edellisen syklin lopputuloksesta saadun palautteen pohjalta edellä mainittu ohjelmistokehityksen sykli voi alkaa uudestaan tarvemäärittelystä alkaen. Tarvemäärittelyssä määritellään palautteen pohjalta uudet korjaus- ja jatkokehitysvaatimukset seuraavalle syklille. [17]

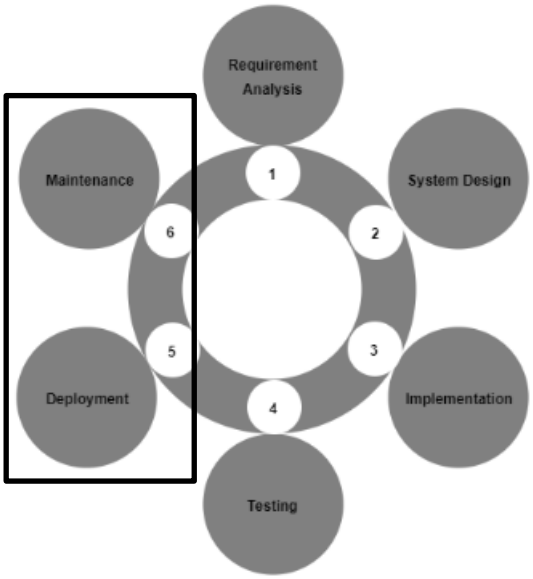
Ohjelmistokehityksen elinkaari on tärkeä ymmärtää, sillä se on eräänlainen kattotermi kaikille eri ohjelmistokehityksen ja sitä kautta ohjelmistotestaamisen muodoille. Tämän kattotermin alle kuuluvista kehitystavoista kaksi kenties tunnetuinta ovat niin kutsuttu vesiputousmalli ja ketterän kehityksen malli. Lisäksi tämän kattotermin alle kuuluu myös esimerkiksi niin kutsuttu V-malli. Näistä malleista kerrotaan enemmän seuraavissa alaluvuissa.

2.1.1 Vesiputousmalli

Vesiputousmallissa (engl. waterfall model) ohjelmistokehitys etenee muuten samoin kuin ohjelmistokehityksen syklissä, mutta evoluutiovaihe jaetaan ohjelmiston julkaisun

vaiheeseen sekä ylläpitovaiheeseen (Kuva 3). Tämä johtuu siitä, että vesiputousmalli on ensimmäinen ja vanhin syklinen ohjelmistokehityksen malli. Tämän takia se oli myös aikoinaan yleisin syklisen kehityksen malli, kunnes sai rinnalleen kilpailevia malleja. Vesiputousmallin ideana on, että suunnitteluun ei palata enää suunnitteluvaiheen loputtua. Vesiputousmallissa kehitys etenee vaiheesta seuraavaan kuukausien tai jopa vuosien aikana katsomatta taaksepäin. Tällä tavalla edetään, kunnes ohjelmistokehityksen sykli on ohitse ja ensimmäinen ohjelmistoversio valmistuu. [17]

Vesiputousmalli soveltuu tilanteisiin, joissa on saatavilla runsaasti tietoa tarvittavan suunnittelun ja dokumentaation tekemiseen. Vaihtoehtoisesti vesiputousmalliin voidaan päätyä tilanteessa, jossa kehitettävän järjestelmän täytyy onnistua ensimmäisellä yrityksellä, sillä virheisiin ei ole varaa. [17] Tällainen projekti voi olla esimerkiksi ydinvoimalahanke. Vesiputousmallin ollessa laajasti käytössä UML-mallit olivat suosittuja ja ajateltiin, että tekemällä riittävästi kaavioita kaikki mahdolliset ongelmat saataisiin kartoitettua. [18] Vesiputousmallin etuna on hyvä dokumentaatio vaatimuksista. Lisäksi vaatimukset pysyvät vakiona ja vaiheittaisen kehityksen ansiosta resursseja tarvitaan vähemmän. Tämä johtuu siitä, että projektin aikana ei tarvitse työllistää esimerkiksi kehitystiimiä ja testaustiimiä samaan aikaan. [17]



Kuva 3. Vesiputousmalli, jossa verrattuna SDLC-malliin evoluutio vaihe on jaettu julkaisun ja ylläpidon vaiheeseen. Tämä merkattu kuvassa nelikulmiolla. [17]

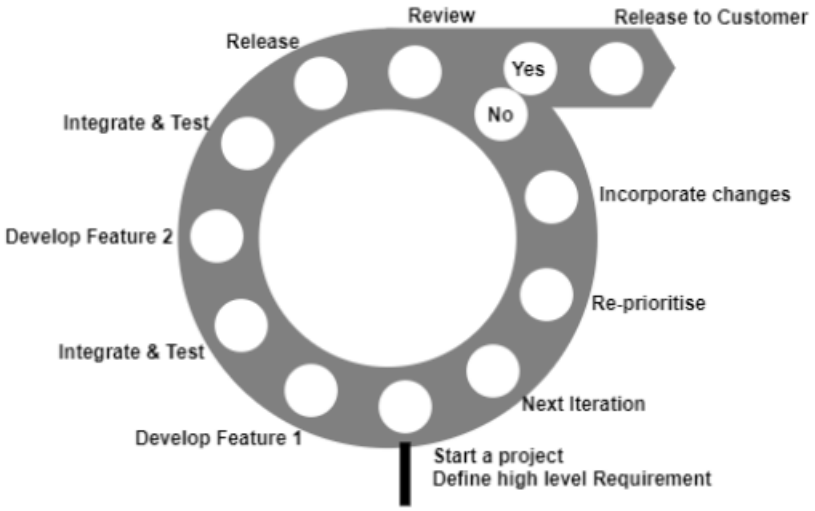
Vesiputousmallin ongelmana on kuitenkin ohjelmistoprojektin etenemisen hitaus. Yksi syklin vaihe voi kestää kuukausia tai jopa vuosia. Tämän takia muissa vaiheissa työskentelevät tiimit voivat joutua olemaan toimettomina tai työskentelemään tehottomammin. [17] Tämä voi johtaa myös siihen, että tiimit työskentelevät eri projektien parissa, jonka takia tiimit eivät kykene keskittymään yhteen projektiin saman aikaisesti [18, s.107]. Myös joustamattomuuden puute syklin keston takia voi aiheuttaa virheitä, joiden korjaaminen on paitsi kallista mutta myös aikaa vievää. [17] Usein myös mahdollinen käyttäjän palaute tulee yksinkertaisesti liian myöhään vaikuttaakseen projektin lopputulokseen [18, s.108]. Muita yleisiä ongelmia ovat integraation haasteet, edistyksen mittaamisen haasteet sekä suuri epävarmuus projektin aikana. [17]

2.1.2 Ketterän kehityksen malli

Vesiputousmallin ongelmien takia yhä useamman ohjelmiston kehityksessä ja sen testauksessa on päädytty niin kutsuttuun ketterän kehityksen malliin. Ketterän kehityksen perusperiaatteisiin sisältyy muun muassa [17]:

- Uusien toimivien ohjelmistoversioiden julkaisu kuukausien tai vuosien sijaan viikoissa.
- Asiakastyytyväisyyden ylläpito jatkuvilla ohjelmistojulkaisuilla.
- Jatkuva asiakkaiden, suunnittelijoiden, ohjelmistokehittäjien ja testaajien välinen kommunikaatio ja yhteistoiminta.
- Jatkuva mukautuminen uusiin vaatimuksiin ja tilanteisiin.
- Eri alojen asiantuntijoista koostuva tiimipohjainen kehitys.

Nämä periaatteet kuvaavat ketterän kehityksen etuja suhteessa vesiputousmalliin. Ketterässä ohjelmistokehityksessä ohjelmisto on jaettu erillisiin moduuleihin, joita pyritään kehittämään ja testaamaan osana syklistä iteraatioprosessia. Yksi iteraatiosykli kestää viikosta kuukauteen ja sisältää eri asiantuntija-alueista koostuvia tiimejä. Nämä tiimit vastaavat kukin jonkin moduulin ketterän kehityksen syklin eri osa-alueista kuvan 4 mukaisesti. [17]



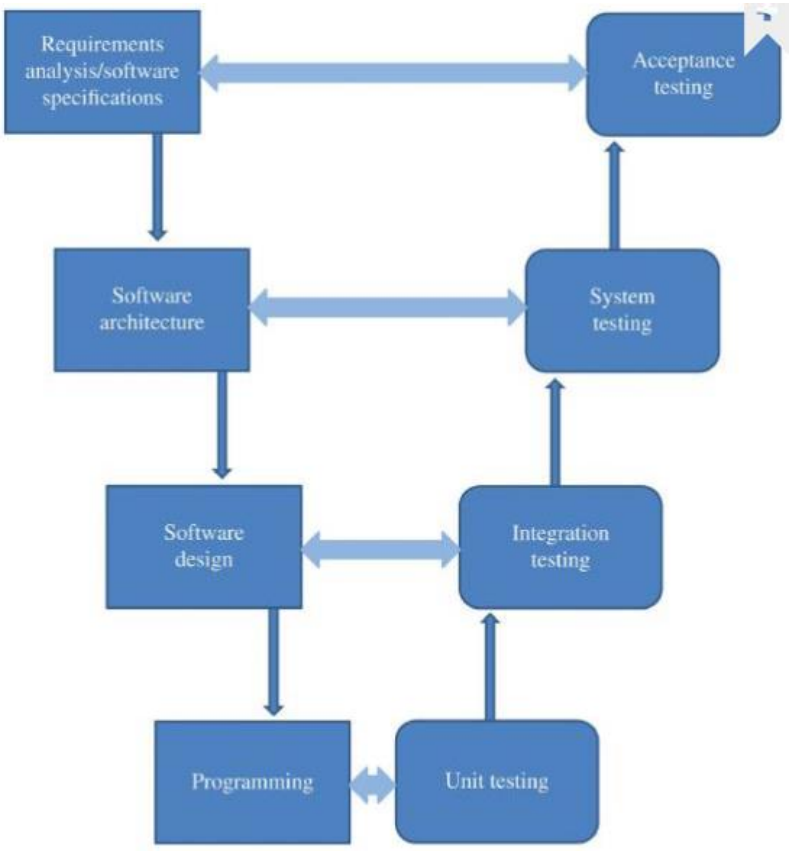
Kuva 4. Ketterän kehityksen sykli [17]

Ketterän kehityksen syklissä ei käytetä aikaa laajaan vaatimustarkasteluun tai suunnitteluun. Sen sijaan luodaan hyvin korkean tason suunnitelma, joka määrittelee juuri ja juuri projektin laajuuden. Tämän jälkeen tiimit käyvät lävitse useita iteraatiokierroksia, joissa suunnitellaan aina yksi uusi ominaisuus. Jokaisen syklin lopussa arvioidaan, onko kehitetty ominaisuus julkaisukelpoinen. Ketterässä kehityksessä tavoitteena on kehittää, testata ja julkaista yksi uusi ominaisuus iteraatiokierroksen aikana. Ketterän kehityksen etuina on muun muassa: nopeasti saavutettava toimiva versio, vähäisempi resurssien tarve, kannustus tiimityöskentelyyn, rinnakkainen kehitys, sekä vähäinen suunnittelun ja dokumentaation tarve. Ketterä kehitys pohjautuu nopeaan toimitukseen ja jatkuva integraatio auttaa ketterää kehitystä saavuttamaan tämän. [17] Lisäksi ketterä kehitys mahdollistaa testiautomaation hyödyntämisen osana ohjelmistoprojektia ja näin mahdollistaa työskentelyn entistä ketterämmin. Ketterän kehityksen osana voi myös ottaa hyötykäyttöön jatkuvan integraation putken (engl. pipeline), joka mahdollistaa jatkuvan testaamisen osana ohjelmiston kehityssykliä. [18, s. 109].

2.1.3 V-malli

Ohjelmistokehityksen ja sen testauksen voi esittää vesiputous- ja ketteränkehityksen mallin sijaan V-mallin avulla (Kuva 5). V-mallin vasemmalla puolella kulkee ohjelmistokehityksen eri vaiheet ylhäältä alaspäin aina järjestelmävaatimuksista ohjelmoinnin aloittamiseen. Vastaavasti ohjelmistotestauksen vaiheet kulkevat oikealla

puolella alhaalta ylöspäin siten, että jokaista ohjelmistokehityksen vaihetta vastaa oma ohjelmistotestauksen vaihe. V-malli ilmentää sen, kuinka ennen ohjelmointia pitää suunnitella sekä ohjelman rakenne, että mahdolliset testit. Lisäksi V-mallin mukaisessa ohjelmistokehityksessä ohjelmistotestausta tulisi toteuttaa samanaikaisesti ohjelmistokehityksen kanssa. Tällä tavalla pyritään löytämään mahdolliset ohjelmistovirheet nopeasti kehityksen edetessä [8, s. 48–49]. V-malli pohjautuu ajatukseen siitä, että jokainen vaihe tarkistetaan ja vahvistetaan ennen siirtymistä seuraavaan. [26, s. 1776]



Kuva 5. Ohjelmistotestauksen V-malli [8, s. 49].

V-mallissa edetään ensin projektin määrittelystä pienempiin osa-alueisiin ja isosta perspektiivistä aina pienimpään ohjelmakoodin palaan. Samalla suunnitellaan näille osille testejä yhdessä ja erikseen. Tämän jälkeen projektin myöhemmässä vaiheessa testit ajetaan aina pienistä kooditoteutuksista kohti yhä suurempia osa-alueita. Testit ajetaan vaihe vaiheelta yksikkö-, integraatio- ja järjestelmätesteistä aina hyväksymistestaukseen saakka, kunnes koko ohjelmisto on testattu tarpeeksi kattavasti projektin alussa määriteltyjen vaatimuksien pohjalta [8, s. 49]. V-mallista on olemassa myös useita erilaisia variaatioita kuten arvopohjainen V-malli. [26, s. 1777]

2.2 Testauksen mallit ja -vaiheet

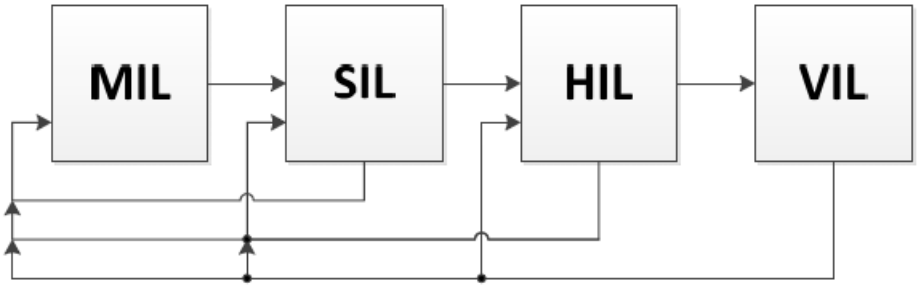
Tässä luvussa käsitellään mallipohjaista suunnittelua ja sen hyödyntämistä osana testaamista ja sen automatisointia. Luvussa myös kerrotaan lyhyesti termit eri ohjelmistokokonaisuuksien testaamiseen sekä millä eri tavoilla näitä kokonaisuuksia voidaan testata aiempien mallien pohjalta.

2.2.1 Mallipohjainen suunnittelu

Ohjelmistokehityksen elinkaarella sekä niiden vaiheilla on suuri vaikutus mallipohjaiseen suunnitteluun (engl. Model-based-design), jonka pohjalta testaaminen voidaan jakaa neljään eri vaiheeseen. Model-in-the-loop (MIL) testaus, simulator-in-the-loop (SIL) testaus, harware-in-the-loop (HIL) testaus ja vehicle-in-the-loop (VIL) testaus. Nämä muodostavat mallipohjaisessa testauksessa neljä eri vaihetta, joissa testattavaa järjestelmää testataan eri tavoilla koko ajan enemmän realistisesti. Virheen löytyessä testattava järjestelmä siirtyy vaiheissa taaksepäin. Vastaavasti testivaiheen onnistuessa ohjelmisto siirtyy testivaiheissa eteenpäin, kunnes lopulta järjestelmäversio on läpäissyt kaikkien eri testivaiheiden testit. Mallipohjainen testaus voi koostua yhdestä tai useammasta edellä mainituista vaiheista riippuen testattavan järjestelmän laajuudesta. [5]

Model-in-the-loop-testaus koostuu kokonaan simuloidusta ympäristöstä, jossa ei ole yhtään lopullisen alustan osaa, vaan ohjelmistoa testataan täysin virtuaalisessa ympäristössä esimerkiksi Simulink-mallissa. Model-in-the-loop-testauksen jälkeen ohjelmisto siirtyy simulator-in-the-loop-testaukseen, jossa manipuloitavaan simulaattoritietokoneeseen on liitetty osia lopullisen alustan laitteistosta. Simulaattoria manipuloimalla, joko automatisoidusti osana automaatiotestejä tai käsin, voidaan testata ohjelmiston toimintaa simulaattorilla. Simulator-in-the-loop testauksessa voidaan toteuttaa esimerkiksi integraatiotestausta ja järjestelmätestausta. Simulator-in-the-loop-testauksen jälkeen ohjelmiston monimutkaisia osa-alueita voidaan testata hardware-in-the-loop testauksessa erikseen niihin kuuluvissa laitteiston osissa. [5] Esimerkiksi auton turvatyynyjen ohjausjärjestelmä liitetään aitoon turvatyynyyn ja katsotaan sen toimivuus. Hardware-in-the-loop-testauksen jälkeen ohjelmisto voi vielä siirtyä vehicle-in-the-loop-testaukseen, jossa lopullista fyysistä järjestelmää manipuloidaan hallitussa testausympäristössä. [5] Testitapauksena voi toimia esimerkiksi tyhjää suoraa tietä ajava auto, jolle testi lähettää anturitietoa auton eteen liikkuvasta jalankulkijasta. Tällä testataan käynnistääkö auto hätäjarrutuksen. Nämä testauksen vaiheet ja miten

testattava malli voi siirtyä vaiheissa joko eteenpäin tai taaksepäin voidaan nähdä kuvasta 6 [5].



Kuva 6. Mallin kehityksen vaiheet [5].

Mallipohjaisen testauksen eri vaiheita voidaan hyödyntää edellä käsiteltyjen ohjelmistokehityksen mallien testausvaiheessa. Mallipohjainen testien suunnittelu ja toteutus voidaan suorittaa, joko ketterästi hyödyntäen testiautomaatiota tai manuaalisesti esimerkiksi osana regressiotestausta. Mallipohjaisen suunnittelun suurimmat hyödyt kuitenkin liittyvät ketterään kehitykseen, jossa projektin eri osa-alueita voidaan testata ketterästi mallipohjaisen suunnittelun kautta. Tämän avulla testausta voidaan suorittaa iteratiivisesti osana yhä realistisempia ympäristöjä. Tämä mahdollistaa testaamisen aloittamisen mahdollisimman aikaisessa vaiheessa osana ohjelmistokehitysprosessia.

2.2.2 Testaamisen vaiheet ja testiautomaation hyödyntäminen

Ohjelmistotestaus koostuu pääasiassa neljästä eri testausmetodista, jotka ovat yksikkö-, integraatio-, järjestelmä- ja hyväksymistestaus. Näitä testausmetodeja käytetään ohjelmistotestauksen eri vaiheissa ohjelmistovirheiden löytämiseksi. Yleissääntönä voidaan pitää: mitä aiemmassa ohjelmistotestauksen vaiheessa ohjelmistovirhe löydetään, sitä halvemmalla ja nopeammin ohjelmistovirhe saadaan korjattua.

Ohjelmistotestauksen neljästä eri metodista yksikkötestaus testaa yksittäistä koodin funktiota tai metodia eli yksikköä selvittääkseen toimiiko se oikein [19]. Usein tämä tapahtuu syöttämällä yksikön sisään haluttu arvo ja tarkastamalla, että yksiköstä ulostuleva arvo on oletetun mukainen. Yksikkötestit tekevät ja ajavat yleensä itse ohjelmakoodista vastaava ohjelmistokehittäjät, sillä yksikkötestaus toteutetaan matalalla tasolla testattavien kokonaisuuksien yksinkertaisuuden takia. Kun yksikkötestit ovat menneet lävitse, ne voidaan yhdistää omiksi ohjelmistokomponenteiksi integraatiotestausta varten. [9, s. 179] Yksikkötestaus voi olla esimerkiksi edellisessä

alaluvussa käsiteltyä MIL-testausta, jossa malliin syötetään arvoja ja katsotaan, tuleeko mallista odotettuja arvoja pihalle. Yksikkötestauksen tarkoituksena on varmistaa, että kehitettävä järjestelmä saavuttaa minimivaatimukset ennen integraatio- ja järjestelmätestausta [19].

Integraatiotestauksessa testataan usean yksikön toimintaa yhdessä osana isompaa ohjelman kokonaisuutta. Integraatiotesti voi testata jotain järjestelmän komponenttia erikseen, esimerkiksi auton moottoria. Se sisältää useita eri yksikköjä, jotka toimivat yhdessä [9, s. 182]. Integraatiotestaus voi olla esimerkiksi simulaattoritestausta (SIL), jossa jotain komponenttia testataan simulaattorilla. Integraatiotestaus voi olla myös Hardware-in-the-loop-testausta (HIL), jossa fyysistä järjestelmän komponenttia testataan erikseen muusta järjestelmästä. Vaikka yksikkötestaus ja integraatiotestaus ovat keskeisessä roolissa ohjelmiston julkaisun ja -laadun osalta, niin siitä huolimatta ne usein laiminlyödään ja tehdään tarvittaessa vain pintapuolisesti [19].

Integraatiotestauksen onnistuttua voidaan ohjelmiston eri komponentit integroida yhdeksi kokonaiseksi järjestelmäksi ja aloittaa järjestelmätestaus. Tästä esimerkkinä voi olla auton koko ohjelmiston testaus. Kun järjestelmätestaus onnistuu, lopullinen tuote siirretään asiakkaalle hyväksymistestausta varten [9, s. 184]. Sekä integraatio- että järjestelmätestauksessa on yhteistä se, että molempia toteuttavat ja suorittavat yleensä ohjelmistokehittäjien sijaan erilliset testaajat. Tämä siksi, että korkeamman tason testien luominen, ajaminen ja ylläpito on yleensä hyvin aikaa vievää ja erilliset testaajat tuovat testaamiseen uutta kokemusta ja perspektiiviä. Järjestelmätestaus voi olla esimerkiksi SIL ja VIL testausta, jossa koko järjestelmää testataan joko simulaattorilla tai aidossa ympäristössä.

Hyväksymistestauksessa lopullinen asiakas testaa järjestelmää varmistuakseen, että se vastaa asiakkaan ohjelmistotoimittajalle asettamia projektin vaatimuksia. Hyväksymistestaus on toteutettava aina asiakkaan toimesta, koska järjestelmätoimittajan intresseissä voi usein olla testata järjestelmä huolimattomasti esimerkiksi aikataulupaineiden takia. Jos hyväksymistestaus menee läpi, lopullinen ohjelmisto voidaan ottaa käyttöön. [9, s. 185]

Ohjelmistotestauksen eri vaiheissa voidaan myös käyttää useita eri tekniikoita tehostamaan ja nopeuttamaan ohjelmistotestausta. Yleisin keino tähän on testiautomaatio, joka voi olla tilanteesta riippuen joko hyvin yksinkertainen tai monimutkainen. Yksinkertaisimmillaan testiautomaatio voi olla aina ihmisen matkimisesta ja toistamisesta, automaatioskripteihin tai mallipohjaista automaatiotestausta. Mallipohjaisessa testauksessa ohjelmakoodin toiminta

yksinkertaistetaan tilakonemalliksi MIL-testauksen mukaisesti [19]. Tilakonetta käytetään ohjelmakoodin toiminnan testaamiseen tai jopa tekoälyä hyödyntävään testaukseen, jossa tekoäly sille syötetyn datan pohjalta luo uusia testejä. Testiautomaatio voi myös olla erilaista stressitestausta, jossa testataan ohjelmakoodin kestävyyttä äärirajoilla ja varmistetaan, että se toimii vaadituissa olosuhteissa. [9, s. 182]

Eri testausvaiheista yksikkötestaus kannattaa lähtökohtaisesti aina toteuttaa testiautomaatiolla, sillä yksittäisten funktioiden tai metodien testaus manuaalisesti isoissa ohjelmistokokonaisuuksissa ei ole kannattavaa eikä miellyttävää. Tämä johtuu siitä, että testattavia yksiköitä on usein liikaa manuaalisesti testattavaksi. Yksikkötestauksen automatisointi on myös yksinkertaista ja nopeaa toteuttaa, mikäli testien runko on hyvin luotu kaikkia testejä varten. [19]

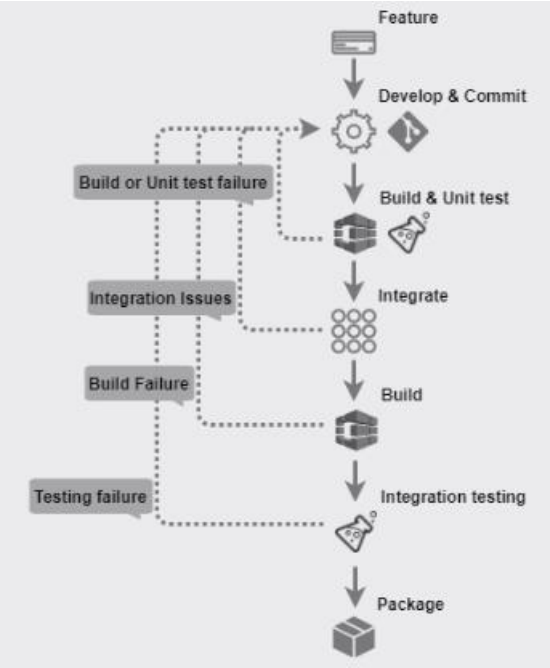
Sen sijaan yksikkötestaukseen verrattuna integraatiotestauksen ja järjestelmätestauksen automatisointi voi tuottaa enemmän vaivaa ja vaatia alusta saakka järjestelmän suunnittelua automaatiotestauksen kannalta yhteensopivaksi, kuten V-malli osoittaa. Integraatio- ja järjestelmätestauksessa voidaan usein tarvita esimerkiksi simulaattoria SIL-testausta varten, joka monimutkaistaa järjestelmän testausympäristöä huomattavasti ja tekee myös testien automatisoinnista yhä vaikeampaa. Yleisesti ottaen testaaminen tulee sitä vaikeammaksi, mitä korkeammalle tasolle ohjelmistotestauksessa edetään. Sen sijaan hyväksymistestausta ei voi lähtökohtaisesti automatisoida, sillä siinä asiakkaan tulee testata järjestelmä henkilökohtaisesti. Asiakkaalla ei myös usein ole tarvittavia järjestelmiä automaatiotestausta varten. Toki teoriassa hyväksymistestausta voisi osittain toteuttaa simulaattorilla, mutta sellaista asiakkaalla harvoin on valmiina. Hyväksymistestaus pitäisi myös lähtökohtaisesti toteuttaa aidossa käyttöympäristössä, mikä ei käytännössä ole simulaattorilla mahdollista.

2.3 Testausjärjestelmien teoria

Tässä luvussa käsitellään testausjärjestelmien teoriaa ja miten niitä yleensä käytetään ohjelmistokehityksessä. Työn ensimmäisessä alaluvussa käsitellään jatkuvaa integraatiota osana ohjelmistokehityksen elämänkaarta ja tähän liittyvää jatkuvan integraation palvelusta Jenkinsiä. Toisessa alaluvussa käsitellään esimerkkinä SIL-testauksesta Creanex-simulaattoria, jolla SIL-testaus voidaan toteuttaa. Näitä testausjärjestelmiä kuvaillaan tarkemmin, koska Jenkins ja Creanex-simulaattorit ovat Sandvikilla käytössä. Näin ollen niiden ymmärtäminen on oleellista käytännön osiossa. Lisäksi jatkuvan integraation ymmärtäminen on tärkeää, sillä tämän työn käytännön arkkitehtuuriratkaisuna toteutetaan jatkuvan integraation ratkaisu Jenkinsin ja Creanex-simulaattoreiden välille.

2.3.1 Jatkuva integraatio ja Jenkins

Jatkuva integraatio (engl. Continious Integration, CI) on ohjelmistokehityksen tapa, jossa kehittäjät jatkuvasti integroivat heidän työnsä osaksi yhteistä työaluetta ja luovat käännöspaketin (engl. build) (Kuva 7). Käännöspaketti voi epäonnistua johtuen esimerkiksi huonosta koodista tai inhimillisestä virheestä käännöspaketin aikana. Virheen sattuessa kehittäjät korjaavat vian ja tekevät uuden käännöspaketin. [17] Jenkins on yksi jatkuvan integraation työkalu, jota käytetään käännöspakettien säännölliseen luomiseen ja testaamiseen osana Jenkins pipelinea [1]. Jenkinsin lisäksi jatkuvan integraation työkaluja ovat esimerkiksi Build Forge, Bamboo ja TeamCity. [17]



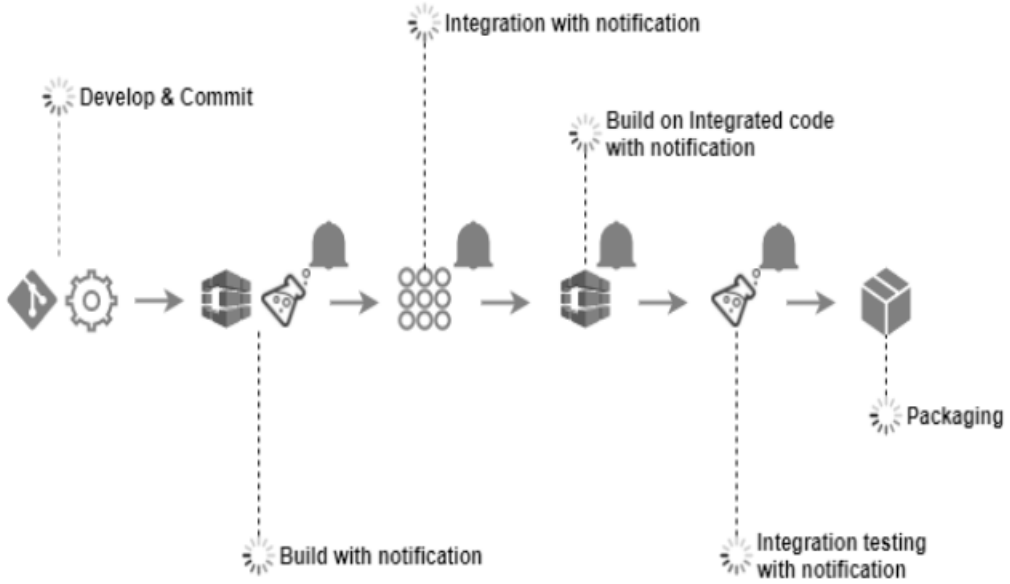
Kuva 7. Jatkuvan integraation prosessi [17]

Jenkins on käytännössä palvelin, joka mahdollistaa säännöllisen ohjelmistotestauksen eli jatkuvan testauksen lisäämisen osaksi ohjelmistokehityksen syklejä [16, s. 1]. Jenkins tukee yli 400 liitännäistä (engl. plugin), jotka mahdollistavat eri kommunikaatiomahdollisuuksia Jenkinsin ja yrityskohtaisten palveluiden välillä. Tämä mahdollistaa pohjan jatkuvalle integraatiolle. Jatkuva integraatio mahdollistaa ohjelmistovirheiden löytymisen ohjelmistokehityksen aikana eikä vasta sen lopussa. Tämä nopeuttaa virheiden korjaamista ja vähentää kustannuksia. Virheiden myöhäisempi löytyminen voisi eri arvioiden mukaan nostaa virheen korjaamisen kustannuksia jopa 100–1000 kertaiseksi. [16, s. 1–2]

Jenkinsin vahvuuksia ovat muun muassa sen toimintavarmuus, avoin lähdekoodi, yksinkertainen käyttöliittymä, selkeä isännän ja orjan topologia ja liitännäisten

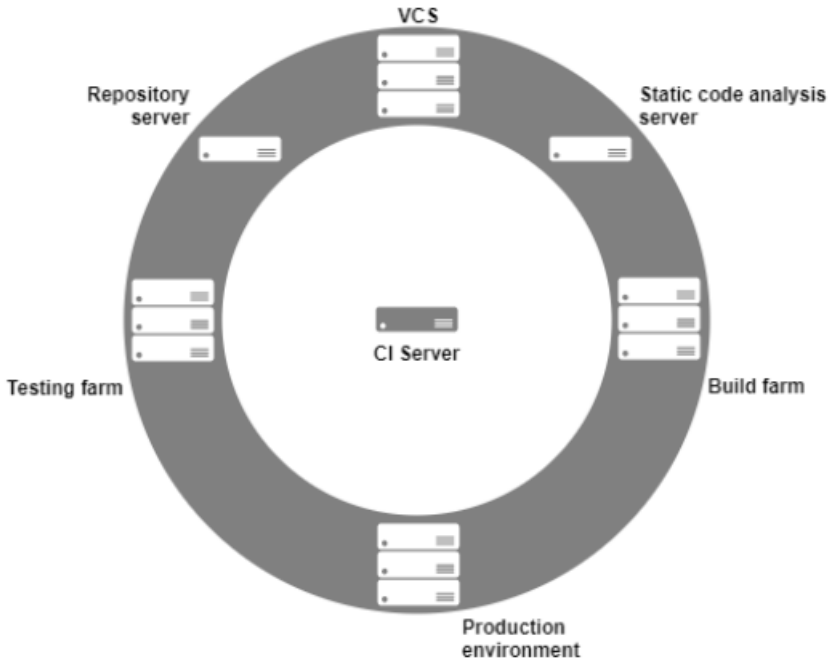
mahdollistama laaja tuki. Jenkinsin laaja yhteisö myös mahdollistaa laajan tuen saatavuuden ongelmatilanteissa, mikä helpottaa Jenkinsin laajentamista ja vikojen korjaamista. [16, s. 3]

Aiemmassa luvussa 2.1.2 käsitelty ketterä kehitys hyötyy suuresti Jenkinsin kaltaisien järjestelmien jatkuvasta integraatiosta. Ketterä kehitys pohjautuu pääasiassa nopeaan toimitukseen, joka hyötyy rinnalla ajettavasta jatkuvasta integraatiosta. Jatkuvaan integraatioon usein liitetään paitsi uuden käännöspaketin tekeminen, mutta myös sen yksikkö-, integraatio- ja järjestelmätestaaminen. Jatkuvan integraation edetessä on myös tärkeää kommunikoida tehokkaasti ohjelmistokehittäjille mahdollisista jatkuvan integraation vaiheista ja virheistä (Kuva 8). Ketterä kehitys on suurilta osin mahdollinen Jenkinsin kaltaisien ketterän kehityksen palveluiden ansiosta. [17]



Kuva 8. Jatkuvan integraation prosessi ja mahdolliset ilmoitukset [17]

Tiivistettynä jatkuvan integraation järjestelmät kuten Jenkins ovat orkestroijia, jotka kommunikoivat esimerkiksi version hallinnan, käännöspaketin hallinta työkalujen, tietokantojen, testiautomaation ja erilaisien analyysityökalujen välillä (Kuva 9). Näitä palveluita käyttäen jatkuvan integraation palvelut luovat erilaisia prosessiketjuja eli putkia (engl. pipeline). Putkissa eri prosesseja sidotaan yhteen muodostaen kehitykselle hyödyllisiä arvoketjuja esimerkiksi ohjelmistokehityksen ja testaamisen välillä. [17]



Kuva 9. Keskitetty CI-palvelin [17]

Nämä putket voivat käyttää hyödyksi mallipohjaisessa suunnittelussa mainittuja model-in-the-loop-, simulator-in-the-loop- ja hardware-in-the-loop-testausta. Vastaavasti putket voivat edetä yksikkötestauksesta integraatiotestaukseen ja integraatiotestauksesta järjestelmätestaukseen, jossa eri vaiheet testataan eri malleilla riippuen testattavan osion laajuudesta. Jatkuva integraatio mahdollistaa eri testaustapojen ja näiden testiautomaation integroimisen osaksi yhtä yhteistä arvoprosessia.

2.3.2 Creanex-simulaattori

Sandvikilla käytössä olevista Creanex-simulaattoreista puhuttaessa tarkoitetaan kuvassa 10 olevaa Creanex PC:tä ja siihen liitettyjä CAN (Controller area network) -pohjaisia kaivoslaitteen käsinoja- ja muita paneeleja. [2]. Creanex-simulaattori pitää sisällään simulaattoriohjelmiston. Simulaattoriohjelmisto tarjoaa kaivoslaitteen ohjausjärjestelmälle vasteita, kuten virtualiset venttiilit ja anturit ja toisaalta mahdollistaa myös käsin kontrolloidut syötteet testattavaan järjestelmään, kuten ohjainsauvat ja käsin asetetut anturien arvot. Näiden avulla voidaan testata testattavaa järjestelmää. Creanexin simulaattorissa yhdistyy luvussa 2.2.1 mainittu SIL- ja HIL-testaus. Simulaattoria ohjataan paitsi pilotin kosketusnäytöstä, myös joko virtuaalisten kaivoslaitteen ohjaimien tai fyysisten ohjaimien avulla riippuen tilanteesta. Esimerkiksi manuaalisessa testauksessa yleensä käytetään fyysisiä ohjaimia, sen sijaan

testiautomaatio käyttää ohjaamisessa virtuaalisia ohjaimia, kuten virtuaalisia sauvaohjaimia ja painonappeja.

Kuva 10. Creanex PC-simulaattoriympäristö.

Simuloinnin etenemistä ja tuloksia voi seurata ajonäytöstä niin kuin aidolla kaivoslaitteella, mutta myös laitteen tilakoneesta ja simulaattorin virtuaaliventtiileistä. Lisäksi laitteen tilaa voi myös seurata Creanex-simulaattorin graafisesta 3D-simulaatiosta, joka näyttää laitteen graafisen tilanteen simulaatioympäristössä. Manuaalisessa testauksessa testauksen etenemistä seurataan yleensä graafisesta käyttöliittymästä ja pilotista. Testiautomaatio seuraa testattavan koneenohjausjärjestelmän toimintaa sen simulaattoriin syöttämistä arvoista ja CAN-rajapinnan liikkuvista viesteistä.

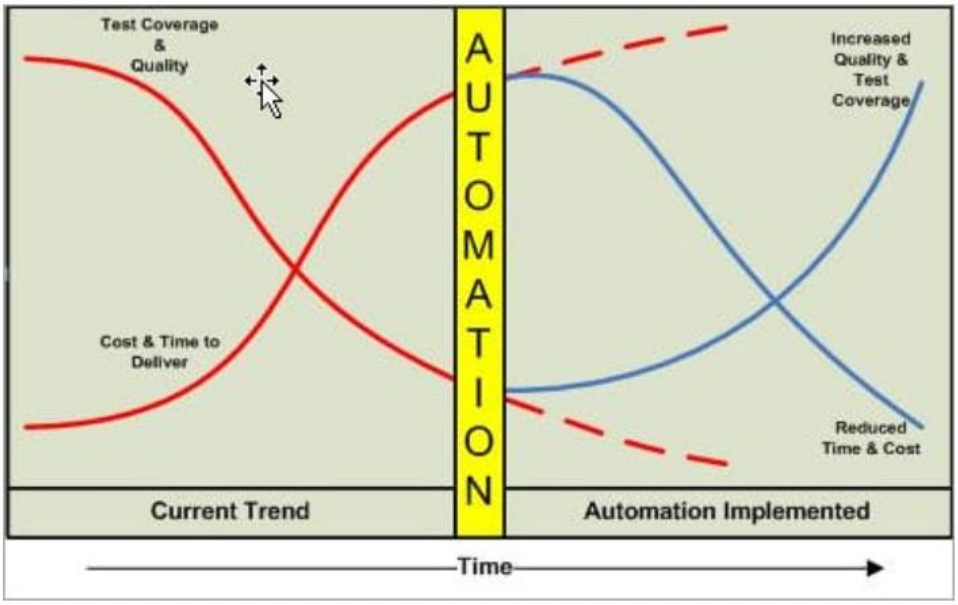
Creanexin valmistamat simulaattorit ovat yksi esimerkki siitä, kuinka simulator-in-the-loop-testaus voidaan toteuttaa käytännössä. Simulaattoriin myös yhdistetään luvun 2.2.1 mukaisesti mahdollisimman paljon oikeaa laitteistoa. Näin pyritään siihen, että simulointiympäristö olisi mahdollisimman aito, jotta testausta voitaisiin suorittaa HIL-testauksena. Simulaattorit mahdollistavat myös luvun 2.1.2 mukaisen ketterän

kehityksen ja jatkuvan integraation, jossa jokainen ohjelmistoversio pyritään testaamaan joko manuaalisesti tai testiautomaation avulla.

2.4 Ohjelmistotestaamisen hyödyt ja haasteet

Ohjelmistotestaus kannattaa lähtökohtaisesti aina tehdä, oli projekti sitten pieni tai iso. Mitä isompi ohjelmistoprojekti, sitä välttämättömämpää ohjelmistotestaus on, jotta projektin vaatimukset saadaan suoritettua aikataulussa. Tämä johtuu siitä, että mitä korkeammassa testaamisvaiheessa ohjelmistovirhe löytyy, sitä suurempi määrä mahdollisia ohjelmistovirheen aiheuttajia koodissa on. Näin tapahtuessa suuria osia koodista voidaan joutua käymään lävitse ja muuttamaan. Tämä lisää ohjelmistokehityksen vaatimaa aikaa ja sitä kautta kustannuksia. [9, s. 179] Tämän takia testauksen tulee olla jokaisella ohjelmoinnin tasolla mahdollisimman kattavaa ja tehokasta, jotta ohjelmistovirheet löydettäisiin mahdollisimman aikaisessa vaiheessa. Myös testien toteuttaminen on sitä helpompaa, mitä matalammalla tasolla testejä toteutetaan, sillä pienien testien toteuttaminen on huomattavasti nopeampaa ja helpompaa kuin suurien testien [9, s. 180].

Automaatiotestaus nopeuttaa huomattavasti testien ajamista. Tästä huolimatta automaatiotestausta ei ole aina kannattavaa toteuttaa sillä toteutukseen, ja testien ylläpitoon kuluu merkittävästi aikaa. Tämä johtuu siitä, että järjestelmän ominaisuudet ja vaatimukset voivat muuttua useaan kertaan ohjelmistoprojektin aikana. Tämä voi johtaa pahimmillaan siihen, että automaatiotestejä ei pidetä ajan tasalla ohjelmakoodin muuttuessa, jolloin lopulta automaatiotestauksen tulokset eivät heijasta ohjelmakoodin todellista tasoa. Tämän takia on kannattavaa miettiä paitsi automaatiotestauksen laajuutta, myös tarpeellisuutta. Kannattaa myös arvioida automaatiotauksen toteutukseen ja ylläpitoon vaadittavaa aikaa suhteessa saavutettavaan hyötyyn. Parhaimmillaan testikattavuuden noustessa manuaalitestauksen määrää voidaan vähentää, jolloin testauksen kustannukset ja testaukseen käytettävä aika vähenevät ajan funktiona. Testiautomaatio myös parantaa laatua, koska testien määrän kasvaessa kaikkea ei voi testata manuaalisesti (Kuva 11).



Kuva 11. Testiautomaation hinnan ja kattavuuden kehitys [25]

Ongelmana voi myös olla ongelmat testaussuunnitelman kattavuuden suhteen. Vaikka testejä olisi satoja kappaleita, se ei siltikään tarkoita, että ohjelmisto on testattu kattavasti. Tämä voi johtua siitä, että osa testeistä voi testata samaa asiaa, osa testeistä voi olla vanhentuneita ja jotkin uudet osa-alueet ohjelmistosta voivat olla täysin ilman testejä. On siis tärkeää huolehtia siitä, että testit kattavat tasaisesti koko ohjelmakoodin.

Yksi testauksen ongelma on myös rajapintaongelmat eri testausvaiheiden välillä, etenkin automaatiotestauksessa. Esimerkiksi mistä integraatiotestausjärjestelmä tietää, että tarpeeksi moni yksikkötesti on mennyt läpi, jotta integraatiotestaus voidaan aloittaa. Tätä varten on erilaisia ratkaisuja, esimerkiksi jatkuvan integraation palvelu Jenkins. Se tarjoaa erilaisia mittareita yksikkötestauksen etenemisen seurantaan ja sen jälkeisten vaiheiden automaattiseen käynnistämiseen [1]. Jenkinsin liitännäiset myös mahdollistavat sen soveltumisen hyvin erilaisiin ohjelmistoprojekteihin. Lisäksi Jenkinsin luotettava jatkuvan integraation palvelu mahdollistaa ohjelmistojen jatkuvan testaamisen ja ohjelmistokehityksen helpon integroimisen osaksi testiautomaatiota. [17] Tässä diplomityössä kyseessä on loppujen lopuksi tämä sama rajapintaongelma, eli kuinka ratkaista kahden erilaisen testiympäristön välinen rajapintaongelma siten, että se mahdollistaisi jatkuvan integraation.

3. JÄRJESTELMÄINTEGRAATION SUUNNITTELUMALLIT JA NIIDEN KOMMUNIKAATIO

Tämä luku on jaettu viiteen alalukuun. Ensimmäisessä alaluvussa käsitellään yleisellä tasolla järjestelmäintegraation hyötyjä ja haasteita. Toisessa alaluvussa jaotellaan järjestelmäintegraatiot kahteen yläluokkaan eli orkestrointiin ja koreografiaan. Kolmannessa alaluvussa viittä keskeisintä järjestelmäintegraation systeemiarkkitehtuurista mallia. Neljännessä alaluvussa käsitellään kaksi viestinnän ja informaation integraatiomallia toisen alaluvun pohjalta. Viides alaluku käsittelee REST-arkkitehtuurin teoriaa. Kuudes alaluku käsitellee kommunikaatiota ja tärkeimpiä kommunikaatiotekniikoita eli unicastia, broadcastia, multicastia ja anycastia. Seitsemäs alaluku käsittelee tietokantoja tarkastellen yleisesti relaatio- ja oliotietokantojen eroja sekä kuinka ne soveltuvat hajautetun järjestelmän integraatioon.

3.1 Järjestelmäintegraation hyödyt ja haasteet

Tässä luvussa käsitellään järjestelmäintegraatiota teorian näkökulmasta. Luvussa kerrotaan, mitä syitä järjestelmäintegraatioon on, ja mitä hyötyjä ja haasteita järjestelmäintegraatioon voi liittyä.

3.1.1 Järjestelmäintegraation hyödyt

Järjestelmäintegraation tavoitteet ja hyödyt voidaan jaotella kolmeen erilliseen vaiheeseen. Ensimmäinen on datan kerääminen useista eri järjestelmistä yhteiseen tietokantaan. Tähän vaiheeseen kuuluu myös datan välittäminen järjestelmän eri osille toiminnan tehokkuuden ja -valvonnan parantamiseksi. Tämä voi tarkoittaa erilaisien yhteisten tietokantojen käyttöönottoa. [4, s. 32]

Toisessa vaiheessa integroitavia järjestelmiä yhdistetään oliotasolla ja tarjotaan yhteisiä metodeja tiedon hallinnointiin, hakemiseen ja keräämiseen. Tämä tarkoittaa kommunikaatiota sovelluksien ulkopuolisien olioiden kanssa. [4, s. 32]

Kolmannessa vaiheessa järjestelmien prosesseja integroidaan toisiinsa luomalla erilaisia työnkulullisia hallintajärjestelmiä, jossa integroitavien järjestelmien tapahtumia linkitetään toisiinsa yhteisiksi prosessiketjuiksi. Tällä tavoin pyritään luomaan uusia tuotannollisia prosesseja ja uusia seurantamahdollisuuksia prosesseille. [4, s. 32]

Integraation tavoitteena on luoda yhteisiä tiedonkeruuratkaisuja, yhteisiä metodeja tiedon käsittelyä varten ja ratkaista, kuinka metodit voidaan yhdistää yhteisiksi prosesseiksi. Integroiminen voi siis parantaa erillisten järjestelmäosien seuraamista ja hallinnointia. Lisäksi se voi luoda uusia prosessiketjuja ja niiden seurantamahdollisuuksia.

Järjestelmäintegraation perimmäisenä tarkoituksena on saada järjestelmät kommunikoimaan ja myös tekemään uudesta kokonaisesta järjestelmästä entistä joustavamman ja laajennettavamman [20, s. 27]. Tämän tekemiseen on useita eri arkkitehtuurisia vaihtoehtoja, joissa jokaisessa on hyviä ja huonoja puolia.

3.1.2 Järjestelmäintegraation haasteet

Luvussa 3.1.1 käsitellyt järjestelmäintegraation tavoitteet aiheuttavat kuitenkin haasteita, jotka tulee arkkitehtuuriratkaisua mietittäessä huomioida parhaan lopputuloksen saavuttamiseksi. Ensimmäinen ongelma on se, kuinka mahdollistaa integraatio siten, että järjestelmäosat voivat yhdessä muodostaa uusia tiedonkeruutapoja, metodeja ja prosesseja ilman, että järjestelmän osien erillinen laajennettavuus ja ylläpito vaarantuu. Tämä tarkoittaa sitä, että järjestelmien osien tulisi pyrkiä pysymään toimintakykyisinä, vaikka jokin integroiduista järjestelmistä kaatuisi tai muuttuisi ajan kuluessa.

Toisena haasteena on se, kuinka integroidut järjestelmät saadaan kommunikoimaan keskenään. Kommunikaation puute on keskeisessä roolissa järjestelmän integraation onnistumisen kannalta [21, luku 4.1.1]. Tämä sisältää päätöksen siitä, toteutetaanko kommunikaatio tasa-arvoisessa verkkorakenteessa eli koreografioidusti vai vastaako joku järjestelmän osa kommunikaatiosta eli orkestroi järjestelmää.

Kolmantena haasteena on tiedon hallinnointi ja varastointi eli miten järjestelmien osa-alueet tallentavat tietoa, kuinka tietoon pääsee käsiksi, kuinka tietokannasta saadaan haettua tietoa ja miten varmistaa tiedon oikeellisuus. Lisäksi tulee päättää, vastaavatko järjestelmien osat erikseen tiedon varastoinnista vai varastoidaanko tieto keskitetysti.

Neljäntenä haasteena on, kuinka tarjota rajapinnat järjestelmien väliseen kommunikaatioon, tiedon varastointiin ja hallinnointiin. Tätä päätettäessä on tiedettävä tarvittavan integraation laajuus, tavoitteet ja mikä integraatioratkaisu ei ole liian raskas halutun lopputuloksen saavuttamiseksi. Mikäli tarvittavia rajapintoja ei ole olemassa tai niistä ei ole riittävästi tietoa, voi integroiminen muuttua etenkin suurien järjestelmien integroimisessa hyvin vaikeaksi [21, luku 4.1.10].

Viidentenä haasteena on semanttiset haasteet, eli onko kommunikoinnissa ja tiedon varastoinnissa käytettävät tiedot missä kontekstissa. Esimerkiksi jos kommunikaatiossa järjestelmän osa A saa osalta B datan osana päivämäärän: ”1. Heinäkuuta”, järjestelmän osa B ei automaattisesti tiedä missä muodossa A on tarkoittanut sen olevan. Ongelmana voi olla esimerkiksi tarkoittaako päivä päivämäärää vai pitäisikö se tulkita tekstinä osana suurempaa datakokonaisuutta. Esimerkiksi suurempi datakokonaisuus voisi olla: ”Johannan syntymäpäivä 1. Heinäkuuta”. Semanttista ongelmaa varten tarvitaan yleensä jokin yhteinen tapa kommunikoida siten, että järjestelmien osat tietävät missä muodossa dataa siirretään. [4, s. 30–31]

Kuudentena haasteena on dokumentaation puute. Integrointi voi vaikeutua huomattavasti, mikäli integroitavia järjestelmiä ei ole dokumentoitu kunnolla ja se on esimerkiksi vajaata tai vanhentunutta. Pahimmassa tapauksessa dokumentaatiota ei ole tehty ollenkaan, jolloin järjestelmän integroiminen voi muuttua käytännössä mahdottomaksi. [21, luku 4.1.2]

Seitsemäntenä haasteena on arkkitehtuurin valinnan mahdollinen epäonnistuminen. Arkkitehtuurin valinta voi epäonnistua esimerkiksi dokumentaation puutteen takia. Tämä voi tarkoittaa sitä, että järjestelmän kehittäjillä on puutteellinen tai virheellinen käsitys integroitavista järjestelmistä. Epäonnistunut arkkitehtuuriratkaisu voi johtaa ongelmiin kehityksen aikana, ja arkkitehtuurin valinnan onnistuminen on keskeinen osa tämän luvun aiempien haasteiden ratkaisussa. Tämän takia on tärkeää arvioida integroitavaa järjestelmää kokonaisuutena ja vertailla eri arkkitehtuuriratkaisuja käyttäen avuksi esimerkiksi UML-kaavioita. [21, luku 4.1.4]

3.2 Orkestraatio vai Koreografia

Eri arkkitehtuuriratkaisut voidaan karkeasti jaotella orkestraatioon ja koreografiaan, jotka tunnetaan myös nimillä koostuminen (engl. composition) ja koordinaatio (engl. coordination) [12, s. 185]. Koreografiassa järjestelmän osat kommunikoivat keskenään verkostomaisessa rakenteessa ja ovat keskenään tasa-arvoisia, kun taas orkestraatiossa yksi keskitetty järjestelmän osa ohjaa koko järjestelmän kommunikaatiota ja toimintaa. Toisin sanottuna koreografiassa jokainen järjestelmä keskittyy yksittäiseen pariin, ja orkestraatiossa sen sijaan tarkkaillaan järjestelmää ylhäältä päin katsottuna [12, s. 228].

Koreografiaa ovat esimerkiksi tässä luvussa käsiteltävät point-to-point- ja pipeline-pohjaiset arkkitehtuuriratkaisut, joissa jokainen järjestelmän osa on keskenään tasa-arvoisia. Orkestraatio pohjaisia arkkitehtuureja ovat esimerkiksi hub-and-spoke-

pohjainen arkkitehtuuri, jossa keskitetysti hubi ohjaa kommunikaatiota. Sen sijaan SOA voi olla orkestraatiota tai koreografiaa riippuen verkon rakenteesta ja siellä sijaitsevista järjestelmistä. SOA:ssa tärkeää ovat yhteiset kommunikaatioprotokollat ja rajapintarakenteet, ei niinkään järjestelmän osien väliset valtasuhteet.

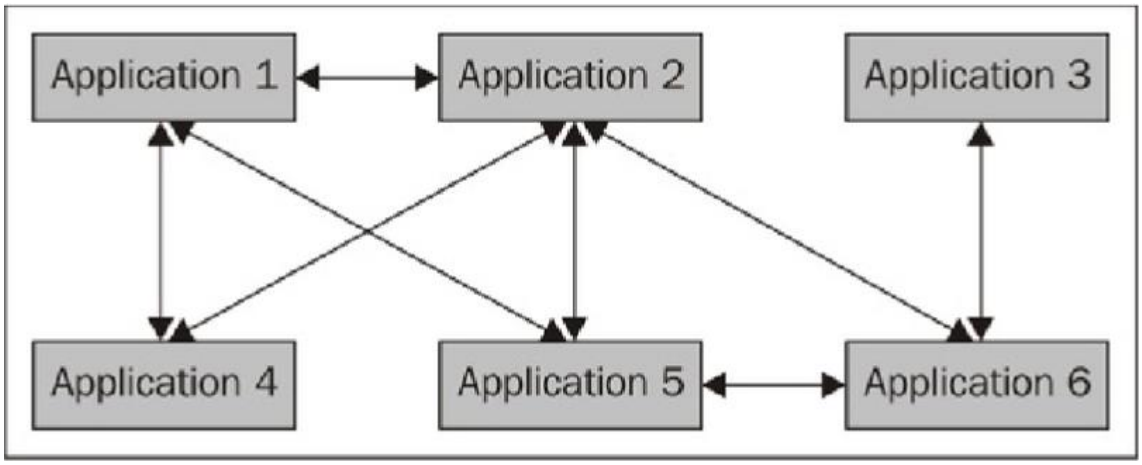
Lisäksi orkestraatiosta puhuttaessa voidaan puhua yleisesti middlewaresta. Middlewarella tarkoitetaan kommunikaatioarkkitehtuurin osaa, joka sallii usean eri komponentin kommunikoida keskenään tämän kautta. Middleware ei välitä kommunikoivan komponentin ohjelmistokielestä tai käyttämistä protokollista. Näin ollen middlewareja voi olla hyvin erilaisia ja ne voidaan jakaa kommunikaatiotavasta riippuen keskustelu-, pyyntö- tai vastauspohjaisiin. Lisäksi ne voidaan jakaa viestit välittäviin, viestijonoihin ja tilauspohjaisiin middlewareihin. [4, s. 40]

3.3 Integraatiomallit ja niiden ominaisuudet

Tässä luvussa käsitellään keskeisimpiä integraatioarkkitehtuurin variantteja, jotka ovat point-to-point-, hub-and-spoke-, pipeline-, palvelupohjainen- ja mikropalveluarkkitehtuuri (engl. service-oriented-architecture, SOA). Lisäksi luvussa toteutetaan integraatiomallien vertailua paitsi yleisesti, mutta myös tämän diplomityön näkökulmasta. Näistä arkkitehtuurimalleista neljään päädyttiin sen takia, että ne on listattu keskeisimmiksi integraatioarkkitehtuurin varianteiksi tämän luvun keskeisessä lähteessä [4, s. 25]. Lisäksi lukuun on lisätty mikropalveluarkkitehtuuri, koska se on SOA:n lisäksi nykyään hyvin laajasti käytetty integraatiomalli.

3.3.1 Point-to-Point

Point-to-point-integraatiolla tarkoitetaan arkkitehtuuria, joka koostuu useista itsenäisistä sovelluksista, jotka kommunikoivat keskenään tasavertaisesti. Point-to-point-integraatiossa sovellukset ovat yhdistettyinä toisiinsa verkkomaisena rakenteena, kuten on nähtävissä alla olevasta kuvasta 12. [4 s. 45].



Kuva 12. Point-to-point-arkkitehtuuri [4, s. 45].

Point-to-point-arkkitehtuuri on yleensä ensimmäinen ja alkuun yksinkertaisin integraation muoto, jossa jokainen sovellus kustomoidaan erikseen kommunikoimaan keskenään halutulla tavalla. Tämä johtaa siihen, että sovelluksien määrään nähden sovelluksien välisiä yhteyksiä tarvitsee olla moninkertaisesti. [7, s. 117] Järjestelmän laajentuessa point-to-point-arkkitehtuuri voi osoittautua turhan monimutkaiseksi ylläpitää eikä sillä saa toteutettua kaikista parhaimpia integraation tuloksia [7, s. 118].

Point-to-point-arkkitehtuurin etuna on, että se ei tarvitse paljon uuden suunnittelua, sillä jo valmiiksi olevat järjestelmät pääosin kustomoidaan toisiinsa verkostomaiseksi rakenteeksi. Ongelmana on kuitenkin se, että point-to-point-järjestelmän laajentuessa integraatiotapa käy nopeasti monimutkaiseksi, kalliiksi laajentaa ja vaikeaksi ylläpitää joustamattomuuden ja standardien puutteen takia. Koreografioidun järjestelmän takia myös datan kerääminen keskitettyyn lähteeseen ei ole mahdollista. [4, s. 45]

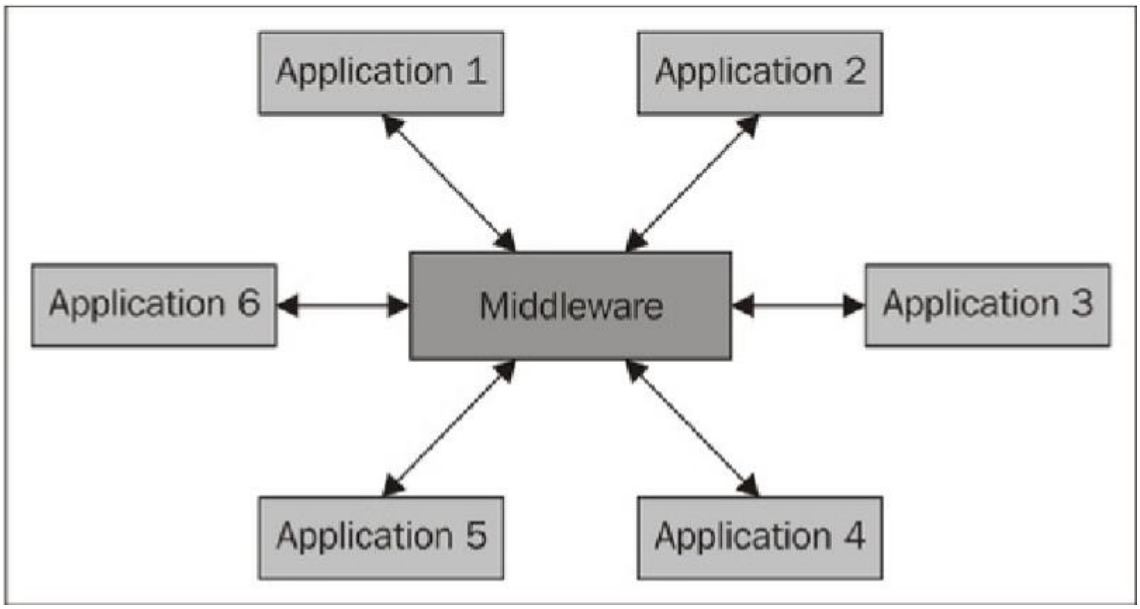
Voidaan näin käytännön läheisesti ajatella, että point-to-point arkkitehtuurissa järjestelmät kytketään toisiinsa yksittäisillä johdoilla, joista jokainen voi olla erilainen ja erilaisilla liittimillä. Kahden järjestelmän integroinnissa tämä ei ole ongelma. Siitä voi kuitenkin tulla ongelma, sillä mitä enemmän järjestelmiä integroidaan, sitä enemmän yksilöllisiä johtoja tarvitaan. Järjestelmien määrä voi kasvaa, kunnes järjestelmän kokonaiskuvan näkeminen ja sitä kautta järjestelmän ylläpito ja laajentaminen muuttuvat hyvin vaikeaksi, ellei mahdottomaksi.

Point-to-point-järjestelmä sopeutuu parhainten integraatioon, jossa integroitavia järjestelmiä ei ole liian paljon ja integraatio halutaan tehdä nopeasti ja tehokkaasti ilman datan keruuta. Myös yhteyksien määrä sovellusta kohden tulee olla rajallinen, jotta järjestelmien rajapinnat eivät muutu liian monimutkaisiksi. Jos näin ei ole, on kannattavampaa miettiä muita integraatioarkkitehtuurisia ratkaisuja. Tämä tarkoittaa, että integraatio toteutetaan hyvin matalalla tasolla ilman yhtä kattavia integraatioetuja,

kuin muissa integraatioarkkitehtuurisissa vaihtoehdoissa. Hajautetun järjestelmän integraatiota pohtiessa point-to-point-arkkitehtuuri olisi huonosti sopeutuva yhteyksien standardoinnin ja yhteisen ohjelmistorajapinnan puuttumisen takia. Lisäksi yksi järjestelmäintegraation keskeisimmistä hyödyistä on keskitetty tietokanta, kuten mainitaan luvussa 3.1.1. Tämä ei ole toteutettavissa point-to-point-arkkitehtuurissa. Tämän takia voi usein olla kannattavampaa käyttää muita järjestelmäintegraatiomalleja hajautetun järjestelmän integroimiseen.

3.3.2 Hub-and-spoke

Hub-and-spoke-arkkitehtuurissa tarkoitetaan arkkitehtuuriratkaisua, jossa keskitetysti hallitaan koko järjestelmän kommunikaatiota (Kuva 13) [4 s. 25]. Hub-and-spoke-järjestelmän keskitetty ratkaisu mahdollistaa laajemman integraation verrattuna point-to-point-arkkitehtuuriin [4, s. 47]. Tämä tarkoittaa sitä, että hub-and-spoke-arkkitehtuurin keskitetty middleware eli keskushubi mahdollistaa keskitetyn datan keräämisen ja prosessoinnin toisin kuin point-to-point-arkkitehtuuri. Lisäksi hub-and-spoke-arkkitehtuuri mahdollistaa yhteisen ohjelmistorajapinnan eri ohjelmistojen välille sekä sitä kautta yhteisen kommunikaatiostandardin [4, s. 47]. Hub-and-spoken erottaa monista muista arkkitehtuureista kyky mahdollistaa kevyet yhdistystavat. Tällä tarkoitetaan sitä, että yhdistettäviin sovelluksiin ei tarvitse tehdä ollenkaan tai hyvin vähän muutoksia integraation mahdollistamiseksi. Tämä johtuu siitä, että kaikki yhteyteen vaadittavat rajapinnat ja yhdistyksen mahdollistavat metodit luodaan hub-and-spoke-arkkitehtuurin sisään. [22, s. 38]



Kuva 13. Hub-and-spoke-arkkitehtuuri [4 s. 47].

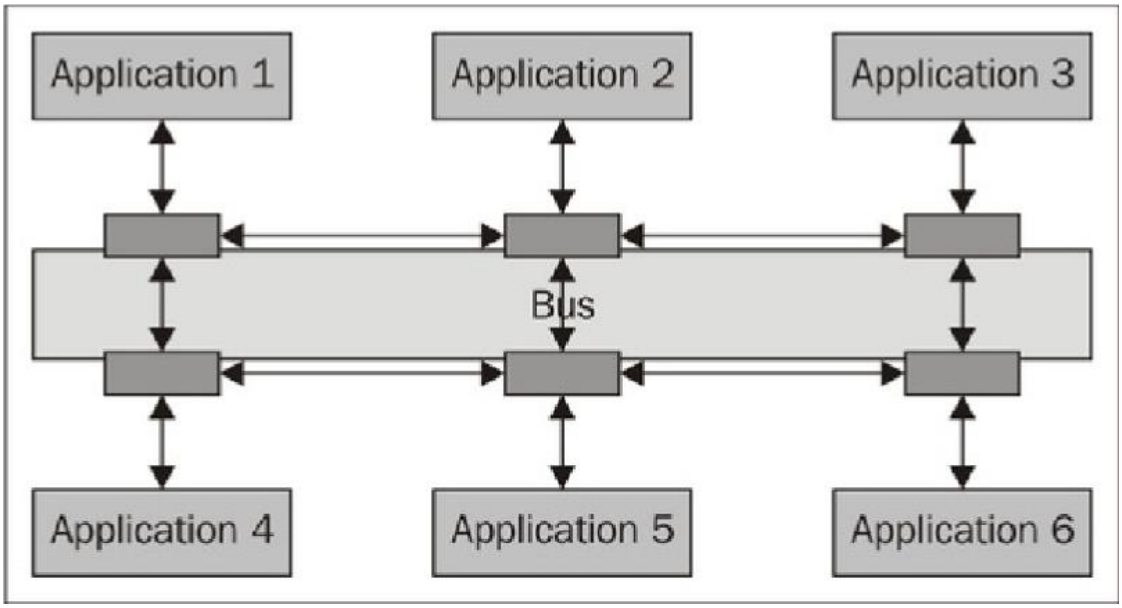
Hub-and-spoken etuna on laajemmat integraatiomahdollisuudet ja yksinkertaisemmat verkkorakenteet etenkin suurissa verkoissa. Tämä johtuu siitä, että yhteyksien määrä suhteessa sovelluksien määrään on point-to-point-arkkitehtuuria vähäisempi. Hub-and-spoke on korkeiden aloituskustannuksien jälkeen helpompi laajentaa ja muuttaa yhteisien standardien ansiosta. Haittapuolena on se, että hub-and-spoke integraatio on kalliimpi ja vaatii alussa enemmän työtä. Lisäksi se vaatii taustalleen monimutkaisemman rakenteen eli yleensä verkkoinfrastruktuurin. Kuitenkin järjestelmän valmistuttua edut verrattuna point-to-point-arkkitehtuuriin ovat selkeät lukuun ottamatta sitä, että suurissa datamäärissä keskushubista voi tulla järjestelmän kannalta pullonkaula. Tämä tarkoittaa myös sitä, että hub-and-spoken kaatuessa koko järjestelmäintegraatio lakkaa toimimasta. Tästä haittapuolesta ei voi kuitenkaan päästä eroon, sillä koko hub-and-spoken ideana on mahdollistaa yksi middleware-sovellus arkkitehtuuriin, joka hallitsee koko järjestelmää. [4, s. 47] [22, s. 38]

Hub-and-spoke-arkkitehtuuri soveltuu hyvin hajautetun järjestelmän integraatioon sillä, se tarjoaa yhteisen kommunikaatiostandardin yhteisen ohjelmistorajapinnan kautta, joita kaikki sovellukset voivat käyttää. Joustavan ohjelmistorajapinnan kautta järjestelmään on helppo integroida uusia sovelluksia tarpeen vaatiessa, ja joustava rajapinta mahdollistaa hyvin erilaisten järjestelmien integroimisen.

3.3.3 Pipeline

Pipeline-arkkitehtuuri koostuu putkista ja suodattimista, jotka jakavat järjestelmän useisiin sarjamaisiin prosessivaiheisiin. Vaiheissa datan virta kulkee putkessa suodattimien läpi, jotka muuttavat viiveellä prosessin dataa vaiheittain. (Kuva 14) [6, luku 2]

Pipeline-arkkitehtuurissa eri järjestelmän osat toimivat itsenäisesti osana arvoketjua. Arvoketjussa jokainen järjestelmä hoitaa osansa aina alusta loppuun saakka, kunnes saavutetaan haluttu lopputulos [4 s. 25]. Pipeline-arkkitehtuuri muistuttaa osaltaan hub-and-spoke-rakennetta, sillä molemmissa on middleware eli keskushubi, joka toimii operatiivisena keskuksena eri sovellusten välillä. Erona hub-and-spoke-rakenteeseen on, että sovellukset ovat kytketty keskusväylään (engl. central-bus), joka tarjoaa sovelluksille paikallisen pääsyn väylän yhteiseen käyttöliittymään. Tämä mahdollistaa hub-and-spoken tapaan yhteisen käyttöliittymästandardin, joka helpottaa kommunikaatio ja käyttöliittymä ongelmia toisin kuin point-to-point-integraatiossa. Haittapuolena on sen sijaan se, että pipeline-arkkitehtuuri ja sen tarvitsema infrastruktuuri on vaikeampi rakentaa. [4, s. 48]



Kuva 14. Pipeline arkkitehtuuri [4, s. 48].

Pipeline-arkkitehtuuri on väylärakenteen takia erittäin hyvä erilaisten pitkien toisistaan erillään olevien prosessien integroimiseen yhdeksi prosessiketjuksi. Tällaisia voivat olla esimerkiksi ohjelmistokehityksessä yksikkötestaus, jossa luodaan pitkiä työnkulkuketjuja

(engl. workflow). Tästä esimerkkinä on Jenkins-pipeline, jota käsitellään tarkemmin luvussa 2.3.1.

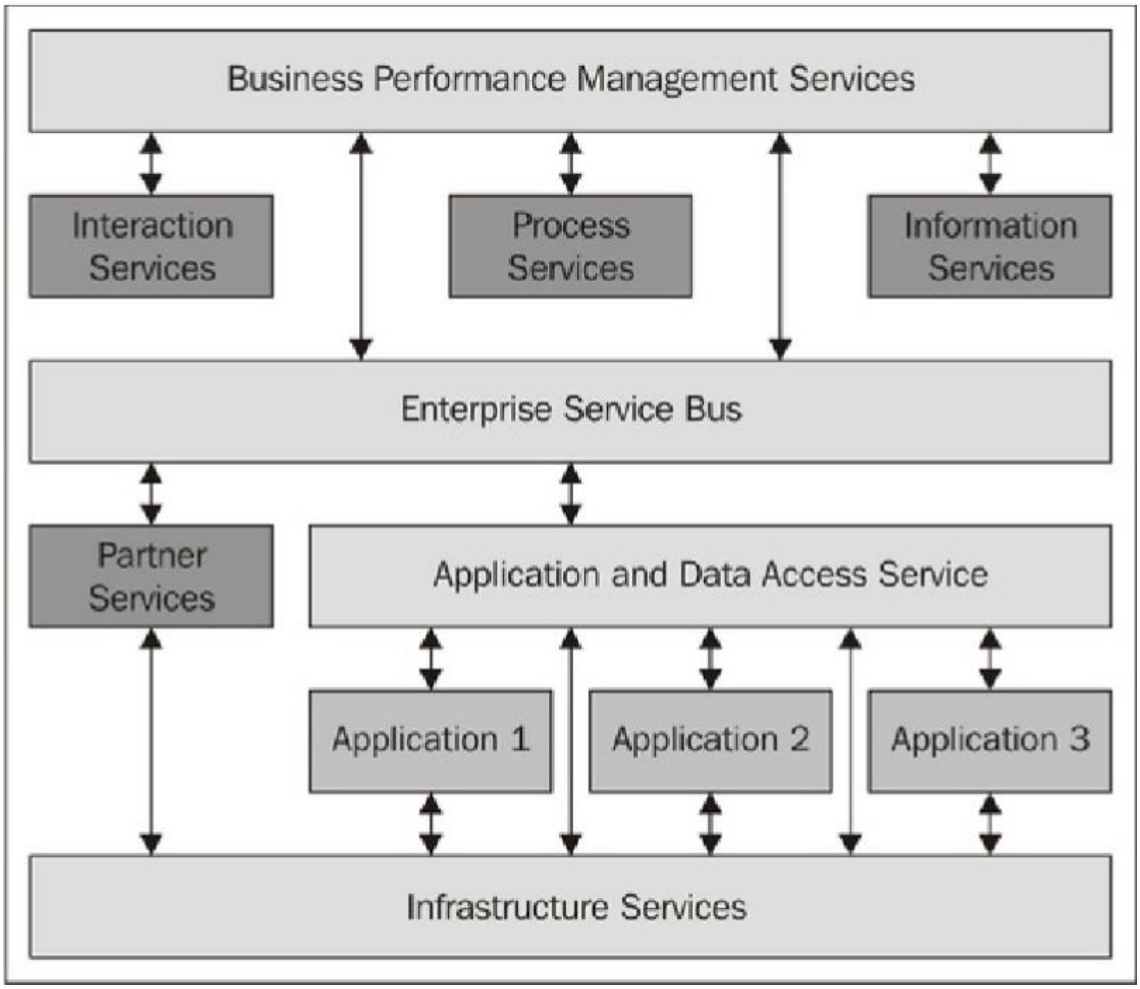
Pipeline arkkitehtuuri myös mahdollistaa keskitetyn datan keräämisen ja hyödyntämisen, samoin kuin hub-and-spoke-integraatio. Pipeline arkkitehtuuri myös toimii väylärakenteen ansiosta broadcastina eli jokainen viesti, joka väylää pitkin lähetetään, saapuu jokaiselle sovellukselle väylän varrella. Kommunikaatio on tärkein ero hub-and-spoke-arkkitehtuuriin, joka toimii yleensä unicastilla. [4, s. 49]

Pipeline arkkitehtuuri soveltuu hyvin hajautetun järjestelmän integraatioon, mikäli integraation tavoitteena on muodostaa erilaisia arvoketjuja, jossa järjestelmän sovellukset ajetaan ketjussa. Samoin kuin hub-and-spoke, pipeline-arkkitehtuuri tarjoaa myös yhteisen rajapinnan järjestelmien kommunikointia varten, mikä helpottaa hajautetun järjestelmän laajentamista.

3.3.4 Palvelupohjainen arkkitehtuuri

Palvelupohjaisessa arkkitehtuurissa (SOA) integroitavan järjestelmän osat pidetään selkeästi toisista erillään [4, s. 50] [22, s. 39]. Järjestelmän osat kommunikoivat yhteisellä protokollalla ja rajapinnalla, joko keskenään tai keskitetysti. Sen järjestelmät toimivat myös täysin itsenäisesti välittämättä muiden osa-alueiden tilasta (Kuva 15). Tavoitteena on ennen kaikkea saada data keskitettyyn sijaintiin ja mahdollistaa avoin ja yhteinen standardi integraation mahdollistamiseksi ilman, että järjestelmän osien erillinen toiminta ja kehittäminen vaarantuu. [4 s. 28–32]

SOA:n keskeisenä ideana on rakentaa integraatio yhteisen väylän eli ”enterprise service busin” (ESB) päälle [4, s. 50] [22, s. 39]. ESB tarjoaa sovelluksille eri rajapintojen kautta pääsyn prosessitason eri palveluihin. ESB:llä on paljon samaa hub-and-spoke-arkkitehtuurin keskushubiin ja pipeline-arkkitehtuurin keskusväylään. Tärkeintä on kuitenkin se, että se mahdollistaa useiden toisistaan täysin itsenäisten sovelluksien ja prosessien yhdistämisen ESB:hen omien rajapintojen kautta. [4, s. 50] Tämä tarkoittaa sitä, että ESB toimii tilattomasti eikä asiakkaan tarvitse välittää siitä missä päin verkkoa käytettävä palvelu sijaitsee. [4, s. 38] ESB on eräänlainen middleware, joka orkestroi SOA-arkkitehtuurin kommunikaatiota hub-and-spoke-arkkitehtuurin hubin tapaan. ESB tarjoaa hub-and-spoke arkkitehtuurin tapaan kevyen ja älykkään tavan liittyä osaksi verkkoa. Tämä tarkoittaa sitä, että SOA-arkkitehtuurin sovellukset käyttävät omaa yhteistä rajapintaa. Tämän avulla kommunikaatio verkossa voidaan toteuttaa mahdollisimman vähällä sovelluksen muokkaamisella. [22, s. 39]



Kuva 15. Palvelupohjainen arkkitehtuuri [4, s. 50].

SOA on yleisesti käytetty arkkitehtuuritapa verkkopohjaisten palveluiden kehityksessä, jossa simple-object-access-protocol (SOAP) ja representational-state-transfer (REST) ovat kaksi tapaa toteuttaa SOA:n pohjainen arkkitehtuuri [23, s. 207]. Näistä SOAP on enemmän kommunikaatioprotokolla, kun taas REST on arkkitehtuuritapa verkkopohjaisille palveluille [23, 210] [23, s. 211]. REST-arkkitehtuuria käsitellään tarkemmin luvussa 3.5.

SOA soveltuu hyvin hajautetun järjestelmän integraatioon, ja yhdistää pipeline-arkkitehtuurin ja hub-and-spoke-arkkitehtuurin hyviä puolia. SOA voi kuitenkin olla ylläpidon kannalta tarpeettoman raskas, jonka takia hub-and-spoke ja pipeline-arkkitehtuurimallit voivat tilanteen mukaan olla riittäviä hajautetun järjestelmän integraatioon. SOA arkkitehtuuri tarjoaa myös yhteisen rajapinnan järjestelmien kommunikointia varten, mikä helpottaa hajautetun järjestelmän integraatiota.

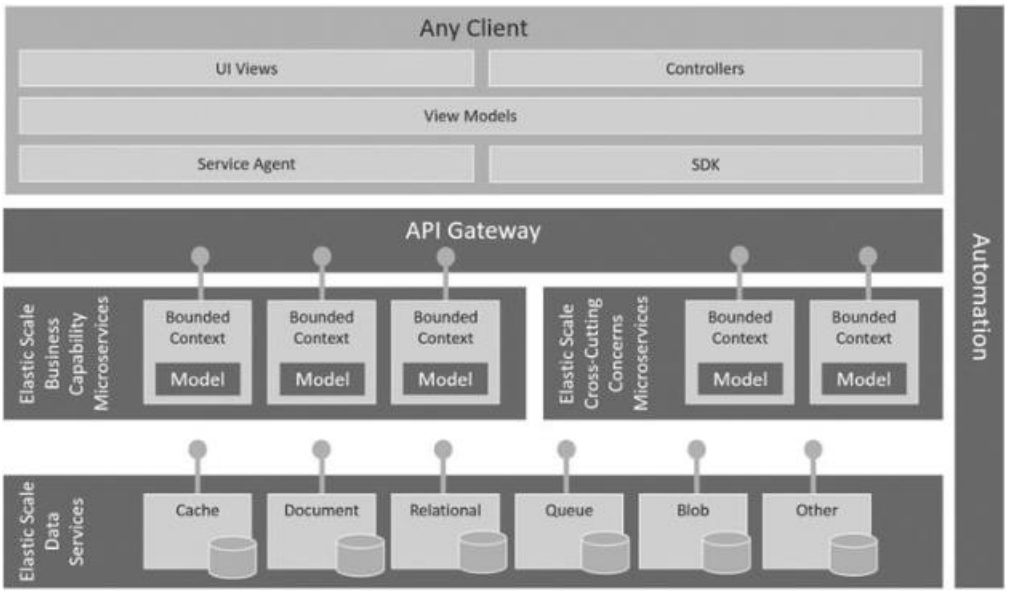
3.3.5 Mikropalveluarkkitehtuuri

Mikropalveluarkkitehtuuri (engl. microservices architecture) pohjautuu ajatukseen sovelluksesta palveluna. Mikropalveluarkkitehtuuri on vaihtoehto valtaville sovelluksille, joissa sovellus tarjoaa lukemattomia palveluita. Sen sijaan mikropalveluarkkitehtuuri koostuu mikropalveluista (engl. microservice), joista jokainen tekee vain yhden palvelun hyvin. Ideana on tarjota lukuisia mikropalveluita, joilla on yksinkertainen rajapinta ja lisätä näitä mikropalveluita tarpeen mukaan ja näin kasvattaa järjestelmää. [24, luku 1]

Mikropalvelut ovat luonteeltaan automatisoituja eli toimivat toisistaan itsenäisesti, mutta myös eristettyjä eli ne voivat toimia hajautetusti eri aikoina. Tämä tarkoittaa sitä, että ne voivat toimia itsenäisesti vain löyhästi kytkettyinä toisiinsa. Lisäksi niitä voidaan kehittää, testata ja julkaista itsenäisesti. Mikropalveluiden etuna on, että ne ovat joustavia, kestäviä ja toipuvat toisistaan riippumatta ongelmatilanteissa. Lisäksi ne reagoivat nopeasti toisistaan riippumatta. [24, luku 1]

Mikropalvelut kuitenkin vaativat alustakseen palvelun, jossa on laaja tarjonta erilaisia tilallisia ja tilattomia palveluita. Tämän voi saavuttaa esimerkiksi Azuren kaltaisella pilvipalvelulla. Mikropalveluiden lisäksi tarvitaan myös erillinen käyttöliittymä, josta voidaan monitoroida, hallita, laajentaa, konfiguroida ja suorittaa eri palveluita. Mikropalvelu on enemmän kuin vain REST API (Application programming interface), sillä se on joukko APE:ja, joita hallitaan yhteisestä käyttöliittymästä [24, luku 1]

Mikropalveluarkkitehtuuri pohjautuu ajatukseen kolmesta järjestelmän tasosta, jossa erotellaan, esitys, yritys ja data omiin tasoihin. Tässä mikropalvelu arkkitehtuuri eroaa muista arkkitehtuurimalleista siinä, että palvelut pilkotaan useisiin pienempiin palveluihin kuten on nähtävissä kuvasta 16. [24, luku 2]



Kuva 16. Mikropalvelu arkkitehtuuri [24, luku 2]

Nämä eri palvelut sijaitsevat pilvessä ja niihin voidaan olla yhteydessä erillisen porttina toimivan mikropalvelun kautta riippumatta laitteesta. Portin kautta pääsee käsiksi eri mikropalveluihin, niihin voi rekisteröityä, niitä voidaan tilata ja niihin voidaan lisätä tietoa. [24, luku 2]

Mikropalvelut voivat sopia hyvin hajautetun järjestelmän integraatioon hajautetuista järjestelmistä riippuen. Jos hajautetut järjestelmät ovat hyvin erilaisia ja ne tarvitsevat omat palvelut sekä tietokannat, mikropalveluarkkitehtuurin käyttäminen voi olla aiheellista. Kuitenkin jos palveluita ei ole montaa ja ne ovat hyvin samankaltaisia, mikropalveluille ei välttämättä ole yhtä suurta tarvetta, sillä palvelupohjaisella arkkitehtuurilla voi toteuttaa saman asian.

3.3.6 Integraatiomallien vertailua

Tässä luvussa verrataan aiemmissa alaluvuissa käsiteltyjä integraation arkkitehtuurimalleja käymällä läpi niiden vahvuuksia ja heikkouksia taulukkoa apuna käyttäen (Taulukko 1). Luvussa kootaan ylös edellä käytyjen alalukujen tieto helposti vertailtavaan muotoon, kun arvioidaan mahdollisia arkkitehtuuriratkaisuja hajautetun järjestelmän integraatioon.

Taulukko 1. Integraation arkkitehtuurimallien vertailua [4, 7].

Arkkitehtuurimalli	Edut	Haitat
Point-to-point	- Edulliset aloituskustannukset [4, s. 46] - Erilliset ja autonomiset järjestelmät [4, s. 46]	- Suuri yhteyksien määrä [7, s. 118] - Joustamattomuus laajentuessa [7, s. 118], [4, s. 46] - Ei yhteistä rajapintaa [4, s. 46] - Ei tue keskitettyä datan keräämistä [4, s. 46] - Kallis laajentaa [4, s. 46] - Standardoinnin puute [4, s. 46]
Hub-and-spoke	- Yhteinen rajapinta ja standardit [4, s. 47] - Edullinen laajentaa ja ylläpitää [4, s. 47] - Helppo monitoroida [4, s. 47] - Mahdollistaa keskitetyn datan keräämisen	- Kalliit aloituskustannukset [4, s. 47] - Vaatii raskaamman infrastruktuurin tueksi [4, s. 47] - Korkeilla tiedonsiirtomäärillä keskus voi ylikuormittua [4, s. 47] - Keskushubin kaatuminen kaataa koko järjestelmän [4, s. 47]
Pipeline	- Erittäin joustava [4, s. 48] - Edullinen laajentaa ja ylläpitää [4, s. 48] - Yhteiset rajapinnat ja standardit [4, s. 48] - Mahdollistaa autonomiset arvoketjut [4, s. 48]	- Kalliit aloituskustannukset [4, s. 48] - Vaatii raskaamman infrastruktuurin tueksi [4, s. 47] - Suurilla datansiirtomäärillä on vaarana muodostaa pullonkauloja [4, s. 49]

	- Mahdollistaa keskitetyn datan keräämisen	
Palvelupohjainen arkkitehtuuri	- Edullinen laajentaa [4, s. 50] - Erittäin joustava arkkitehtuuri [4, s. 50] - Tukee useita standardeja [4, s. 50] - Eri sovellukset voi helposti liittää järjestelmään [4, s. 50]	- Kalliit aloituskustannukset [4, s. 50] - Vaatii tarkkaa strategiaa ja hallinnointia järjestelmän ylläpidon ja laajennuksen mahdollistamiseksi [4, s. 50]
Mikropalvelu arkkitehtuuri	- Erittäin joustava [24, luku 1] - Kestäviä [24, luku 1] - Reagoi nopeasti [24, luku 1] - Edullinen laajentaa [21, luku 1] - Voi tarjota yhtenäisen rajapinnan [24, luku 2]	- Vaatii saavutettavan alustan, jossa mikropalvelut toimivat. Esim. pilvipalvelu. [24, luku 1] - Monimutkainen rakenne vaatii tarkkaa strategiaa ja hallinnointia [24, luku 2] - Vaatii erillisen käyttöliittymän [24, luku 1]

Yllä olevan taulukon (Taulukko 1) pohjalta voidaan todeta, että vaikka point-to-point-arkkitehtuuri on kaikista kevyin aloitusvaiheessa, niin sen joustamattomuuden takia sillä ei pystytä tekemään kovin korkeatasoista integraatiota [7, s. 118], [4, s. 46]. Sen sijaan hub-and-spoke-arkkitehtuuri, pipeline-arkkitehtuuri, palvelupohjainen arkkitehtuuri ja mikropalveluarkkitehtuuri ovat hyviä järjestelmäintegraation arkkitehtuurimalleja myös korkeamman tasoiselle integraatiolle, vaikka niiden aloituskustannukset voivat olla point-to-point-arkkitehtuuria suuremmat.

Pipeline arkkitehtuuri soveltuu etenkin, jos sovellukset halutaan yhdistää autonomiseksi arvoketjuksi esimerkiksi prosessiluontoisessa integraatiossa, jossa ketjun viidennen sovelluksen lopputulos riippuu neljän aiemman sovelluksen lopputuloksesta [4, s. 48]. Sen sijaan hub-and-spoke-arkkitehtuurissa arvoketjua ei välttämättä ole. Sen sijaan siinä pyritään yhdistämään N määrä sovelluksia toisiinsa ilman varsinaista arvoketjua mahdollistaen yhteisen rajapinnan ja standardit [4, s. 47]. Palvelupohjainen arkkitehtuuri yhdistää pipeline ja hub-and-spoke-arkkitehtuurien hyviä puolia, mutta on raskas

aloituskustannuksilta sekä vaatii suunnitelmallisen strategian järjestelmän ylös ajoon, ylläpitoon ja laajentamiseen [4, s. 50]. Mikropalveluarkkitehtuuri tarjoaa paljon samoja palveluita mitä palvelupohjainen arkkitehtuuri, sillä se on joustava, helppo aloittaa ja laajentaa. Tästä huolimatta se vaatii taustalleen pilvipalvelun. Lisäksi sen monimutkainen rakenne voi vaatia monimutkaista hallinnointia ja erillisen käyttöliittymän. [24, luku 1 ja 2] Tämä johtaa siihen, että riippuen integroitavasta hajautetusta järjestelmästä palvelupohjainen arkkitehtuuri voi olla parempi ratkaisu.

Yhteenvetona voidaan sanoa, että arkkitehtuurimalleista hajautetun järjestelmän integraatioon soveltuvat parhainten pipeline-, hub-and-spoke-, palvelupohjainen-arkkitehtuuri ja mikropalveluarkkitehtuuri. Mihin näistä järjestelmää kehittäessä tulisi päätyä, riippuu siitä, millaisia tavoitteita hajautetun järjestelmän integraatiolle on asetettu ja millainen integroitava järjestelmä on. Siinä missä pipelinella saadaan aikaiseksi arvoketjuja, niin hub-and-spoke tarjoaa enemmän mukautettavaa kommunikaatiota sovellusten välillä. SOA sekä mikropalveluarkkitehtuuri tarjoavat sekä pipelinen, että hub-and-spoke-arkkitehtuurin hyviä puolia, mutta ovat välillä tarpeettoman raskaita ratkaisuja.

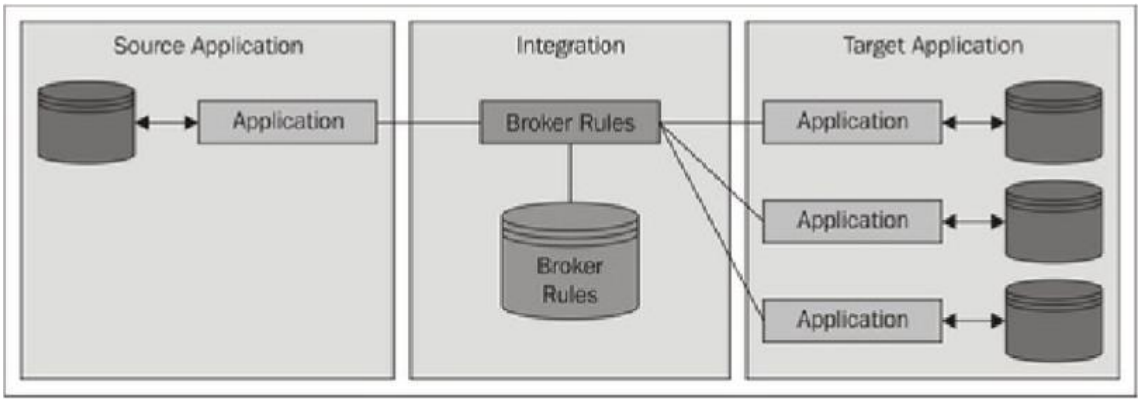
3.4 Viestinnän ja informaation integraatio

Tässä luvussa käsitellään viestinnän ja informaation integraation toteuttamista. Tämä on erityisen tärkeää mietittäessä kommunikaation ja informaation keräämistä eri sovellusten välillä. Tässä luvussa käsitellään kaksi arkkitehtuurimallia viestinnän ja informaation integraatiossa, jotka pohjautuvat suurilta osin edellä luvussa 3.3.2 käsiteltyyn hub-and-spoke-arkkitehtuuriin.

3.4.1 Broker

Broker-arkkitehtuurimalli pohjautuu suoraan yhteyteen kahden järjestelmän välillä, mutta laajentaa sen siten, että yksi tai useampi sovellus voi välittää viestejä M-määrälle sovelluksia (Kuva 17). Käytännössä siis broker-rakenteessa on selkeästi N-määrä sovelluksia, jotka viestivät brokerin logiikan kautta M-määrälle sovelluksia. Sovellukset voivat olla yhteydessä brokerin kautta usean sovelluksen kanssa, mutta sovellusten ei tarvitse itse olla tietoisia siitä kenen kanssa ne kommunikoivat vaan broker vastaa siitä. Broker-arkkitehtuuri pyrkii pitämään tarvittavat 1:1 suorat kommunikaatiot minimissä, mutta samalla selkeästi määrittelemään kommunikaatiossa lähettävän ja vastaanottavan osapuolen. [4, s. 53]

Brokerin logiikasta vastaa brokerin säännöstö, jonka pohjalta broker tulkitsee viestivirtoja ja uudelleen ohjaa niitä säännöstön mukaisesti. Yleisiä brokerin säännöstön tukemia toimintoja voivat olla esimerkiksi julkaise-, tilaa-, pyyntö- ja vastaa pohjaiset käskyt. [7, s. 129]



Kuva 17. Broker-arkkitehtuurimalli [4, s. 53].

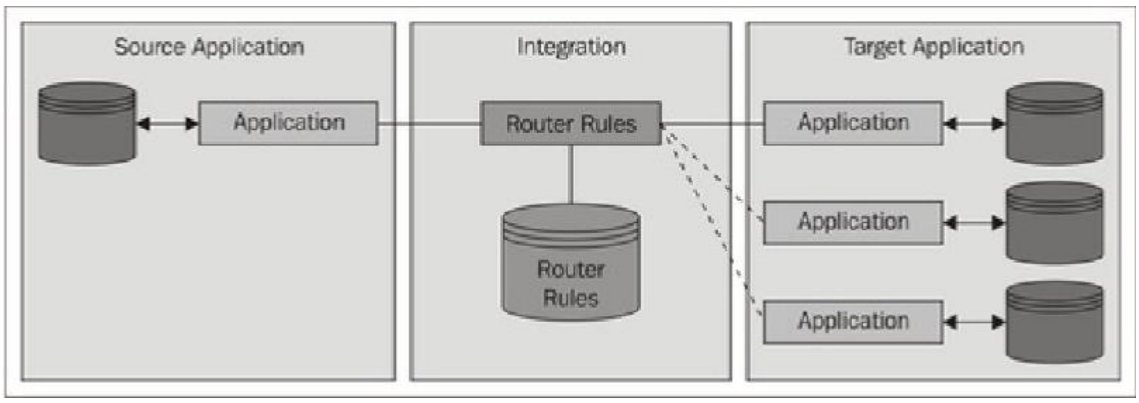
Broker-arkkitehtuuri pohjautuu pitkälti hub-and-spoke-arkkitehtuuriin. Tärkein ero on se, että broker arkkitehtuurissa on konkreettisesti määritelty lähettävät ja vastaanottavat sovellukset. Viestien reitityksestä vastaa broker itse sen sisällä olevien sääntöjen pohjalta eikä rajapinnan kautta yhteydessä oleva sovellus.

Broker-arkkitehtuurin etuna on se, että se sallii useiden sovellusten välisen kommunikaation, minimoi tarvittavien yhteyksien määrän, sallii viestinnän olla standardoitua ja kevyttä. Tästä huolimatta lähettäjä ja vastaanottaja eivät käytä samaa rajapintaa, mikä mahdollistaa erillisten protokollien käytön viestinnässä eri osapuolien osalta. Haittana on kuitenkin se, että järjestelmän logiikka ja säännöstö viestien välittämiseen tulee määritellä etukäteen brokerilla eikä se salli viestien purkamista ja uudelleen kokoamista ennen viestin välittämistä. [4, s. 55]

3.4.2 Reititin

Reititinarkkitehtuurimalli on muunnos broker-arkkitehtuurimallista. Reitittimen tärkein ero brokeriin on, että reitittimessä lähettävän sovelluksen viesti päätyy aina yhteen vastaanottavaan sovellukseen reitittimellä olevan säännöstön pohjalta. Sen sijaan brokerilta viestit kulkevat M-määrään sovelluksia (Kuva 18). [4, s. 56] Reitittimen ideana on, että reititintä ympäröivät sovellukset ovat täysin tiedottomia reitittimen olemassaolosta ja reitittimen vastuulla on vain siirtää viestejä. Reitittimen kiistaton etu

on, että logiikka viestin välityksestä on vain ja ainoastaan reitittimen sisällä, jolloin sen hallinnointi on helpompaa. [27, Luku 3]



Kuva 18. Reititinarkkitehtuuri [4, s. 55].

Reititinarkkitehtuurimallista seuraa loppujen lopuksi integraatiomalli, joka pitää sisällään yksittäisiä yhteyksiä, mutta viestit kulkevat hub-and-spoke-arkkitehtuurin kaltaisesti keskitetyn hubin kautta. Näin ollen reititinarkkitehtuurissa reitittimen kommunikaatioarkkitehtuuri yhdistetään hub-and-spoke-pohjaiseen arkkitehtuuriin. Reitittimen hyviä puolia ovat, että se sallii kommunikaation useiden sovellusten välillä, minimoi tarvittavien yhteyksien määrän ja tarjoaa sekä lähettäjille että vastaanottajille eri rajapinnat, mikä sallii eri ohjelmistojen käyttää eri viestintäprotokollia. Reititin myös vähentää rajapinnan monimutkaisuutta verrattuna hub-and-spoke-arkkitehtuuriin. Haittapuolina on, että reitittimeen pitää määritellä kommunikaation säännöstö etukäteen ja se ei salli viestien purkua ja uudelleen kokoamista ennen viestin välittämistä. [4, s. 56] [27, luku 3] Reititin ei myös välttämättä ole paras mahdollinen ratkaisu, mikäli reitityslogiikkaa joudutaan muuttamaan usein. [27, luku 3]

3.5 REST-arkkitehtuurin teoriaa

REST-arkkitehtuuri (representational state transfer) on http-protokollia käyttävä eli web-pohjainen arkkitehtuuri. REST-arkkitehtuuri luotiin aikoinaan internetin tarpeisiin, jotta verkkopohjaiset palvelut voitaisiin toteuttaa mahdollisimman joustavasti ja tehokkaasti. REST-pohjaisten palveluiden kehitys pohjautuu palvelimiin, joiden kanssa saadaan kommunikoitua yhteisen hypertekstin siirtoprotokollan (engl. hypertext transfer protocol, HTTP) avulla tilattomasti [23, s. 211]. Tiedon visualisointi onnistuu yhteisen HTML-tiedostojen avulla, joita voidaan manipuloida Cascading style sheettien (CSS) ja Javascript tai extensive markup language (XML) koodin avulla käyttäen verkkoselainta [23, s. 212].

REST-arkkitehtuuri koostuu seuraavista pääpiirteistä [11, luku 1] [23, s. 212]:

1. Käyttäjän ja palvelimen eriyttäminen front-endiin ja back-endiin.
2. Yhtenäinen rajapinta, esimerkiksi http-pohjainen kommunikaatio.
3. Resurssien eriyttäminen URI:n (Uniform resource identifier) avulla, esimerkkinä URL-osoite (Uniform resource locator).
4. Resurssien manipulointi käyttäjän käyttöliittymästä, esimerkiksi HTML-pohjainen käyttöliittymä selaimessa ja JSON-pohjainen ohjelmisto, jossa JSON-sisältää tarvittavia tietoja, joita käyttöliittymä manipuloi ja välittää palvelimelle tarvittaessa.
5. Resurssin haluttu tila voidaan esittää osana käyttäjän pyyntöä palvelimelle, esimerkiksi URL-osoitteen parametri.
6. HATEOAS (Hypermedia as the engine of application state), eli linkkipohjainen tapa siirtyä web-sovelluksessa eri ominaisuuksien välillä.
7. Kerroksinen järjestelmä eli voi sisältää erilaisia proxyja ja portteja, mutta ne toteutetaan läpinäkyvästi, jotta kommunikointi verkossa voi olla turvallista.
8. Välimuisti eli ”cache” joka sijaitsee jossain käyttäjän ja palvelimen välillä ja joka sisältää ohjeita palvelimelle käyttäjän toiminnasta sovelluksessa ja lisää siten luotettavuutta ja turvallisuutta. Välimuisti voi sisältää esimerkiksi käyttäjien kirjautumistietoja.
9. Tilattomuus, eli palvelin ei välitä eikä muista sen käyttäjiä ja niiden tilaa. Sen seurauksena käyttäjän pitää joka kerta palvelimen kanssa kommunikoidessa informoida palvelimelle parametreilla sen ominaisuuksista. Tämän ansiosta palvelin pystyy palvelemaan useampia käyttäjä mutta käyttäjien tarvitsee tallettaa ja välittää enemmän tietoa itsestään palvelimelle.
10. Koodia pyynnöstä, eli palvelin voi välittää käyttäjälle osia koodista, jotka suoritetaan tietyissä tilanteissa. Tämä voi olla esimerkiksi javascript tai flash-koodia.

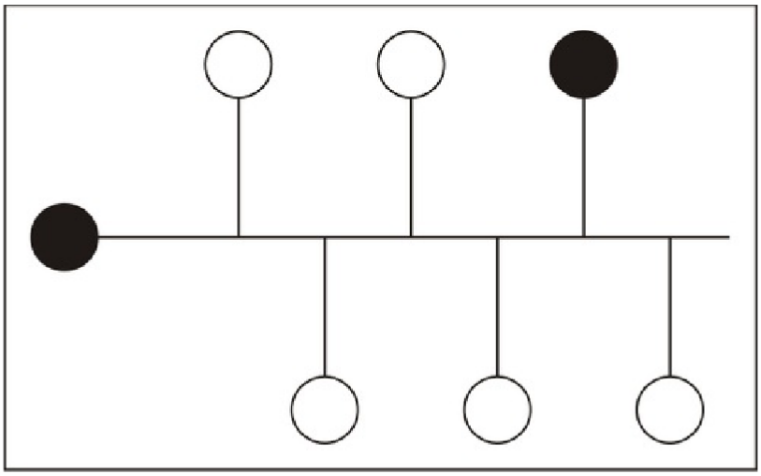
REST-arkkitehtuurin etuna on se, että se on yleisesti käytetty standardi verkkopohjaisten ohjelmien kommunikaatioon rajapinnan ylitse. REST-arkkitehtuurin suurimpina hyötyinä ovat joustavuus ja laajennettavuus, jotka ovat tärkeitä järjestelmän integraation osalta. REST-pohjaisen arkkitehtuurin etuna on myös sen helppo käytettävyys, sillä jokainen voi käyttää sitä selaimen avulla. REST-arkkitehtuuria voi myös hyödyntää osana erilaisia integraatioarkkitehtuureja ottamalla REST:in mukaisen kommunikaatiorajapinnan osaksi valittua integraatiomallia.

3.6 Kommunikaation periaatteet

Tässä luvussa käsitellään erilaisia kommunikaatiotekniikoita, joita erilaiset systeemiarkkitehtuuriset ratkaisut voivat käyttää keskenään kommunikointiin. Näitä kommunikaatiotapoja tarvitaan pohdittaessa erilaisia mahdollisia ratkaisuja hajautetun järjestelmän integraatiota varten. Luku koostuu neljästä alaluvusta, joissa käsitellään yleisimmät kommunikaatiotekniikat lyhyesti eli unicast, broadcast, multicast ja anycast sekä mihin integraation suunnittelumalleihin ne soveltuvat parhaiten.

3.6.1 Unicast

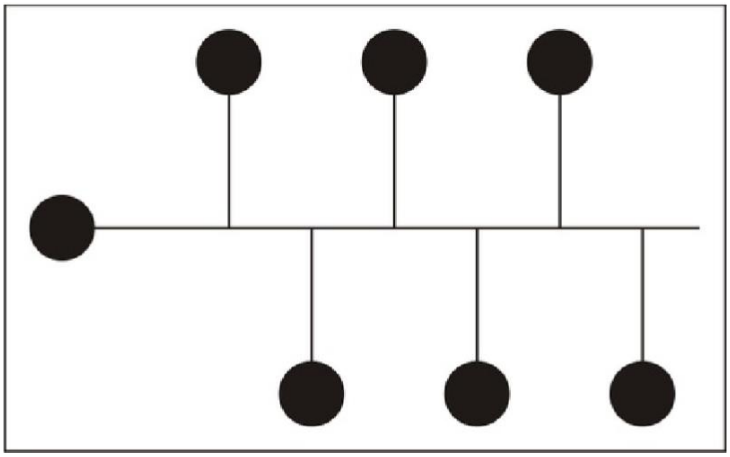
Unicast on lähetystapa, jossa dataa lähetetään yhdestä lähteestä yhteen kohteeseen [4, s. 42] [28, s. 508]. Kommunikaatio on siis täysin kahden osapuolen välistä, vaikka verkossa olisi useampia osapuolia (kuva 19) [4, s. 42]. Unicast on käytännössä ainut tapa toteuttaa point-to-point ja reititinarkkitehtuurin kommunikaatio. Point-to-point koostuu useista unicast pohjaisista kommunikaatioista eri sovellusten välillä, kun taas reitittimessä jokainen sovellus ylläpitää unicast pohjaista kommunikaatiota yllä reitittimen kanssa. Lisäksi hub-and-spoke, broker ja pipeline voivat eri tilanteissa käyttää unicast pohjaista kommunikaatiota.



Kuva 19. Unicast-kommunikaatio, jossa mustat pallot edustavat kommunikoivia osapuolia [4, s. 42].

3.6.2 Broadcast

Broadcast on lähetystapa, jossa viesti lähetetään jokaiselle verkossa olevalle osapuolelle välittämättä vastaanottavasta osapuolesta (Kuva 20) [4, s. 42] [28, s. 508]. Broadcast on hyvin tilannekohtainen ja muita kommunikaatiotekniikoita yksinkertaisempi toteuttaa. Lisäksi se voi saavuttaa saman lopputuloksen kuin unicast vaikka samalla aiheuttaa verkkoon paljon enemmän liikennettä. [4, s. 42]

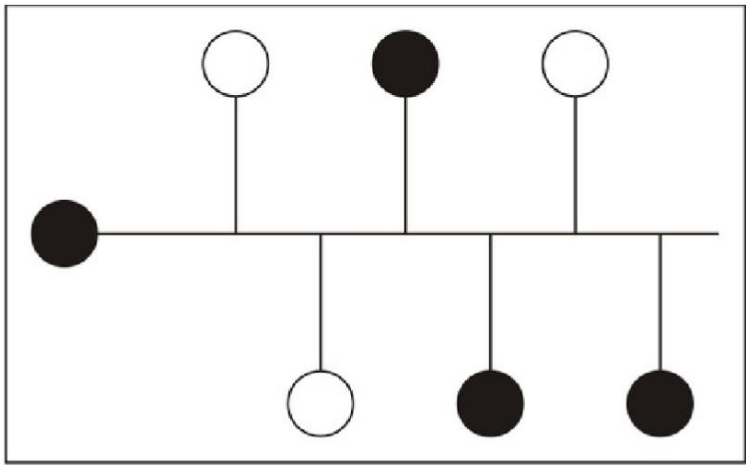


Kuva 20. Broadcast-kommunikaatio, jossa mustat pallot edustavat verkossa kommunikoivia osapuolia [4, s. 43].

Broadcast on tilannekohtainen mutta toimiva vaihtoehto joissakin tilanteissa hub-and-spoke-arkkitehtuurissa ja SOA:ssa mikäli järjestelmällä on tarve viestittää kaikille mahdollisille alijärjestelmille. Tällainen tilanne voisi olla esimerkiksi se, jossa orkestroiva järjestelmä tiedustelee alijärjestelmien tilaa.

3.6.3 Multicast

Multicast on lähetystapa, jossa viesti lähetetään useammalle kuin yhdelle verkossa olevalle osapuolelle [4, s. 43] [28, s. 508]. Kommunikaatio on siis samanlainen kuin broadcastissa 1:N mutta verkosta vain osa vastaanottaa viestin (kuva 21) [4, s. 43]. Multicast on siinä mielessä lähempänä unicastia kuin broadcastia, sillä molemmat lähetystekniikat vaativat rajauksen siitä mille kohteille viesti lähetetään.

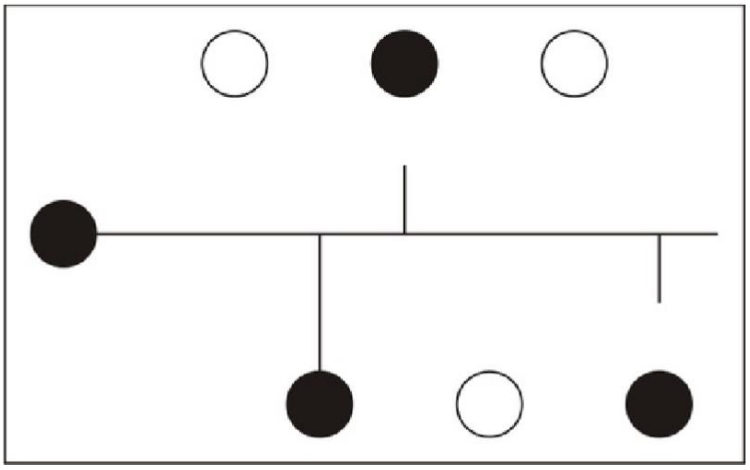


Kuva 21. Multicast-kommunikaatio, jossa mustat pallot edustavat verkossa kommunikoivia osapuolia [4, s. 43].

Multicast voi olla toimiva ratkaisu järjestelmien integraatiossa, jos sama viesti halutaan välittää usealle, eikä vain yhdelle järjestelmän osalle, mutta ei silti kaikille mahdollisille vastaanottajille. Multicast voi olla toimiva vaihtoehto käytännössä kaikissa luvussa 3.3 käsitellyille arkkitehtuurimalleille ja lisäksi 3.4.1 luvun broker-arkkitehtuurille.

3.6.4 Anycast

Anycast on lähetystapa, jossa tieto välitetään tiettyjen asetusten mukaan, kuten multicastissa, yhdelle lähimmälle ja nopeimmalle vastaanottajalle (Kuva 22) [4, s. 44] [28, s. 508]. Anycast siis on aina vain kahden osapuolen välinen viestintätapa, mutta viestinnän osapuoli määritellään laajemmin. [4, s. 44]



Kuva 22. Anycast-kommunikaatio, jossa mustat pallot edustavat verkossa mahdollisesti kommunikoivia osapuolia [4, s. 44].

Anycast voi tulla kyseeseen järjestelmissä, jossa on useita vastaavia komponentteja eikä ole tiedossa tai väliä mikä niistä viestin vastaanottaa ja prosessoi. Anycast voi olla

toimiva käytännössä kaikissa luvun 3.3 arkkitehtuurimalleissa ja 3.4.1 luvun broker-mallissa.

3.7 Tiedon tallentamisen mallit

Tietokantaa valittaessa on hyvin tärkeää tiedostaa järjestelmän vaatimukset ja tarvittavan tietokannan laajuus. On myös hyvä selvittää, onko markkinoilla tarjolla jo olemassa olevia tietokantaratkaisuita, jotka tarjoavat tarvittavat ominaisuudet. Markkinoilla olevat tietokantaratkaisut voidaan karkeasti jakaa relaatiotietokantoihin eli SQL-pohjaisiin (Structured Query Language) ratkaisuihin ja oliotietokantoihin eli NoSQL-pohjaisiin (Not Only Structured Query Language) ratkaisuihin.

SQL on yleisesti käytössä oleva kyselykieli, jolla voi muodostaa tietokantataulukkoja ja hakea niistä olevaa tietoa käyttäen avuksi erilaisia kyselyitä. Sen sijaan NoSQL-pohjaiset tietokannat poikkeavat perinteisistä SQL-tietokannoista, sillä niissä ei ole mitään perinteistä SQL-tietokantataulukkorakennetta. Seuraavissa alaluvuissa käsitellään lyhyesti hajautetun järjestelmän integraation etuja ja tiedon keruun keskittämisen kannalta SQL ja NoSQL tietokantaratkaisujen etuja ja haittoja.

3.7.1 Relaatiotietokanta

Relaatiotietokannoista puhuessa puhutaan SQL-tietokannasta eli ”structured query languagesta”. SQL on standardoitu tietokantakoodi, jossa SQL-koodilla alustetaan taulukoita, sekä muokataan niiden sarakkeiden ja rivien sisältöä käyttäen avuksi SQL:län operaattoreista tehtyä logiikkaa. Logiikan avulla taulukoiden sisältöä voi linkittää keskenään, tätä kutsutaan niin sanotusti relaatiomaiseksi tietomalliksi eli RDBMS (Relational Database Management System), joista SQL on tunnetuin [10, s. 17].

SQL:n taulukon osien linkittämisen etuna on, että melkein kaikenlaisia tietokantoja pystyy luomaan ja niiden logiikkaa ylläpitämään. Vaatimuksena on lähinnä se, että kehittäjä näkee tietokantarakenteen ja sen tarvitseman tietokantalogiikan luomisen, sen tarvitseman ajan arvoiseksi. Lisäksi SQL:n etuna on sen relaatiomaisuus. Tällä tarkoitetaan sitä, että SQL-taulukon sarakkeita ja rivejä voi linkittää toisiin taulukoihin. Näin ollen yhden taulukon sarakkeen tai rivin muokkaaminen johtaa suoraa muiden taulukoiden sisällön muuttumiseen. Toimiessaan tämä on hyvin yksinkertainen ja varma tapa ylläpitää tietokannan välisiä yhtenäisyyksiä. SQL:n niin kuin muiden RDBMS-tietokantojen ongelmana on kuitenkin niiden valmistelun ja luomisen monimutkaisuus. Ensinnäkin SQL-taulukkoon haluttavaa tietoa voi olla useissa eri formaateissa, jonka muuttaminen SQL-taulukon muotoon on vaikeaa ja työlästä eikä siltikään takaa haluttua

lopputulosta. Näin voi käydä esimerkiksi erilaisten tiedostojen kuten javascript object notation- (JSON) ja XML-tiedostojen kanssa. [10, s. 33] Suurien SQL-tietorakenteiden tehokkuus voi myös vaarantua siinä tapauksessa, jos SQL-tietokannassa on riittävän monia taulukoita ja niiden välisiä relaatioita keskenään. Tämä johtaa siihen, että yhden taulukon sarakkeen muokkaaminen voi aiheuttaa taulukoiden välillä pitkän ketjureaktion, jossa useampi taulukko muuttuu. Tämä taas johtaa tietokannan tehokkuuden laskemiseen. Näin ollen SQL-tietokannat ovat parhaimpia silloin, kun taulukosta haetaan tietoa, mutta ei silloin kun sitä muutetaan. Tämän takia tietokantaa valittaessa kannattaa miettiä, kuinka usein tietokannan sisältöä muutetaan. SQL-tietokannan kyselyt myös vaativat määrittelyä sen suhteen, minkälaisia arvoja tietokannasta tulee pihalle. Tämä voi aiheuttaa ongelmia tietokannan logiikan luonnissa tilanteissa, joissa tietokannan sisältö ei ole täysin varma. [10, s. 36] SQL-tietokanta niin kuin muutkaan RDBMS-tietokannat eivät ole välttämättä paras ratkaisu, jos suunnitellaan hajautetun sovelluksen tietokantaratkaisua etenkään silloin, jos tietokantaan on tarkoitus tallentaa valtavia määriä tietoa eli ”big dataa”. [10, s.36] “Big datan” suurimmat haasteet ovat: suuret datanopeudet, datan muodon erot, suuret datamäärät ja se, että tieto on hajautettuna useisiin sijainteihin [29, s. 2]. Relaatiotietokannat eivät ratkaise hyvin näitä haasteita johtuen tietokannan rakenteesta, joka vaikeuttaa laajentamista ja suorituskykyongelmista suuria datamääriä käsiteltäessä [29, s. 6].

Hajautetussa järjestelmässä tietokantaan tallennettua tietoa on usein useissa eri muodoissa ja sitä voidaan joutua muokkaamaan paljon. Näin ollen SQL ei ole paras mahdollinen ratkaisu. Lisäksi hajautettujen järjestelmien tietokantoihin voidaan haluta tallentaa järjestelmien toiminnan logiikkaa sekä myös integroidun järjestelmän sovellusten lähettämää lokitietoa ja muuta dataa. Tämä voi johtaa tietokannassa suureen tietokannan muokkaamisen tarpeeseen, suureen tiedonsiirron tarpeeseen ja useisiin erilaisiin tallennettaviin datatyyppeihin, mistä johtuen SQL ei ole välttämättä paras mahdollinen tietokantaratkaisu hajautetun järjestelmän tietokantaa mietittäessä.

3.7.2 Oliotietokanta

Oliotietokannoista puhuttaessa tarkoitetaan NoSQL-tietokantoja. NoSQL eli ”Not only SQL” on kehitetty edellisessä luvussa esitettyjen ongelmien takia ratkaisuksi. NoSQL pitää sisällään ei relaatiomaisia tietokantoja (Non-Relational Databases) eli niissä ei ole taulukoiden välistä linkitystä ja SQL:n kaltaista logiikkaa, jossa ohjelmistokehittäjän tulisi luoda tietokanta alusta loppuun itse. Uudemmat NoSQL-tietokannat tarjoavat joustavia ratkaisuja, esimerkiksi metodeja, joilla voi sekä luoda tietokantoja yleisen syntaksin

mukaisesti, että lisätä tietoa tietokantaan. NoSQL-tietokannoissa tieto on usein tallennettuna sanakirjamaisiin rakenteisiin, joissa valittu kuvaava avain osoittaa siihen linkitettyyn arvoon. Dataa voi tietokannasta hakea avaimen ja avaimeen linkitetyn arvon avulla. [10, s. 38] [29, s. 5]

NoSQL:n relaatiomattomuus mahdollistaa moderneissa tietokantaratkaisuissa monia asioita, joita relaatiomaiset tietokannat eivät tarjoa. Esimerkiksi suurissa data määrissä relaatiomaisessa tietokannassa taulukoiden linkittäminen voi hidastaa tietokannasta tiedon poistoa ja muokkaamista. Sen sijaan relaatiomattomissa tietokannoissa tietoa ei linkitetä keskenään, joten tätä ongelmaa ei ole. NoSQL-tietokannat myös mahdollistavat monimutkaisempia ja hajautetumpia tietokantaratkaisuja esimerkiksi useille eri palvelimille toisin kuin konventionaaliset SQL-tietokannat. Tämä mahdollistaa yhdessä nopeamman tiedon prosessoinnin avulla suurien datamäärien tallentamisen eli ”big datan”, tehokkaammin kuin konventionaaliset SQL-ratkaisut. [10, s. 38]

NoSQL:n vahvuus on myös se, että NoSQL-pohjaista tietokantaa ei kiinnosta se missä muodossa eli ”schemassa” sille annettava data on eikä se vaikuta tietokannassa tehtäviin hakuihin. Tämä helpottaa huomattavasti paitsi tietokantaan tiedon lisäämistä mutta myös sieltä tiedon hakemista [10, s. 38]. Toisaalta muodon puute voi aiheuttaa ongelmia, esimerkiksi vaikeuttaa kustomoitujen näkymien tekemistä. Tämä johtuu siitä, että ohjelmistokehittäjä ei voi olla täysin varma tietokannassa sijaitsevan tiedon muodosta. [29, s. 5] NoSQL pitää siis sisällään tietokantoja, jotka eivät talleta tietoa perinteisesti taulukoihin, käyttävät erilaista syntaksia kuin SQL ja eivät rajoita tietokantaan syötettävän tiedon muotoa ja tukevat datan hajauttamista. [10, s. 38]

Edellä mainitun teorian pohjalta voidaan sanoa, että NoSQL soveltuu monilta osin hajautetun järjestelmän integraatioon tehokkaammin. Tämä johtuu siitä että, se on paitsi paremmin laajennettava ja se ei välitä siihen syötettävän datan muodosta. NoSQL myös mahdollistaa konventionaalista SQL-rakennetta nopeamman ja tehokkaamman tavan muokata tietokannan sisältöä. Lisäksi NoSQL:n metodipohjainen tiedon haku ja syöttäminen mahdollistaa selkeän ja helpon tavan paitsi lisätä tietoa niin myös luoda tapoja hakea ja muokata sitä.

4. SANDVIKIN TESTIYMPÄRISTÖ JA SEN KEHITYSTARPEET

Tässä luvussa käsitellään johdannossa kuvaillun Sandvikin testausympäristön toiminta nykyään sekä ohjelmistokehityksen, että integraatio- ja järjestelmätestauksen näkökulmasta. Luvussa kerrotaan myös Sandvikin hajautetun testausympäristön kehitystarpeista ja integraatiolle asetetut vaatimukset ja tavoitteet, joihin pyritään vastaamaan työn käytännön toteutuksen aikana.

4.1 Ohjelmistokehitys ja testaus nykyisellään

Tässä alaluvussa käsitellään tarkemmin ohjelmistokehitystä ja testausta Sandvikilla aiemmin käsitellyn teorian pohjalta. Ensimmäisessä alaluvussa kuvaillaan lyhyesti ohjelmistokehitystä Sandvikilla testauksen näkökulmasta. Toisessa alaluvussa kuvaillaan integraatio- ja järjestelmätestausta Sandvikilla.

4.1.1 Ohjelmistokehitys Sandvikilla

Sandvikilla käytetään luvussa 2.3.1 kuvattua jatkuvan integraationpalvelua Jenkinsiä. Sandvikilla Jenkinsiä käytetään, kuten monissa muissa ohjelmistokehitysprojektissa, ensin yhdistämään eri ohjelmistokomponentit säännöllisesti yleensä yöllä yhdeksi käännöspaketiksi. Tämä käännöspaketti vastaa tietyn kaivoslaitteen ohjelmistoa ja sille ajetaan yksikkötestejä osana Jenkins CI-putkea heti käännöspaketin valmistumisen jälkeen kuten voi nähdä kuvasta 23.

Kuva 23. Jenkins-pipeline esimerkki.

Nykyisin, kun testaaja tulee aamulla töihin, hän joutuu tarkastamaan ensimmäisenä Jenkinsin käännöspaketin tilan ja, että versio on mennyt yksikkötestauksesta lävitse. Tämän jälkeen testaaja voi asentaa uuden käännöspaketin simulaattorin ohjelmistoon ja vasta tämän jälkeen hän voi suorittaa simulaattoritestauksen. Tämän takia tässä työssä pyritään integroimaan Jenkins ja integraatio- ja järjestelmätestaus osaksi yhteistä prosessia.

4.1.2 Integraatio- ja järjestelmätestaus Sandvikilla

Sandvikilla integraatio- ja järjestelmätestauksessa käytetään luvussa 2.3.2 kuvattua Creanex-simulaattoria. Simulaattoria voidaan käyttää joko manuaalisessa testauksessa tai osana automatisoitua integraatio- ja järjestelmätestausta.

Sandvikilla automatisoitu integraatio- ja järjestelmätestaus Creanex-simulaattorilla toimii siten, että python-pohjaiset testiskriptit ajetaan joko Creanex PC:ltä käsin tai yleisemmin testaajan omalta tietokoneelta. Testiautomaatio kommunikoi simulaattorin ajotietokoneiden kanssa CAN-signaalirajapinnan kautta, lisäksi ajotietokoneiden kanssa kommunikoidaan secure shell (SSH)-yhteyden ylitse käyttäen Open SSH palvelinta. Tämän lisäksi testiautomaatio kommunikoi http-protokollalla Creanex SITA Gateway palvelimen välityksellä. Tämä rakenne mahdollistaa sen, että testaaja voi tehdä töitä hänelle sopivammasta sijainnista, joko simulaattorin vieressä tai etänä http- ja ssh-protokollan avulla. Creanex-simulaattorin tilaa pystyy myös seuraamaan etänä yhdistämällä Creanex-PC:n ruutuun virtual networking computing (VNC)-protokollalla. Tämä koko rakenne on nähtävissä kuvassa 1. Python-pohjainen testiautomaatio myös luo ja kirjoittaa testitapauksen etenemisestä lokitietoa erilliseen html-tiedostoon testiä ajavalle tietokoneelle. Testitapauksen epäonnistuessa tätä html-tiedostoa tarkkailemalla pystytään löytämään se käyttötapaus,

joka aiheuttaa virheen testattavan järjestelmän toiminnassa, tai löytämään kohta, joka pitää muuttaa testaavassa testiskriptissä. Näin nähdään myös ulkopuoliset virhetilanteet, kuten, se jos johonkin tietokoneyksikköön ei ole saatu yhteyttä testin aikana. Tämä helpottaa huomattavasti sekä ohjelmiston testaamista ja testiautomaation ylläpitoa.

4.2 Nykyisen arkkitehtuurin kehitystarpeet

Ohjelmistokehittäjien ja testaajien ympäristöjen erottuminen toisistaan kuvan 1 mukaisesti aiheuttaa ongelmia paitsi manuaalitestauksessa, myös automaatiotestauksessa. Esimerkiksi testattavan ohjelmistoversion vaihtuessa se pitää testata uudelleen. Näin voidaan varmistaa, että uusi versio ei ole tuonut mukanaan virheitä sekä vanhoihin, että juuri toteutettuihin testattavan järjestelmän toimintoihin. Jatkuvan integraation puuttumisen takia tämä kuluttaa tarpeettomasti testaajien työaikaa, koska osana jatkuvaa integraatiota testattavan ohjelmiston voisi asentaa ja testata osittain automaattisesti. Pelkästään testattavan ohjelmiston asentamisessa voi hyvin kulua aikaa 10–15 minuuttia. Nykyinen rakenne ei myöskään mahdollista automatisoitua säännöllistä automaatiotestausta osana jatkuvaa integraatiota esimerkiksi öisin. Tämän takia testaajan pitää käynnistää automaatiotestit manuaalisesti. Testien ajaminen testattavasta järjestelmästä riippuen voi viedä aikaa jopa useita tunteja. Tämä tarkoittaa sitä, että testaajien työpäivästä testiympäristö voi olla tunteja varattuna testiautomaation ajamisen takia. Näin ollen automaatiotestien ajaminen on hyvin epäsäännöllistä ja täysin testaajan omasta ajankäytöstä kiinni. Esimerkiksi testaaja voi laittaa testiautomaation päälle työpäivän päättyessä tai lähtiessään ruokatauolle. Säännöllisen testirutiinin puuttuminen vaikeuttaa testiautomaation ylläpitoa, mutta myös laitteen ohjelmistoversion seurantaa.

Tämän lisäksi nykyisellään automaatiotestien html-pohjaiset testitulokset jäävät vain ja ainoastaan tietokoneelle, jolla python-pohjaiset automaatiotestit ajetaan. Tämä on ongelma sillä testitulokset ovat tärkeitä, paitsi itse testiautomaation ajajille, niin myös testiautomaation ylläpitäjille ja kehittäjille. Näin ollen testitulokset eivät olekaan usein kaikkien saatavilla helposti, vaan niitä pitää kysellä eri laitteiden testaajilta tai vaihtoehtoisesti ajaa samat testit uudelleen. Uudelleen ajo ei kuitenkaan aina ole ratkaisu, sillä samojen tuloksien saaminen ei aina ole yksinkertaista. Tämä johtuu siitä, että simulaattoreiden laitteistoissa ja konfiguraatioissa on eroja, testiautomaation ylläpitäjän simulaattorilla projektia ei välttämättä saa vaihtaa, simulaattoreita on rajallinen määrä, eikä samalla simulaattorilla testaaminen aina ole mahdollista. Tämä vaikeuttaa edelleen eri laitteiden ohjelmistoversioiden ja testiautomaation toimivuuden seuraamista ja mahdollista kehitystyötä. Lisäksi ohjelmistovirheen löytyessä testituloksia voidaan

joutua siirtämään manuaalisesti henkilöltä toiselle, jotta ohjelmistovirheen aiheuttajan löytyminen ja korjaaminen helpottuisi.

Nykyisen järjestelmän kehitystarpeisiin kuuluu paitsi jatkuvan integraatio ympäristö Jenkinsin ja testaajien Creanex-simulaattoreiden integroiminen toisiinsa osaksi CI-putkea, joka mahdollistaisi säännöllisen automaatiotestauksen. Tämä putki sisältäisi uuden ohjelmistopaketin version kunnon tarkistamisen, asentamisen oikeaan simulaattoriympäristöön ja automaatiotestien ajamisen. Kyseisen hajautetun järjestelmän integrointia suunniteltaessa tulee pyrkiä luvun 3.1.1 mukaisesti mahdollisimman helppoon laajennettavuuteen ja ylläpitoon, jotta järjestelmän ylläpito ei monimutkaistuisi liikaa Creanex-simulaattoreiden ja testattavien järjestelmien määrän kasvaessa.

Kehitystarpeisiin kuuluu myös mahdollistaa mahdollisimman helppo tapa tallentaa testitulokset sijaintiin, joka olisi kaikkien testaajien saavutettavissa testaamisen seuraamisen mahdollistamiseksi. Keskitetty sijainti mahdollistaa sen, että testien tuloksia voidaan seurata useiden eri projektien testaajien kesken. Tämä auttaa sillä osa testeistä on identtisiä eri projekteissa ja pitäisi näin tuottaa identtisen testituloslokin. Käytännössä siis järjestelmäintegraatiota suunniteltaessa tulisi suunnitella mahdollinen tietokantaratkaisu, tietokannan rajapinta ja mahdolliset työkalut, joilla tietokantaan pystyy tallentamaan testituloksia automaatiotestauksessa ja hakemaan testituloksia tietokannasta.

4.3 Integroidun järjestelmän vaatimukset ja tavoitteet

Toteutettavan järjestelmäintegraation vaatimukset ja tavoitteet voidaan jakaa kahteen osaan. Ensimmäinen osa koskee itse CI-putken hallintaa ja sitä kuinka eri projektien simulaattoreita voidaan käynnistää projektien testaamiseksi. Toinen osa taas koskee sitä, kuinka testitulokset voidaan integroinnin myötä tallentaa sijaintiin, joka on helposti testaajien ja muiden kehittäjien saatavilla. Alla olevassa taulukossa 2 hajautetun järjestelmän integroinnin vaatimukset ja tavoitteet on esitetty tärkeysjärjestyksessä.

Taulukko 2. Vaatimuksien ja tavoitteiden priorisointi.

Pakollinen	**Tärkeä**	**Toivottava**
Jenkins käännöspaketin integroiminen osaksi testiautomaatiojärjestelmää	Laitteen simulaattoriohjelmiston vaihtaminen osana CI-putkea	Tietokannan varmuuskopion palauttaminen
Jenkins käännöspaketin integroinnin poistaminen järjestelmästä	CI-putken tilan seuranta	Projektien testituloksien vertailu
Jenkins käännöspaketin tilan tarkastelu	Uusimpien tulosten vertailu	Testitulosten vertailu suhteessa aikaan
Jenkins käännöspaketin automaattinen asentaminen testattavaan järjestelmään	Vianseuranta	Tietokannan säännöllinen varmuuskopiointi
Jatkuva integraatio- ja järjestelmätestaus	Testituloksien hakeminen	Testitulokset sähköpostiin
CI-putken asetuksien määrittely	Useiden projektien testaaminen samalla simulaattorilla	Ajetun testitapauksen koodin tarkastelu
Testituloksien tallentaminen keskitetysti	24/7 automaatiotestaus	CI-putken ”stop”-nappi
Testituloksien poistaminen		
Testituloksien tarkastelu		

Kuten taulukosta voi nähdä, pakollisien ominaisuuksien määrä on suurin eli yhdeksän kappaletta. Tärkeitä ominaisuuksia on seitsemän kappaletta samoin kuin toivottavia ominaisuuksia. Nämä vaatimukset ja tavoitteet ovat tärkeässä roolissa, kun seuraavassa luvuissa 5.1 käydään lävitse kolme mahdollista integraatioarkkitehtuuria ja arvioidaan niiden toimivuutta paitsi tutkimuskysymyksien näkökulmasta.

5. KÄYTÄNNÖN TOTEUTUS

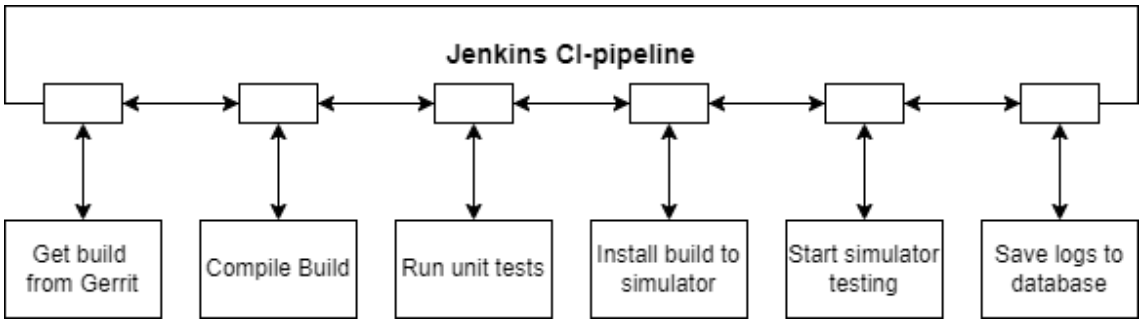
Tässä luvussa mietitään eri vaihtoehtoja Sandvikin testiympäristön integraatioon ja kuvaillaan toteutettu reititinarkkitehtuuriratkaisu käytännössä. Lisäksi kerrotaan, kuinka toteutettuun ratkaisuun päädyttiin. Luku kostuu kolmesta alaluvusta, joista ensimmäisessä vertaillaan kolmea vaihtoehtoista arkkitehtuuriratkaisua. Toisessa perustellaan, miksi reititinarkkitehtuuriin päädyttiin. Kolmannessa alaluvussa käsitellään, miten hajautetun järjestelmän integraatio toteutettiin hub-and-spoke-pohjaisella reititinarkkitehtuurilla, mitä haasteita arkkitehtuurin toteutuksessa oli ja miten järjestelmäintegraation kannalta olennainen tietokanta toteutettiin käytännössä.

5.1 Käytännön arkkitehtuuriesimerkit

Tässä luvussa käsitellään esimerkkinä kolmea eri tapaa, joilla johdannossa ja luvussa 4 esiteltyä Sandvikin simulaattoritestausympäristö voidaan integroida. Luvun käytännön esimerkit pohjautuvat luvun kolme teoriaan. Integraatioratkaisuita verrataan sen pohjalta, kuinka hyvin ne vastaavat luvussa 4 esiteltyihin nykyisen järjestelmän kehitystarpeisiin sekä integroidun järjestelmän vaatimuksiin ja tavoitteisiin. Luvun ensimmäisessä-, toisessa- ja kolmannessa alaluvussa käsitellään UML-kaavioiden avulla, kuinka pipeline-, reititin- ja SOA-arkkitehtuuri voitaisiin käytännössä toteuttaa Sandvikin testausympäristössä. Näihin arkkitehtuureihin perehdytään tarkemmin, sillä ne ratkaisisivat luvun 3 teorian pohjalta parhainten luvussa 4 käsitellyt Sandvikin testiympäristön kehitystarpeet.

5.1.1 Pipeline käytännön arkkitehtuuri

Jos luvun 3.3.3 pipeline-arkkitehtuuria käytettäisiin Sandvikilla hajautetun testiympäristön integrointiin, helpoin tapa olisi jatkaa jo olemassa olevaa Jenkins-pipelinea lisäämällä pipelineen kolme uutta vaihetta. Ensimmäisessä vaiheessa käännöspaketti asennettaisiin halutulle simulaattorille, toisessa vaiheessa automatisoidut integraatiotestit ja järjestelmätestit käynnistettäisiin halutulla simulaattorilla ja viimeisessä vaiheessa testitulokset raportoitaisiin johonkin tietokantaan. Pipeline-arkkitehtuurin tällaisesta tilanteesta voi nähdä alla olevasta kuvasta 24.



Kuva 24. Pipeline-arkkitehtuuri suunnitelma testiympäristössä.

Kyseisessä arkkitehtuurissa on kaksi suurinta ongelmaa, joista ensimmäinen koskee sitä, kuinka toteuttaa helppo laajennettavuus kyseisessä järjestelmässä. Ongelmana on se, että jokaiselle laitteelle Jenkinsissä on vähintään yksi pipeline, joista jokaisen pitäisi tietää missä internet protokollan (IP)-osoitteessa haluttu simulaattori on ja millä käskyillä testiautomaatio kyseisellä laitteella käynnistetään. Näin ollen jokaisen halutun pipelinen loppuun pitäisi joko koodata simulaattori ja asetukset, joilla automaatiotestit halutaan ajaa tai luoda jokin täysin erillinen logiikka, joilla Jenkins tietäisi millä simulaattorilla testiautomaatio halutaan ajaa. Yksinkertaisimmillaan nämä asetukset voisivat olla esimerkiksi JSON-tiedostossa. Tämän perusteella Jenkins voisi määritellä asetukset, joilla testiautomaatio käynnistetään halutulla simulaattorilla, jolloin asetuksia pystyttäisiin muuttamaan keskitetysti. Tähän tarvittaisiin kuitenkin edelleen logiikka joko Jenkinsiin tai erillinen sovellus. Sovellus tämän JSON-tiedoston pohjalta määrittelisi, millä komennoilla simulaattoriin yhdistettäisiin, käännöspaketti asennettaisiin ja testiautomaatio käynnistettäisiin. Järjestelmän muokattavuus Jenkinsin kautta ei myös olisi yhtä yksinkertaista. Toki olisi mahdollista, että jotain Jenkinsin olemassa olevaa liitännäistä voisi käyttää tällaisen käyttöliittymän luomiseen.

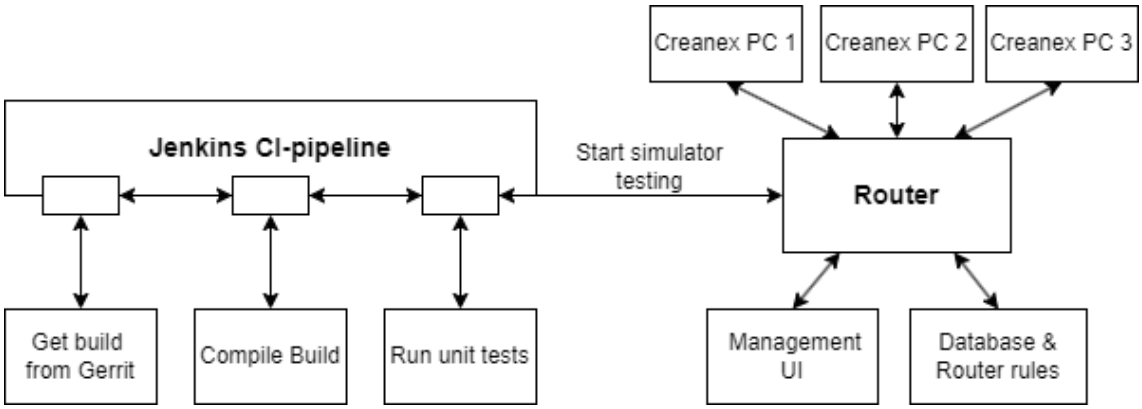
Kuitenkin vaikka tämä osa ongelmasta ratkeaisi, ongelmaksi voisi muodostua rajapinta Jenkinsin ja Creanex-simulaattorin välillä. Nykyinen Sandvikin testiautomaatio on pääosin alusta loppuun Sandvikilla varta vasten suunniteltua, jonka takia olemassa olevaa toimivaa rajapintaa Jenkinsin ja testiautomaation välillä ei ole vaan testiautomaatio ajetaan nykyisellään HTTP ja SSH-yhteyden yli kuten kuvataan tarkemmin luvussa 4.1.2. Ensimmäinen vaihtoehto olisi, että Jenkins ajaisi testiautomaatiota simulaattoreilta SSH:n ylitse, mikä voisi aiheuttaa tietoturvaongelmia ja ei olisi luotettava järjestelmä. Tämä voisi aiheuttaa helposti poikkeustilanteita pipelinessa esimerkiksi yhteysongelmien takia. Lisäksi tämä ei edelleenkään ratkaisisi kommunikaatio-ongelmaa rajapinnan ylitse, esimerkiksi kun testiautomaatio haluaisi raportoida tuloksista Jenkinsin suuntaan. Toinen vaihtoehto olisi, että jokainen Creanex-simulaattori rekisteröitäisiin Jenkinsin orjaksi, jotta Jenkinsillä olisi täysi oikeus hallita sitä. Tämä olisi tietoturvan kannalta parempi ratkaisu.

Toinen ongelma liittyy siihen, kuinka tietokanta ja sen hallinta ratkaistaan, eli lähetettäisiinkö testitulokset suoraan simulaattorilta tietokantaan vai lähettäisikö Jenkins ne. Lisäksi tarvittaisiin jokin täysin erillinen sovellus, jolla tietokannasta voitaisiin hakea ja poistaa tietoa. Toisin sanoen tarvittaisiin toinen täysin Jenkinsistä erillinen sovellus tietokannan hallintaa ja html-pohjaisten testitulosten selaamista varten. Toinen vaihtoehto olisi toki tallentaa testitulokset suoraan Jenkinsiin. Tässä ongelmaksi muodostuu kuitenkin se, että Sandvikin testiautomaatiojärjestelmä on kehitetty Sandvikin tarpeisiin eikä siihen välttämättä löydy sopivaa rajapintaa esimerkiksi html-testilokien tallentamista varten.

Kokonaisuutena pipeline-rakenne vaatisi siis ainakin Jenkinsin käännöspakettien muokkaamista. Tämä voisi tarkoittaa joko logiikan Jenkinsiin tai erillisen sovelluksen, jossa olisi erilliset tietokannat testituloksille, sovelluksen hallinnalle ja tietokannan testitulosten tarkastelulle. Sovelluksia siis tulisi olemaan 2–4 kappaletta. Rakenteesta tuskin tulisi helposti laajennettavaa vaan Jenkinsissä olisi useita erillisiä käännöspaketteja, joiden asennuksia tulisi muuttaa Jenkinsin käännöspaketteja muokkaamalla erikseen. Myös rajapintaongelman ratkaisu vaikeuttaisi tilannetta entisestään, virhehallinnasta tulisi monimutkaista ja pipelinejen luotettavuus laskisi. Voidaan sanoa, että huomattavasti parempi ratkaisu olisi pitää nykyinen järjestelmä ja siihen lisättävät osat ja ominaisuudet mahdollisimman erillään toisistaan, jotta järjestelmien luotettavuus erikseen ja yhdessä ei vaarantuisi.

5.1.2 Reitittimen käytännön arkkitehtuuri

Jos hub-and-spoke-arkkitehtuuriin pohjautuvaa reititinarkkitehtuuria päädyttäisiin käyttämään Sandvikilla testijärjestelmien integroimisessa toisiinsa lukujen 3.3.2 ja 3.4.2 pohjalta. Looginen ratkaisu olisi luoda täysin erillinen reititinpohjainen järjestelmä. Tämä välittäisi viestit Jenkins CI-pipelinen ja Creanex-simulaattoreiden välillä ja se integroitaisiin Jenkins CI-pipelinen loppuun. Kyseisen arkkitehtuurin esimerkin voi nähdä kuvasta 25.



Kuva 25. Reititinarkkitehtuuri suunnitelma testiympäristössä.

Kyseisen arkkitehtuurin suurin haittapuoli olisi työn määrä, joka vaaditaan reitittimen tekemiseen. Lisäksi rajapinta ja sen käyttämät protokollat Jenkins CI-pipelinen ja reitittimen välillä pitäisi miettiä ja perustella. Myös kysymys siitä, tallennetaanko testitulokset simulaattoreilta käsin vai tallennetaanko ne keskitetysti reitittimeltä käsin suoraan tietokantaan, on tärkeä kysymys. Ylipäänsä se, onko tietokannan hallintaan käytettävät metodit osana simulaattoreiden testiautomaatiota vai reititinarkkitehtuurin sisällä on mietittävä ja mitä hyötyjä ja haittoja näissä ratkaisuvaihtoehdoissa on. Reitittimen sisällä ollessa rajapinnan hallinta on helpompaa, mutta toisaalta sen luominen on alussa työläämpää, koska metodit ja rajapinta pitää luoda reitittimeen. Toisaalta, jos tietokantaan kirjoitetaan testitulokset suoraan simulaattoreilta, eikä reitittimen kautta, järjestelmän saa alussa nopeammin toimimaan mutta sen ylläpito ja laajennettavuus on vaikeampaa. Tämä johtuu siitä, että testitulosten tallentamiseen tarkoitettu logiikka löytyisi jokaiselta testejä ajavalta tietokoneelta erikseen. Reititin-arkkitehtuuria suunnitellessa pitäisi myös valita, onko kyseessä ”puhdas” reititin, joka sisältää vain metodeja viestien välitykseen, koskematta itse viestin rakenteeseen. Vai onko reitittimessä hub-and-spoke-arkkitehtuurin piirteitä kuten metodeja, jotka muokkaavat viestin rakennetta. Joka tapauksessa reitittimen reititintaulun kaltainen ratkaisu olisi hyvä ratkaisu, koska se helpottaisi simulaattoreiden integroimista järjestelmään. Toisaalta reititintauluun pohjautuva integraatio voi olla huono ratkaisu, jos IP-osoitteet muuttuvat usein.

Hyötyinä järjestelmässä on kuitenkin se, että reititin olisi pääosin Jenkinsistä täysin erillinen ratkaisu, jonka voi räätälöidä Sandvikin testiautomaation tarpeiden mukaan. Reititinpohjaista järjestelmää voi myös tulevaisuudessa helposti laajentaa muuttamalla reititystaulua. Mikäli reitittimen rajapinnan metodit on hyvin suunniteltu, reititintä voi laajentaa uusilla sovelluksilla myös tulevaisuudessa tarpeen mukaan. Samalla SSH-yhteyden katkeamisen kaltainen uhka Jenkins CI-pipelinen ja simulaattoreiden välillä ei olisi yhtä suuri, sillä reititin voisi sijaita samassa verkossa Creanex-simulaattoreiden

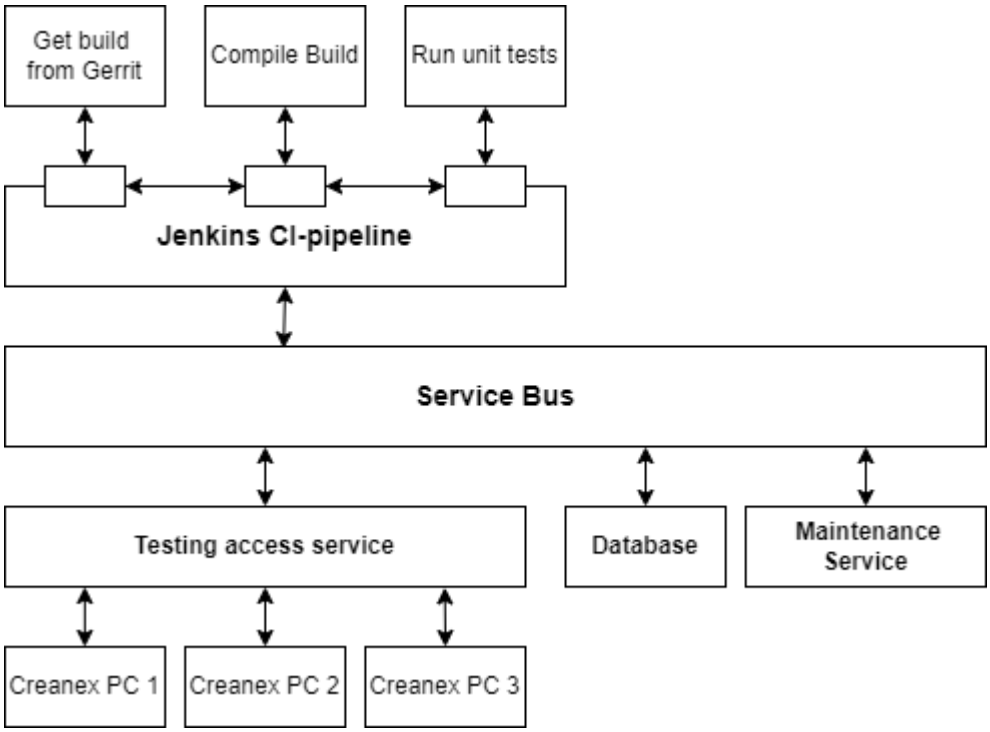
kanssa ja ottaa SSH-yhteyden eri simulaattoreihin. Tämä johtuisi siitä, että sijoittamalla reitittimen simulaattoreiden kanssa samaan LAN-verkkoon yhteyden katkeaminen on huomattavasti pienempi riski SSH:n osalta. Näin ollen Jenkins voisi säilyä pääosin erillisenä ratkaisuna ohjelmistokehittäjille ja yksikkötestaukselle, kun taas reititin vastaisi testaajien tarpeista integraatio- ja järjestelmätestauksen osalta. Lisäksi myös yhteyden yhteyden muodostaminen olisi Jenkinsin ja reitittimen välillä helpompaa, sillä vain reitittimen voisi rekisteröidä Jenkinsin orjaksi sen sijaan että rekisteröisi jokaisen Creanex-simulaattorin erikseen.

Hyötynä on myös, että reitittimen sisään voisi luoda rajapinnan tietokannan selaamista ja tietokannan tiedon prosessointia varten. Kyseinen rajapinta voisi olla esimerkiksi palvelin, joka vastaisi jonkin tietyn protokollan http:n mukaisesti sille esitettyihin kyselyihin. Kyselyt voisi joko tehdä selaimelta eli ratkaisu olisi perinteinen selaimella avattava web-pohjainen palvelu tai erillinen omalla koneella toimiva vapaamuotoinen, joka vain tekisi reitittimen rajapinnan ylitse kyselyitä tietokantaan.

Reitittimen suurin etu pipelineen verrattuna on helppo laajennettavuus, vapaamuotoinen muokkaaminen ja laajentaminen, varmempi toimivuus ja mahdollisuus yhteiseen rajapintaan simulaattoreiden ja Jenkinsin välillä. Suurin haitta sen sijaan on suurempi vaadittava työmäärä järjestelmän luomisen osalta.

5.1.3 Palvelupohjainen arkkitehtuuri käytännössä

Luvun 3.3.4 pohjalta jos palvelupohjaista arkkitehtuuria päädyttäisiin käyttämään Sandvikin testiympäristön integroimiseen, simulaattoreille tulisi luoda yhteinen rajapinta eli kuvassa 26 oleva ”testing access service”. Tämän ylitse simulaattorit olisivat yhteydessä ”service bus” -väylään, johon integroitaisiin tietokanta, Jenkins ja jokin sovellus, jolla järjestelmän asetuksia voitaisiin muokata. Kyseinen järjestelmä toimisi siten, että Jenkins lähettäisi läpi menneen käännöspaketin tiedot ”service bus” -väylään protokollalla jonka ”testing access service” tunnistaisi. ”Testing access service” varmistaisi tietokannasta, että kyseinen käännöspaketti on integroitu järjestelmään, hakisi sen asetukset ja välittäisi Jenkinsin kutsun simulaattorin rajapintaan. Simulaattori käynnistäisi asetuksien mukaisesti automaatiotestauksen ja lähettäisi tuloksia rajapinnan ylitse tietokantaan. Lisäksi ”maintenance service” olisi joko http-pohjainen rajapinta, jonka kautta voitaisiin hakea paitsi tietoa tietokannasta ja muokata sitä mutta myös muokata siellä olevia simulaattoritestauksen asetuksia.



Kuva 26. Palvelupohjaisen arkkitehtuurin suunnitelma testiympäristössä.

Kyseisen arkkitehtuurin suurin haitta olisi se, että sen luominen vaatisi paljon resursseja ja aikaa. Etenkin eri rajapintojen luominen vaatisi oman aikansa, koska pitäisi miettiä ”testing access servicen” tarvitsemaa logiikkaa ja ”maintenance servicea” käyttävää sovellusta, jolla järjestelmää voisi muokata ja laajentaa. SOA olisi vieläkin monimutkaisempi luoda kuin esimerkiksi edellä käyty reititinarkkitehtuuri.

Toisaalta etuna olisi se, että järjestelmän luomisen jälkeen testausympäristössä jokainen osa pääsisi helposti käsiksi SOA:n sisällä oleviin eri palveluihin yhteisten rajapintojen ylitse. Lisäksi järjestelmää olisi tulevaisuudessa helppoa laajentaa kytkemällä ”service busiin” uusia sovelluksia ja rajapintoja. Myös arkkitehtuurin muokkaaminen SOA:n puitteissa olisi huomattavasti helpompaa ja yhden sovelluksen kaatuminen ei välttämättä vaarantaisi koko järjestelmää, sillä palvelut ovat SOA:ssa toisistaan täysin erillisiä.

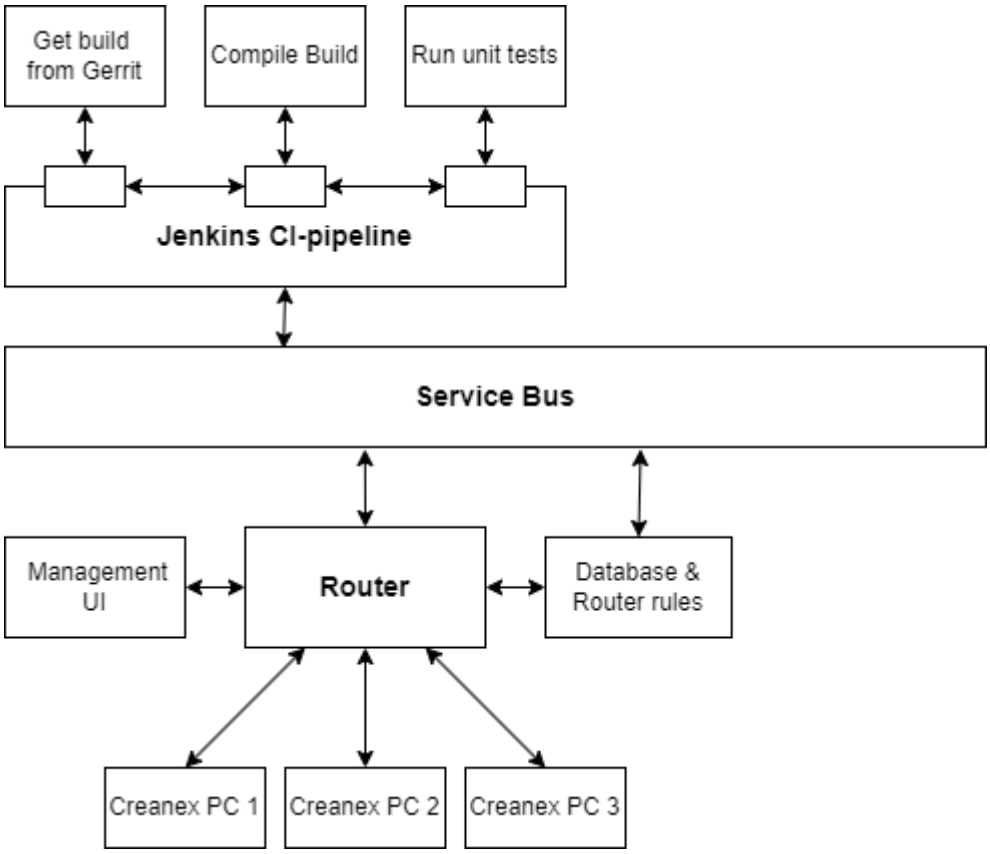
SOA:lla saavutettaisiin käytännössä kaikki, mitä reititinarkkitehtuurilla mutta vielä enemmän etenkin, jos tulevaisuudessa tulisi testaukseen liittyviä uusia tarpeita, koska ne voitaisiin lisätä helposti osaksi SOA:n ”service busia”. Toisaalta SOA-pohjainen järjestelmä olisi vielä monimutkaisempi ja sen myötä työläämpi rakentaa.

5.2 Reititinarkkitehtuurin valinnasta

Tämän diplomityön käytännön tutkimusvaiheessa päädyttiin toteuttamaan Sandvikilla hajautetun testiympäristön integraatio hub-and-spoke-pohjaisella reititinarkkitehtuurilla, pipeline- ja SOA-arkkitehtuurin sijaan. Tähän päädyttiin, koska hub-and-spoke

pohjaisella reitittimellä on teorian lukujen 3.3.2, 3.4.2 ja 3.3.6 pohjalta kyky mahdollistaa luvussa 4.3 käsitellyt testiympäristön integraation vaatimukset ja tavoitteet. Lisäksi se on myös teorian pohjalta helposti laajennettava ja muokattavissa sekä mahdollistaa keskitetyn tietokantaratkaisun. Näin ollen sillä on hyvät mahdollisuudet kyetä vastaamaan mahdollisimman hyvin Sandvikin järjestelmän vaatimuksiin. Sen sijaan esimerkiksi pipeline ei olisi tähän yhtä hyvin kyennyt, etenkään laajennettavuuden osalta kuten luvussa 5.1.1 mainitaan. Tämä johtuu paitsi Sandvikin testijärjestelmän ominaispiirteistä mutta myös Jenkinsin käännöspakettien muokkaamisen aiheuttamista rajapintaongelmista. Sen sijaan SOA:ta ei päädytty toteuttamaan, sillä se olisi tarjonnut melkein kaikki samat hyvät ominaisuudet kuin reititinarkkitehtuuri, mutta SOA olisi ollut huomattavasti työläämpi toteuttaa. SOA:n kaikki hyödyt suhteessa reititinarkkitehtuuriin eivät myöskään realisoituisi diplomityön vaatimusten puitteissa sillä riski siitä, että reititin aiheutuisi pullonkaulaksi testiympäristön laajuudessa arvioitiin vähäiseksi. Tähän lopputulokseen tultiin sillä perusteella, että reititin toteutettiin python-ohjelmointikielen Cherrypy-palvelimella, joka pystyy käsittelemään jopa 4000 pyyntöä sekunnissa [13].

Lisäksi on hyvä havaita, että reititinpohjainen arkkitehtuuri, joka toimii luvussa 3.5 käsiteltävän representational state transfer (REST)-arkkitehtuurin rajapinnan mukaisesti eli http-pyyntöjen pohjalta, voidaan tulevaisuudessa lisätä osaksi SOA-pohjaista arkkitehtuuria. Tämä johtuu siitä, että SOA:n yksi ratkaisuvaihtoehto on REST-pohjainen kuten mainitaan luvussa 3.3.4. Reitittimen muuttaminen osaksi SOA-pohjaista arkkitehtuuria toimisi siten, että reititin olisi käytännössä kuvassa 26 esitellyn SOA:n ”testing Access Service”. Tämä voidaan nähdä alla olevassa kuvassa 27.



Kuva 27. Reitittimen ja SOA:n yhdistämisen esimerkki.

Kuvasta 27 voidaan todeta, että mikäli SOA:n kaltaista arkkitehtuuria tarvittaisiin tulevaisuudessa, reititinarkkitehtuuri ei sulje sitä pois, vaan reititinarkkitehtuuri voisi olla vain yksi sovellus osana SOA:ta.

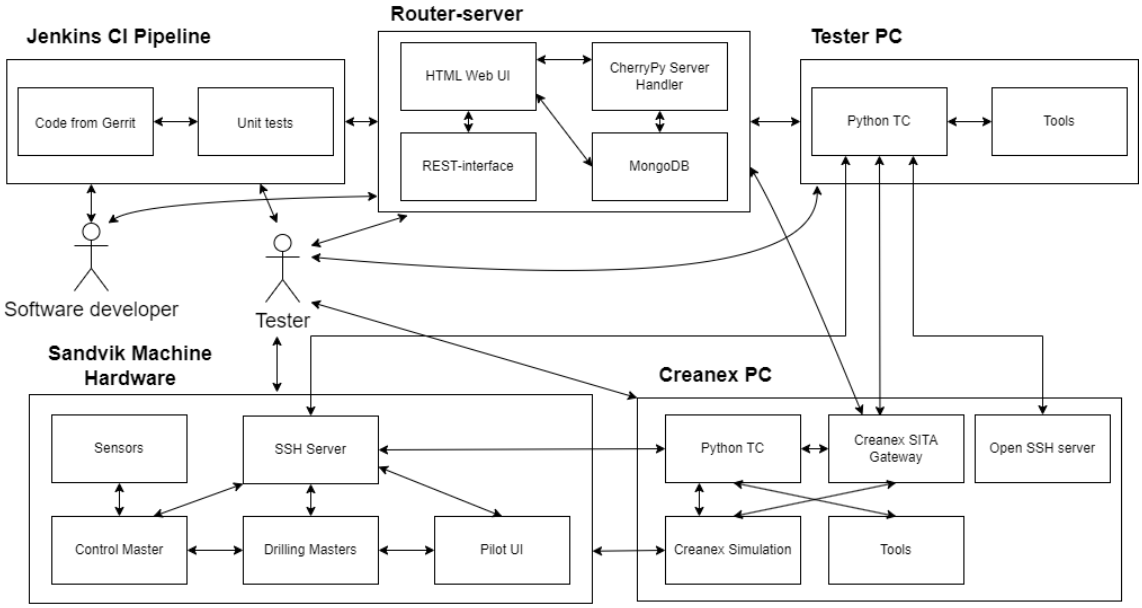
5.3 Reitittimen lopullinen arkkitehtuuri

Reititinarkkitehtuurin lopullisessa versiossa yhdistyi elementtejä sekä hub-and-spoke- ja reititinarkkitehtuurista. Toisaalta reitittimessä oli reitittimen mukainen reititystaulu, mutta toisaalta sen rajapinta oli laajempi eikä se vain keskittynyt viestin välittämiseen vaan myös sen muokkaamiseen ja prosessoimiseen hub-and-spoke arkkitehtuurin tapaan. Voidaan sanoa, että lopullinen ratkaisu oli ominaisuuksiltaan eräänlainen hub-and-spoke- ja reititinarkkitehtuurin yhteenliittymä.

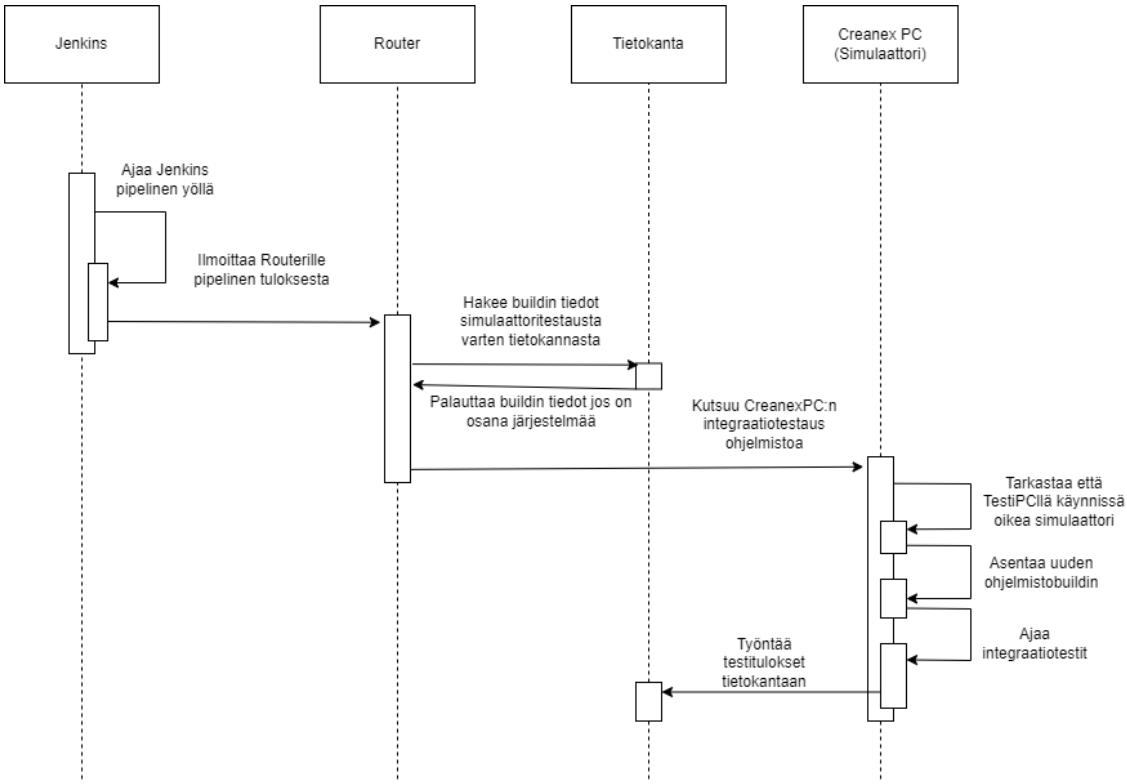
Reititinarkkitehtuurin rakennetta käsitellään erilaisin UML-diagrammein. Esimerkiksi Sandvikilla reititinarkkitehtuuri toteutettiin kuvan 28 mukaisesti liittämällä Jenkins CI-pipeline hub-and-spoke pohjaisen reitittimen kautta Creanex PC-simulaattoreihin. Samalla luotiin reitittimeen selaimella käytettävä html-käyttöliittymä, jonka kautta testaajat voivat myös manuaalisesti käyttää reitittimen tarjoamia toimintoja esimerkiksi

hakemalla ja vertailemalla testituloksia tietokannasta sekä muokkaamalla että käynnistämällä osia testiautomaatioputkesta manuaalisesti eri simulaattoreilla

Toisin kuin luvussa 5.1.2 reititinarkkitehtuuri esimerkissä, reitittimen ja Creanex PC:n välinen kommunikaatio ei kuitenkaan tapahdu SSH-yhteyden ylitse vaan http-protokollan avulla. Kommunikaatiossa käytetään hyödyksi Creanex PC-simulaattoreilla käynnissä olevaa REST-pohjaisia ”Creanex SITA Gateway” -palvelinta. Rajapinta toimii siten, että reititin kutsuu verkossa olevaa ”gatewayta”, joka käynnistää pyynnön mukaisesti osia testiautomaatiosta. Tähän päädyttiin, koska SSH-yhteyden ylitse ei pysty helposti toteuttamaan simulaattoriohjelmiston ajamiseen riittäviä Windowsin käyttöoikeuksia. Lisäksi ”gateway” oli jo olemassa oleva järjestelmä, jota käytetään osana testiautomaatiota, mikä helpotti sen käyttöönottoa osana järjestelmäintegraatiota. Jenkinsin toiminnan mahdollistamiseksi reititin myös rekisteröitiin Jenkinsille orjana, jolloin Jenkins pystyy ajamaan komentoja reititinpalvelimella. Toteutunut arkkitehtuuri ja sen rajapinnat ovat nähtävissä kuvassa 28. Kuvan 28 pohjalta CI-pipelineä ja sen toimintaa voi myös kuvata sekvenssikaavioilla (Kuva 29).

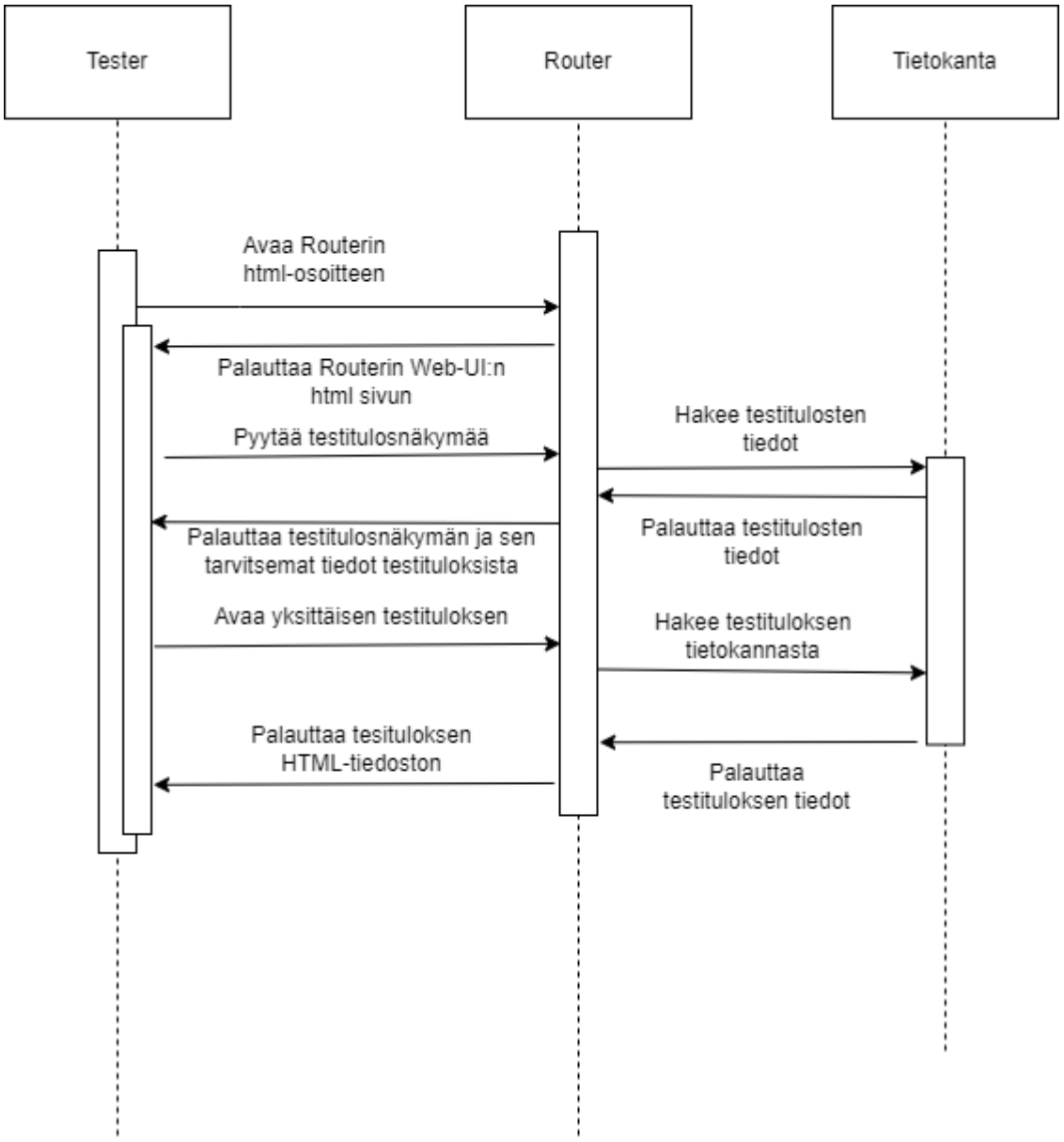


Kuva 28. Sandvik-testiympäristön lopullinen reititinarkkitehtuuri.



Kuva 29. Sekvenssikaavio CI-pipelinen toiminnnasta.

Kuten kuvasta 29 nähdään, reititinarkkitehtuurissa voidaan ajatella olevan neljä rajapintaa, jotka ovat Jenkins, reititin, tietokanta ja Creanex PC-simulaattori. Näistä kuitenkin reititin ja tietokanta voisivat Sandvikilla sijaita fyysisesti joko samassa tai erillisessä sijainnissa riippuen fyysisestä toteutuksesta. Tässä tutkimuksessa reititin ja MongoDB-tietokanta sijaitsevat samalla palvelimella, vaikka mikään reititinarkkitehtuurin tai hub-and-spoke-arkkitehtuurin osalta ei tätä vaadi. Tämä on nähtävissä kuvasta 28.



Kuva 30. Front-endin ja back-endin välinen kommunikaatio testitulosten hakemisessa.

Yleinen rakenne reitittimen kaikissa toiminnoissa on, että joko Jenkins tai testaaja kommunikoivat reititinpalvelimen kanssa http-pyyntöjen avulla. Tämä toimii siten, että joko Jenkins käynnistää GET-pyynnöillä osia palvelimen pipelinen toiminnoista tai testaaja käyttää selaimella palvelimen front-endiä ja tekee sen UI:n kautta pyyntöjä palvelimelle. Pyyntö käynnistää reitittimellä siitä vastaavan metodin tai metodit, jotka hakevat tarvittaessa tiedot MongoDB-tietokannasta ja palauttavat tulokset joko sellaisenaan tai muokattuna front-endin UI:lle (Kuva 30). Toiminta kommunikoidessa reitittimen rajapinnan kanssa on luvun 3.5 REST-arkkitehtuurin mukaista. Tätä ja sen mahdollistamaa käyttöliittymää käsitellään seuraavassa alaluvussa 5.3.1. Lisäksi sitä seuraavassa alaluvussa käsitellään MongoDB-tietokantaa ja miksi siihen päädyttiin.

5.3.1 Käyttöliittymä ja REST-arkkitehtuuri

Käytännön osion rajapinnan toteutuksessa käytettiin luvun 3.5 REST-arkkitehtuuria. Sen mukaisesti lopullisessa ratkaisussa front-end käyttää http-protokollan mukaisesti get-, post- ja delete-metodeita resurssien manipulointiin back-endissä. Esimerkit front-endin post-pyynnöstä on nähtävissä kuvasta 31 ja back-endin post pyyntöjä käsittelevästä metodista ja sen käsittelijästä (engl. handler) kuvasta 32.

```
export function postFunction(url, timeout=1000, data="", handler="", successText="", useResponseText=false){

    var xhr = new XMLHttpRequest();
    xhr.timeout = timeout;
    xhr.open("POST", url, true);
    xhr.setRequestHeader('Content-Type', 'application/json');
    xhr.ontimeout = () => {
        console.error('The request for '+url+' timed out.');
        openPopupFunction("/static/images/cancel_icon.png", "Connection Error!", "Server could not be reached!");
    };

    xhr.onload = () => {
        if (xhr.readyState === 4) {
            console.log(xhr.status)
            console.log(xhr.statusText)
            if (xhr.status === 200) {
                console.log('The POST request for URL "' + url + ' and handler "'+ handler +'" Exited as OK');
                if (successText != ""){
                    if (useResponseText == true){
                        openPopupFunction("/static/images/OK_icon.png", "Success!", xhr.responseText);
                    }
                    else{
                        openPopupFunction("/static/images/OK_icon.png", "Success!", successText);
                    }
                }
            }
            else {
                console.log(xhr)
                console.error(xhr.statusText);
                var text = "When sending POST request received response: " + xhr.responseText +" from: " + url
                openPopupFunction("/static/images/cancel_icon.png", xhr.statusText, text);
            }
        }
    };

    var sentdata = {};
    sentdata['handler'] = handler;
    sentdata['data'] = data;
    var json = JSON.stringify(sentdata);
    xhr.send(json);

    return xhr.status;
}
```

Kuva 31. Front-endin http post-pyyntö.

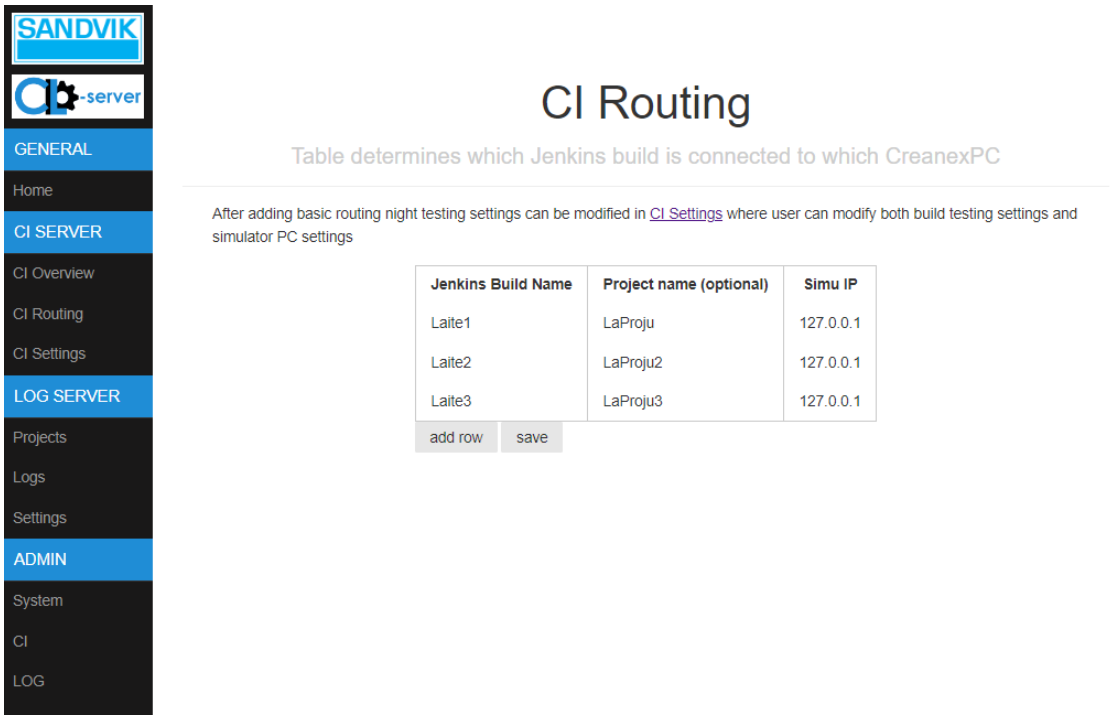
```python
@cherrypy.expose
@cherrypy.tools.json_in()
def POST(self):
  """
  Info: Secure POST handler for handling post requests, POST data is dict which contains two main sections
        "handler", which contains name of the handler to be used
        "data", which contains data to be used inside of the handler
  """
  input_json = cherrypy.request.json
  handler = input_json["handler"]
  data = input_json["data"]
  #print(cherrypy.request.json)
  try:
    CI_log.fnWriteToLog("Received POST request to SecureWebService: "
    +cherrypy.request.headers["REFERER"]+", from: "+cherrypy.request.headers["Remote-Addr"]
    + ", and used handler parameters: "+handler)

    if handler == "postpipelinerouting":
      #Handler for handling new CI Routing table
      projects = data["data"]
      updateCIProjectRoutingSettings(projects)
      createOrRemoveCISimulatorSettings(projects)
```

Kuva 32. Back-endin http post-pyynnön käsittelijä.

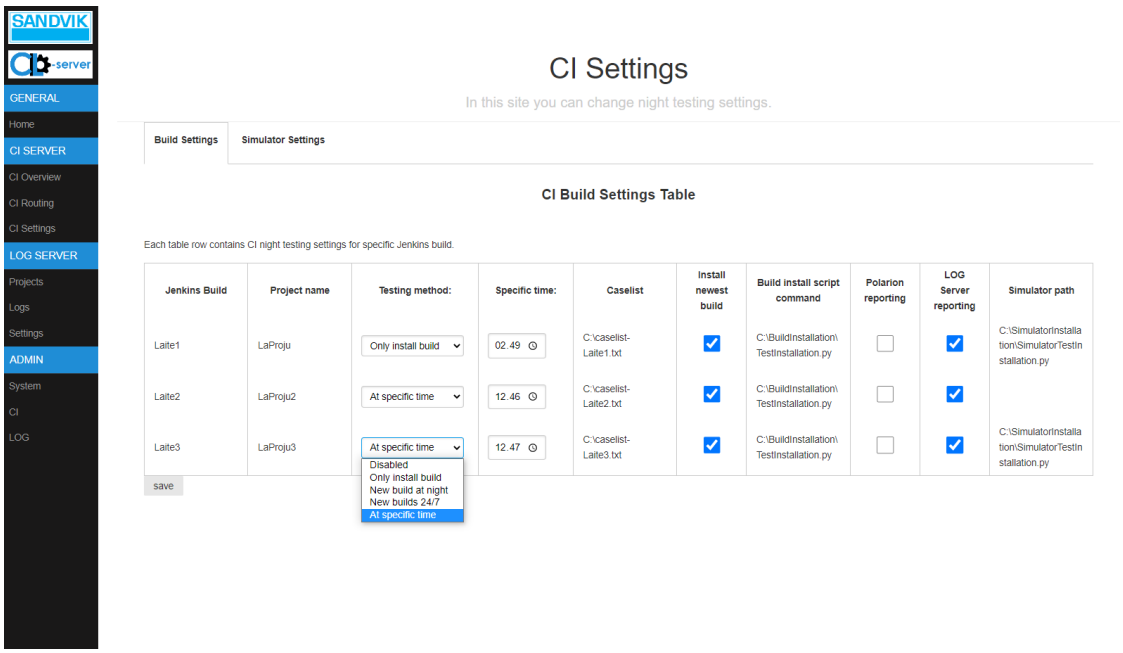
Kuten edellä olevista kuvista 31 ja 32 voi päätellä, REST-arkkitehtuurin mukaisesti toteutettu palvelin ja sen front-end käyttävät kommunikaatiossa JSON-tiedostoja ja käyttöliittymää manipuloidaan osittain front-endissa suoritettavalla javascript-koodilla. Lisäksi palvelimessa on toteutettu välimuistipohjainen autentikaatio, joka on osana pythonin cherrypy-kirjastoa ja resurssien haluttu tila voidaan määritellä URL-osoitteen avulla. REST-arkkitehtuuri näkyy myös palvelimen rajapinnan URL-osoitteen rakenteessa, jossa resurssin haluttu tila on selkeästi nähtävissä. Tämän voi nähdä kuvasta 33 jossa näkyvä URL-osoite ohjaa kuvan 37 lokitulos sivulle määritellyillä hakuparametreilla.

Kuva 33. REST-arkkitehtuurin mukainen URL, jossa resurssin haluttu tila.

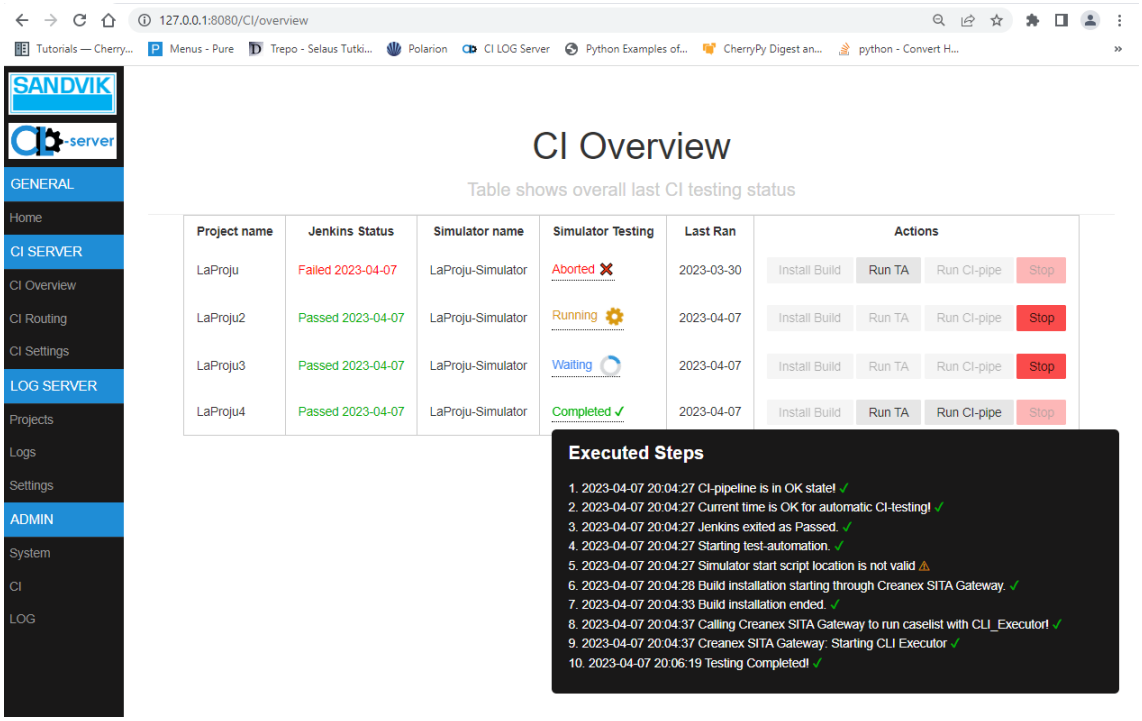
Reititinpalvelimeen on myös käyttöliittymä, jonka avulla voidaan paitsi muokata testiautomaation CI-putken kannalta oleellista reititystä Jenkinsin käännöspaketin ja simulaattorin välillä, määritellä jokaisen käännöspaketin testaamiselle omat asetukset, seurata CI-putken vaiheita ja etenemistä sekä hakea ja muokata tietokannan testituloksia. Nämä ovat nähtävissä kuvissa 34, 35, 36 ja 37.



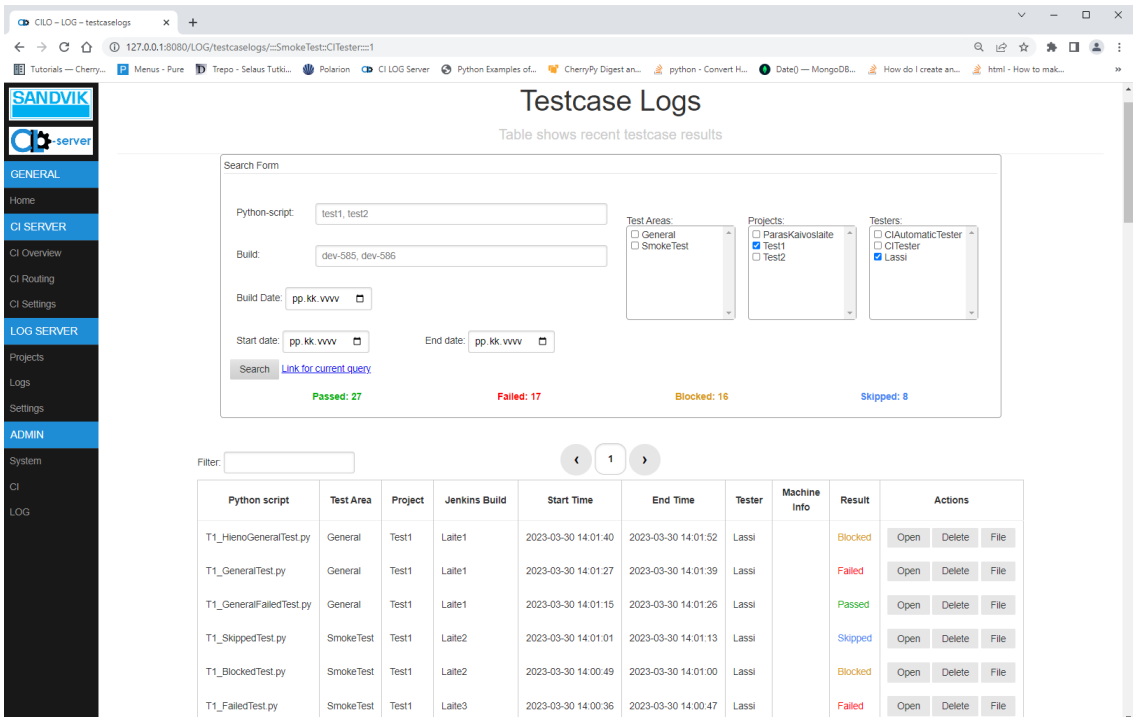
Kuva 34. Reititinarkkitehtuurin reititystaulu.



Kuva 35. CI-putken asetukset.



Kuva 36. CI-putken tilan seuranta.



Kuva 37. Käyttöliittymä testitulosten hakemiseen ja muokkaamiseen.

Kaikki edellä olevien kuvien ominaisuudet ovat tärkeitä paitsi hajautetun järjestelmän integraation seuraamisessa, mutta myös sen hyötyjen maksimoinnissa kuten on aiemmin mainittu ja perusteltu tarkemmin luvussa 3.1.1. Reititinarkkitehtuuri yhdessä REST-arkkitehtuurin kanssa mahdollistaa yksinkertaisen käyttöliittymän, jota on helppo laajentaa ja tietokannan, johon järjestelmän tiedot saadaan talletettuna keskitetysti.

Seuraavassa alaluvussa käsitellään tarkemmin tämän toteutuksen kannalta tärkeää tietokantaa ja sen toteutusta tarkemmin.

5.3.2 MongoDB-tietokanta

Luvun 3.7 SQL- ja NoSQL-teorian osalta voitaan todeta, että NoSQL-pohjaiset oliotietokannat mahdollistavat suurien ja hajautettujen tietokantojen luomisen. Lisäksi toisin kuin relaatiomaiset tietokannat (RDBMS) joihin SQL sisältyy, NoSQL-tietokannat, jotka eivät tarjoa relaatiomaisuutta tietokannassa ovat nopeampia tietoa muokattaessa ja poistettaessa. NoSQL-pohjaisia tietokantoja eivät myöskään rajoita tietokantaan lisättävää tietoa sen muodon mukaan toisin kuin SQL.

Nämä kaikki NoSQL-edut ovat tärkeitä Sandvikin testiautomaatioympäristöä integroitaessa, jonka takia tässä työssä päädyttiin NoSQL-pohjaiseen ratkaisuun. Luvun 3.7.2 pohjalta NoSQL on parempi sillä tietokantoihin lisättävän tiedon muoto voi poiketa hyvin paljon ja uusia tietomuotoja voi tulla lisää riippuen erilaisista testausympäristön sovelluksista. Tällaisia sovelluksia erilaisilla tiedoilla ovat esimerkiksi Jenkins, simulaattorin laitteisto ja testausohjelmisto. Myös tietokannan hajauttaminen useille eri palvelimille voi tulla kyseeseen tulevaisuudessa, vaikka sitä ei tämän työn puitteissa toteuteta. Myös tietokannan muokattavuus on tärkeää, koska tietokannassa olevia testituloksia ja testiautomaation asetuksia täytyy pystyä muokkaamaan tehokkaasti. Lisäksi NoSQL-pohjaisten tietokantojen tarjoama yhteinen ja helppokäyttöinen rajapinta tarjoaa helpon ja yksinkertaisen tavan hakea-, lisätä-, poistaa- ja muokata tietoa eri sovelluksista käsin.

Suosittuja NoSQL-tietokantoja ovat esimerkiksi MongoDB, Amazon Dynamo DB, Couchbase ja Google Cloud Firestone. Näistä tässä työssä käytetään MongoDBtä sen takia, että siitä oli Sandvikilla eniten kokemusta. MongoDB on myös pärjännyt hyvin tietokantojen suoritus vertailuissa [30] [31]. Tämän lisäksi MongoDB:n vuonna 2018 voimaan tullut lisenssi sallii ilmaisen MongoDB-palvelimen ylläpidon niin kauan kuin MongoDBtä käyttämä sovellus ei ole kolmansien osapuolien eli esimerkiksi asiakkaiden saatavilla. Lisäksi lisenssi sallii kolmansien osapuolien käyttää tietokantaa käyttävää sovellusta, mikäli sovelluksen koko ohjelmakoodi on julkisesti saatavilla. [14, luku 13] Näistä lisenssin kohdista tärkein on ensimmäinen eli se että MongoDB-tietokantaa saa käyttää sovelluksessa ilmaiseksi, kunhan kolmannet osapuolet eivät pääse käsiksi sovellukseen. Tämä johtuu siitä, että se on tämän työn puitteissa täysin riittävä ja takaa sen, ettei Sandvikille kehitettävän järjestelmän lähdekoodia tarvitse julkaista.

MongoDB-palvelimen tietokantaan pääsee myös helposti käsiksi pythonin PyMongo-kirjastolla. Kirjastolla pystyy helposti lisäämään tietokannan osia eli ”Collectioneita”. ”Collectioneihin” pystyy myös lisäämään tietoa esimerkiksi komennolla $result = db.collection.insert\_one(\{"Nimi": "Otto", "ikä": "24"\})$ ja hakemaan vastaavasti tietoa komennolle $result = db.collection.find\_one(\{"Ikä": "24"\})$. [3] [15] Tämän helpon rakenteen ja rajapinnan ansiosta MongoDBseen on helppoa suunnitella metodeja sekä rakentaa erilaisia tietokantaratkaisuja. Esimerkiksi luoda erilaisia prosessiketjuja kuten testiautomaatioputken, jossa uusin ohjelmisto ensin asennetaan simulaattoriin, sitten testataan ja lopuksi tulokset lähetetään tietokantaan. Python tuki on tärkeä sillä kuten luvussa 4.1.2 kerrotaan, integraatio- ja järjestelmätestauksen testiskriptit ovat Python-pohjaisia. PyMongo ja MongoDB mahdollistavat myös helpon testitulokset lisäämiseen hakemisen ja vertailuun. Nämä ovat keskeisiä järjestelmäintegraation hyötyjä, jotka voidaan mahdollistaa keskitetyllä tietokannalla kuten on aiemmin kerrottu luvussa 3.1.1. Rajapinnan helppous vähentää myös järjestelmäintegraation haasteita, esimerkiksi järjestelmän luomiseen vaadittavia alkukustannuksia, järjestelmän hallinnointia ja kommunikointia, jotka mainitaan luvussa 3.1.2.

6. LOPPUTULOKSET JA JATKOKEHITYS

Tässä luvussa kuvataan diplomityön käytännönosuuden lopputulosta, toteuttiko reititinarkkitehtuurilla tehty järjestelmäintegraatio työn tavoitteet ja toteutuivatko arkkitehtuurin teorian mahdolliset hyödyt ja haasteet. Luvun ensimmäisessä alaluvussa verrataan Sandvikin testiympäristön lähtötilaa ja uutta ratkaisua sekä saavutettiinko luvuissa 4.2 ja 4.3 mainitut kehitystarpeet ja tavoitteet. Toisessa alaluvussa verrataan reititinpohjaisen ratkaisun teoriaa toteutuneeseen reititinpohjaiseen järjestelmään sekä toimiiko se luvun 3 teorian pohjalta. Kolmannessa alaluvussa käsitellään olisiko työn voinut toteuttaa paremmin muilla arkkitehtuuriratkaisuilla vai oliko valittu ratkaisu paras mahdollinen työn tutkimuskysymyksien, -teorian ja -lopputuloksen näkökulmasta. Neljännessä alaluvussa käsitellään vielä työn mahdollista jatkokehitystä ja miten järjestelmää voitaisiin parantaa tulevaisuudessa.

6.1 Lähtötilan ja uuden ratkaisun vertailua

Tämän diplomityön käytännön osuuden alkutilan arkkitehtuuri on nähtävissä kuvassa 1. Työn käytännön osuuden tavoitteena oli yhdistää jatkuvan integraation palvelu Jenkins Sandvikin hajautettuun simulaattoritestausympäristöön, joissa integraatio- ja järjestelmätestaus toteutetaan. Hajautetun järjestelmän integraatio päätettiin toteuttaa yhdistämällä lukujen 3.3.2 hub-and-spoke-arkkitehtuuri ja 5.3 reititinarkkitehtuuri. Sandvikin uuden testiympäristön UML-kaavio on nähtävissä kuvassa 28. Ratkaisussa hub-and-spoke-pohjainen reititin kommunikoi Jenkinsin ja simulaattoreiden välillä sekä tarjoaa testaajille REST-pohjaisen rajapinnan järjestelmän hallinnointiin.

Järjestelmäintegraatiolle asetettiin Sandvikilta erilaisia tavoitteita ja toiveita, jotka ovat nähtävissä luvun 4.3 taulukosta 2. Työn käytännön osuuden tavoitteista saavutettiin valitulla arkkitehtuuriratkaisuilla kaikki pakolliset sekä suurin osa tärkeistä ja toivotuista ominaisuuksista. Lisäksi loput tärkeistä ja toivotuista ominaisuuksista huomioitiin arkkitehtuuria suunniteltaessa ja ne voitaisiin toteuttaa tulevaisuudessa (Taulukko 3).

Taulukko 3. Sandvikin uuden ratkaisun tavoitteiden toteutuminen.

Pakollinen	**Toteutuiko?**	**Tärkeä**	**Toteutuiko?**	**Toivottava**	**Toteutuiko?**
Jenkins käännöspaketin integroiminen osaksi testiautomaatio järjestelmää	Kyllä	Laitteen simulaattoriohjelmiston vaihtaminen osana CI-putkea	Kyllä	Tietokannan varmuuskopion palauttaminen	Kyllä
Jenkins käännöspaketin integroinnin poistaminen järjestelmästä	Kyllä	CI-putken tilan seuranta	Kyllä	Projektien testituloksien vertailu	Kyllä
Jenkins käännöspaketin tilan tarkastelu	Kyllä	Uusimpien tulosten vertailu	Kyllä	Testitulosten vertailu suhteessa aikaan	Osittain
Jenkinsin käännöspaketin automaattinen asentaminen testattavaan järjestelmään	Kyllä	Vianseuranta	Kyllä	Tietokannan säännöllinen varmuuskopioinnti	Osittain
Jatkuva integraatio- ja järjestelmätestaus	Kyllä	Testituloksien hakeminen	Kyllä	Testitulokset sähköpostiin	Ei
CI-putken asetuksien määrittely	Kyllä	Useiden projektien testaaminen samalla simulaattorilla	Kyllä	Ajetun testitapauksen koodin tarkastelu	Ei
Testituloksien tallentaminen	Kyllä	24/7 automaatiotestaus	Osittain	CI-putken "stop"-nappi	Ei
Testituloksien poistaminen	Kyllä				
Testituloksien hakeminen	Kyllä				

Uusi järjestelmä mahdollistaa ja hyödyntää vanhaa testiautomaatio-ohjelmistoa, sekä sitoo yhteen ohjelmistokehityksen ja testiautomaation osaksi yhtä ja samaa arvoketjua.

Tämän työn johdannossa jo mainittu laajennettavuus huomioitiin keskittämällä kaikki järjestelmän hallintaan liittyvät asetukset reititinpalvelimelle, kuten on nähtävissä kuvasta 25 ja 28. Reititinarkkitehtuuri mahdollistaa sen, että Jenkins-pipelinea integroitaessa täytyy lisätä vain käännöspaketin reititys ja asetukset kuvien 33 ja 34 mukaisesti.

Odottamattomissa ongelmatilanteissa järjestelmä on myös pyritty kehittämään mahdollisimman pitkälle käyttäen jo olemassa olevia sovelluksia. Esimerkiksi testiautomaation ajaminen vaatii Windows-tietokoneen järjestelmänvalvojan oikeuksia, joita ei SSH:lla onnistuttu saavuttamaan. Tämän takia päädyttiin käyttämään järjestelmänvalvojana käynnistettävää Creanex SITA Gatewayta SSH:n sijaan. Kaikilla simulaattoreilla käynnissä oleva Creanex SITA Gateway -palvelin on ainut reitittimestä irrallaan oleva järjestelmä, josta järjestelmän toiminta on riippuvainen. Lisäksi laajennettavuus huomioitiin REST-pohjaisessa arkkitehtuurissa, jossa reitittimen rajapintaan pääsee http-protokollalla käsiksi Sandvikin toimistoverkosta. Näin reitittimen tarjoamia metodeja voi helposti käyttää tarvittaessa myös tulevaisuudessa eri sovelluksissa.

Edellä mainitun pohjalta voidaan sanoa, että reititinarkkitehtuurilla toteutettu järjestelmä täyttää Sandvikilla testijärjestelmän integraatiolle asetetut pakolliset tavoitteet. Lisäksi arkkitehtuurilla saatiin toteutettua suurin osa tärkeistä ja toivotuista ominaisuuksista. Luvussa 5 esiteltiin kolmea erilaista tapaa toteuttaa luvun 3 pohjalta Sandvikilla hajautetun järjestelmän integraatio, että se voisi täyttää sille asetetut vaatimukset ja tavoitteet. Luvussa 5 myös toteutettiin näistä hub-and-spoke-pohjainen reititinarkkitehtuuri. Tämä toteutti luvussa 3.1.1 olevat mahdolliset järjestelmäintegraation hyödyt tiedon keräämisestä ja prosessoinnista. Samalla se ratkaisi luvun 3.1.2 haasteet järjestelmän kommunikoinnista, -hallinnasta ja -rajapinnoista Sandvikin testausympäristössä.

6.2 Reitittimen teorian ja toteutuksen vertailua

Reitittimen teoriaa luvusta 3.4.2 voidaan verrata toteutetun reitittimen ominaisuuksiin. Näin voidaan tarkastella, onko käytännön toteutuksella samat hyödyt ja haasteet kuin teorian pohjalta tulisi olla. Kuten luvussa 3.4.2 todetaan, reititin ratkaisussa yhdistetään reititinpohjainen kommunikaatio, hub-and-spoke-arkkitehtuuriin. Näin ollen teorian ja

toteutuksen vertailussa voidaan myös tutkia toteutuneen järjestelmän eroja luvun 3.3.2 hub-and-spoke-arkkitehtuurin hyötyihin ja haasteisiin, jotka on kerätty taulukkoon 1.

Teorian ja käytännön toteutuksen pohjalta toteutettiin taulukko, jossa arvioidaan, voitiinko teorian väitteet perustella todeksi, epätodeksi tai mahdolliseksi. Lisäksi myös perustelut sille miksi jokin on tosi, epätosi tai mahdollinen on nähtävissä taulukossa 4. Vastaavasti hub-and-spoke-arkkitehtuurin väittämät, se voitiinko ne todistaa käytännön toteutuksen perusteella todeksi, epätodeksi tai mahdolliseksi sekä perustelut ovat nähtävissä alla olevasta taulukosta 5.

Taulukko 4. Reitittimen teorian ja toteutuksen vertailua

Reitittimen teorian väittämät	**TOSI, EPÄTOSI tai MAHDOLLISTA**	**Perustelut käytännön tutkimuksen osalta**
Sallii kommunikaation useiden eri sovellusten välillä	TOSI	Käytännön toteutuksen perusteella reititinarkkitehtuuri mahdollisti kommunikaation useiden eri sovelluksien välillä
Mahdollistaa kaksi eri rajapintaa lähettäjälle ja vastaanottajalle	TOSI	Käytännön osiossa todistettiin, että reititinarkkitehtuuri pystyi tukemaan eri rajapintoja riippuen kutsutusta handlerista ja sen metodista
Vähentää rajapintojen monimutkaisuutta verrattuna hub-and-spoke-arkkitehtuuriin	TOSI	Hub-and-spoke pohjaisessa ratkaisussa viestinnässä kaikki sovellukset olisivat käyttäneet samaa rajapintaa mikä olisi vaikeuttanut järjestelmän laajentamista sekä vaatinut yksittäisten sovellusten laajemman muokkaamisen verrattuna reitittimen reititystauluun
Reitittimeen pitää määritellä kommunikaation säännöstö etukäteen	TOSI	Toisin kuin hub-and-spoke, joka voi välittää käytännössä mitä vain viestejä vapaasti luotavien metodien pohjalta, reitittimessä kaikkeen kommunikaatioon joudutaan käyttämään lähtökohtaisesti IP-osoitetta ja porttia
Reititin ei salli viestien purkua ja uudelleen kokoamista	MAHDOLLISTA	Ei voitu todistaa, sillä käytännön osiossa luotu reitittimen versio sisälsi muuta kuin vain viestin välitystä

Taulukko 5. Hub-and-spoken-osion teorian ja toteutuksen vertailua

Hub-and-spoke-osion teorian väittämät	**TOSI, EPÄTOSI tai MAHDOLLISTA**	**Perustelut käytännön tutkimuksen osalta**
Mahdollistaa keskitetyn datan keräämisen ja prosessoinnin	TOSI	Käytännön osiossa toteutettiin reitittimen "keskushubiin" MongoDB-tietokantaratkaisu sekä REST-rajapinta sen käyttämiseen ja prosessointiin
Mahdollistaa yhteisen kommunikaatiorajapinnan	TOSI	Käytännön osion lopullinen rajapinta toteutettiin täysin REST-arkkitehtuurin mukaisesti HTTP-protokollalla
Mahdollistaa tarvittavien yhteyksien määrän vähentämisen	TOSI	Käytännön osiossa todistettiin että sen sijaan että simulaattorit ja testaajien omat työtietokoneet kommunikoisivat keskenään, kommunikoinnin voi toteuttaa täysin hub-and-spoken kautta
Suuremmat aloituskustannukset	TOSI	Suurin osa tutkimuksen käytännön osuudesta meni reititinpalvelimen, tietokannan ja niiden rajapintojen kehittämiseen
Keskushubista voi tulla pullonkaula	MAHDOLLISTA	Vaikka tämän tutkimuksen aikana keskushubia ei koskaan rasitettu riittävästi, että siitä olisi voinut tulla pullonkaula, silti se voi olla mahdollista
Edullinen laajentaa	TOSI	Järjestelmän rajapinnan kehittämisen jälkeen järjestelmän integroiminen järjestelmään osoittautui helpoksi
Helppo monitoroida	TOSI	Käytännön osiossa kehitettiin keskitetty järjestelmän monitorointi, testitulosten kuin CI-putken osalta
Keskushubin kaatuminen kaataisi koko järjestelmän	TOSI	Keskushubin kaatuessa kaikki järjestelmien välinen kommunikaatio pysähtyisi ja järjestelmäintegraatio lakkaisi olemasta. Erilliset järjestelmät kuitenkin toimisivat normaalisti

Kuten yllä olevista taulukoista voidaan todeta, sekä reitittimen että hub-and-spoken teoriassa käsitellyt ominaisuudet sekä hyötyjen että haasteiden osalta vastasivat käytännössä täysin toteutetun käytännön reititinpalvelimen ominaisuuksia. Näin ollen

käytännön tutkimuksella pystyttiin todistamaan hub-and-spoke- ja reititinarkkitehtuurin teorian pitävyys. Luvun 3 teoriassa esitetyillä arkkitehtuurimalleilla pystyttiin tarjoamaan erilaisia tapoja vastata tutkimuskysymykseen siitä, millä eri tavoilla voidaan toteuttaa hajautetun järjestelmän integraatio. Lisäksi käytännön osiossa luvussa 5.1 pystyttiin esittämään näistä kolmen mallin toimivuus Sandvikin käytännön ympäristössä. Lopuksi luvussa 5.3 pystyttiin todistamaan näistä hub-and-spoke pohjaisen reititinarkkitehtuurin toimivuus Sandvikin testiympäristössä.

Näin ollen tämän diplomityön kontekstissa pystyttiin esittämään erilaisia vaihtoehtoja järjestelmän integraatioon, soveltamaan teoriaa käytäntöön kolmen eri esimerkin kautta sekä toteuttamaan ja todistamaan näistä yksi siten, että teoria ja käytäntö vastasivat toisiaan. Tämä ei kuitenkaan tarkoita, että ei olisi olemassa muita tapoja integroida hajautettu järjestelmä vaan tämä työ esittelee vain muutamia laajasti käytössä olevia vaihtoehtoja, joita voidaan soveltaa ja yhdistellä tarpeen mukaan. Tämä myös todistetaan tämän diplomityön kontekstissa, kun yhdistetään reitittimen- ja hub-and-spoke-arkkitehtuurin ominaisuuksia keskenään. Lisäksi myös rajapinnassa käytettiin SOA:ssa yleisesti käytettävää REST-rajapintaa.

6.3 Olisiko ollut parempia ratkaisuja

Tutkiessa sitä, olisiko ollut mahdollista toteuttaa Sandvikin testiympäristön integrointi paremmin, voidaan vertailla aiemman luvun 6.1 tuloksia siitä saavutettiinko työn alussa integroinnille asetetut tavoitteet. Lisäksi voidaan tutkia, olisiko jollain toisella luvussa 5 esitellyistä arkkitehtuuriesimerkeistä saavuttanut työn kannalta lopputuloksen, joka olisi täyttänyt paremmin integraatiolle asetetut tavoitteet.

Luvun 5.1 käytännön arkkitehtuuriesimerkkien pohjalta voidaan tehdä taulukko, (Taulukko 6) jossa vertaillaan eri arkkitehtuuriratkaisuiden toimivuutta Sandvikin testiympäristön integraatiossa.

Taulukko 6. Luvun arkkitehtuurien hyvät ja huonot puolet Sandvikin testiympäristössä.

	Hyvät puolet	**Huonot puolet**
Pipeline-arkkitehtuuri	-Kevyempi ja helpompi luoda Jenkinsin pohjalta -Yksinkertainen rakenne -Rakentuu pitkälti nykyisen arkkitehtuurin varaan -Pipelinet toimisivat toisista riippumatta -Ei olisi vaaraa pullonkauloille	-Vaikea laajennettavuus, koska vaatii erillisten pipelinejen muokkaamista -Vaatisi erillisen tietokannan testituloksille -Vaatisi erillisen sovelluksen tietokannan tarkasteluun -Vaikeampi ylläpitää, koska logiikka useissa eri pipelineissä erikseen -Virhetilanteiden hallinta vaikeampaa koska paljon erillisiä pipelinejä -Ei välttämättä tarjoa tarvittavia rajapintaratkaisuja Sandvikin testiautomaatiojärjestelmään
Reititinarkkitehtuuri	-Keskitetty käyttöliittymä -Helppo laajennettavuus -Helppo ylläpito -Keskitetty testiautomaation hallinta -Keskitetty tietokannan hallinta -Rajapinta Jenkinsin ja Sandvikin testiautomaation välillä täysin muokattavissa	-Työläämpi toteuttaa -Vaatisi paremman verkkoinfrastruktuurin tueksi, jotta reititin saa yhteyden kaikkiin simulaattoreihin ja Jenkinsiin -Voi aiheuttaa pullonkaulan reitittimeen -Reitittimen kaatuessa mikään CI-putkessa ei toimi
Palvelupohjainen arkkitehtuuri	-Helppo laajennettavuus -Tarjoaa yhteisen ja yleispätevän rajapinnan -Helppo ylläpitää -Kaikki sovellukset toimisivat toisistaan riippumatta -Ei sisällä reitittimen tapaan pullonkauloja -Myös muut sovellukset voisivat helposti hyödyntää samaa tietokantaa ja verkon sovelluksia	-Hyvin työläs toteuttaa -Vaatisi paremman verkkoinfrastruktuurin tueksi, jotta kaikki simulaattorit voivat hyödyntää SOA-pohjaista verkkorakennetta -Vaatisi mahdollisesti nykyisten sovellusten muokkaamista SOA:n rajapintaan sopivaksi

Kuten edellä olevasta taulukosta voidaan nähdä, arkkitehtuureista ylivoimaisesti yksinkertaisin, mutta lopputulokselta huonoin on pipeline-arkkitehtuuri, jossa nykyisiin Jenkins pipelineihin lisättäisiin lisävaiheita simulaattoritestausta varten. Pipeline-arkkitehtuurin huonot puolet johtuvat pääosin siitä, että arkkitehtuuriratkaisu olisi hajautettu eri pipelinejen välillä. Sen takia tämän diplomityön tavoitteissa mainittu helppo laajennettavuus ja ylläpito olisi ollut vaikeampaa toteuttaa.

Taulukosta myös nähdään, että reititinarkkitehtuuri ja SOA vastaavat huomattavasti paremmin hajautetun järjestelmän integraatioon sekä laajennettavuuden että ylläpidon osalta. Tämä johtuu siitä, että kyseisissä arkkitehtuureissa olisi mahdollista toteuttaa kaikille simulaattoreille yhteinen rajapinta, palvelut ja logiikka. Sen sijaan kyseiset arkkitehtuurit olisivat monimutkaisempia toteuttaa ja vaatisivat enemmän verkkoinfrastruktuurilta. Reititin- ja SOA-arkkitehtuurit myös mahdollistaisivat laajemmin arkkitehtuurin mukauttamisen Sandvikin testiympäristön tarpeisiin sopivaksi. Tämä voisi helpottaa tulevaisuudessa jatkokehitystä ja auttaa välttämään erilaiset ongelmat järjestelmän kehityksen aikana.

Tämän pohjalta voidaan sanoa, että Jenkins pohjainen pipeline-arkkitehtuuri ei olisi soveltunut yhtä hyvin Sandvikin testijärjestelmän integraatioon rajapintaongelmien takia. Myös pipeline-arkkitehtuurilla järjestelmän laajennettavuus olisi ollut vaikeammin toteutettavissa. Sen sijaan SOA-arkkitehtuurilla olisi ollut mahdollista saavuttaa kaikki hub-and-spoke pohjaisella reititin arkkitehtuurilla saavutetut ominaisuudet. Vaikka SOA:lla olisi voitu saavuttaa kaikki reititinarkkitehtuurin edut, järjestelmän aloituskustannukset olisivat olleet reititinarkkitehtuuria suuremmat. SOA:n etuna olisi kuitenkin ollut joustavampi rajapinta, jonka avulla nyt kehitetty järjestelmä olisi voitu integroida tulevaisuudessa helpommin osaksi laajempaa kokonaisuutta. Sille ei kuitenkaan ollut tämän tutkimuksen käytännön toteutuksessa tarvetta. Nykyisen reititinarkkitehtuurin voisi myös tulevaisuudessa muuttaa helposti SOA-pohjaiseksi, kuten on nähtävissä kuvasta 27. Tämä saavutettiin sen ansiosta, että reititin toteutettiin REST-arkkitehtuurin mukaisesti.

Kuten aiemmin taulukon 6 pohjalta todetaan, valitulla arkkitehtuuriratkaisulla saavutettiin tärkeimmät testiympäristön integraatiolle asetetut tavoitteet. Lisäksi toteutumattomat ominaisuudet on huomioitu arkkitehtuurissa ja voidaan toteuttaa tulevaisuudessa. Näin ollen valittu arkkitehtuuriratkaisu toteutti Sandvikin testiympäristön integraatiolle asetetut tavoitteet. Tämän pohjalta ei voida sanoa, että jokin arkkitehtuuriratkaisu olisi soveltunut valittua hub-and-spoke-pohjaista reititinarkkitehtuuria merkittävästi paremmin. Esimerkiksi taulukon 6 pohjalta SOA:lla olisi voitu saavuttaa suunnilleen yhtä hyvä lopputulos. Toisaalta ei voida myöskään sanoa, ettei joku arkkitehtuuriratkaisu olisi voinut pystyä yhtä hyvään tai parempaan lopputulokseen. Tämän diplomityön kontekstissa esitellyistä arkkitehtuuriratkaisuista hub-and-spoke-pohjainen reititinarkkitehtuuri oli kuitenkin onnistunut vaihtoehto ja sen valinta onnistuttiin perustelemaan luvun 3 teorian ja Sandvikin ympäristön tarpeiden perusteella. Tämä on myös nähtävissä taulukossa 6.

6.4 Jatkokehityksestä

Sandvikin testausympäristöön kehitettyä reititinpalvelinta voisi myös jatkokehittää tulevaisuudessa vastaamaan yhä kasvaviin jatkuvan integraation tarpeisiin. Paitsi että valittua arkkitehtuuria voisi tarvittaessa muuttaa SOA:n mukaiseksi, järjestelmään voidaan myös kehittää loput taulukon 3 toteutumatta jäämistä ominaisuuksista.

Lisäksi järjestelmää voitaisiin laajentaa käyttämään muita Sandvikin integraatio ja järjestelmätestausta varten kehitettyjä sovelluksia. Esimerkiksi sovelluksia, jotka muokkaavat testattavan ohjelmiston ominaisuuksia ohjelmiston konfiguraatiotiedostoista. Vastaavasti järjestelmään voitaisiin kehittää esimerkiksi sähköpostipalvelu, joka lähettäisi asetuksissa valitun testaajan sähköpostiin automaatiotestauksen tuloksista.

Järjestelmän tietokantaa voisi myös kehittää tallentamaan testituloslokien lisäksi enemmän erilaisia testaukseen liittyviä tietoja. Näitä voivat olla esimerkiksi erilaiset konfiguraatiotiedostot ja virhelokit, joita voitaisiin hakea testattavalta järjestelmältä automaatiotestauksen epäonnistuessa. Tässä ongelmaksi voisi etenkin pidemmän ajan kuluessa osoittautua rajallinen tietokannan tila reititinpalvelimella. Tässä tilanteessa ajankohtaiseksi voi tulla MongoDB-tietokannan eriyttäminen omalle palvelimelle tai nykyisen palvelimen talletustilan lisääminen. Lisäksi vaikka tällä hetkellä reititinpalvelin ei lähetä testattavaa ohjelmistoa kuin yhteen simulaattoriin testattavaksi sille määritellyn kuvan 34 reitityksen pohjalta, niin mikään ei estä reitittimen muuttamista brokeriksi tulevaisuudessa ja sen vaatiman multicastin-kommunikaation lisäämistä nykyisen unicast-kommunikaation tilalle.

7. YHTEENVETO

Tässä diplomityössä tutkittiin sitä, millaisilla suunnittelumalleilla voidaan toteuttaa hajautetun järjestelmän integraatio, ja kuinka Sandvikille toteutettu järjestelmäintegraatio täyttää sille asetetut vaatimukset. Työn teoriassa perehdyttiin yleisesti ohjelmistokehitykseen ja testaukseen sekä tutkittiin erilaisia arkkitehtuuriratkaisuja, joilla voidaan toteuttaa järjestelmäintegraatio. Teoriassa tutkittiin myös mitä erilaisia viestintätapoja voidaan hyödyntää osana järjestelmän integraatiota. Työn käytännön esimerkki toteutettiin Sandvikin testausympäristössä, jossa integroitiin Jenkins CI-pipeline osaksi Sandvikin hajautettua integraatio- ja järjestelmätestausympäristöä.

Työn teoria toteutettiin tutkimalla yleisesti testausjärjestelmiä tieteellisten lähteiden perusteella keskittyen testauksen eri vaiheisiin, malleihin ja järjestelmiin. Tämän jälkeen käsiteltiin tieteellisten lähteiden pohjalta erilaisia järjestelmäintegraation malleja. Järjestelmäintegraation malleja verrattiin toisiinsa sen näkökulmasta, kuinka niillä pystyttäisiin toteuttamaan hajautetun järjestelmän integraatio. Tämän lisäksi teoriassa käsiteltiin viestinnän integraatiota, eri kommunikaatiotapojen perusteita, sekä relaatio- ja oliotietokantojen etuja ja haasteita hajautetun järjestelmän integraation näkökulmasta.

Työn käytännön osuus aloitettiin esittelemällä Sandvikin testausympäristö nykyisellään keskittyen ohjelmistokehitykseen ja integraatio- ja järjestelmätestaukseen. Luvussa myös kuvattiin Sandvikin testiympäristön kehitystarpeet ja integroidun järjestelmän vaatimukset ja tavoitteet.

Sandvikin testausjärjestelmän integroimisen toteuttaminen aloitettiin esittelemällä kolme vaihtoehtoista tapaa toteuttaa Sandvikin hajautetun testiympäristön integraatio. Näissä malleissa käytettiin aiemmin käsiteltyä teoriaa ja sovellettiin pipeline-, reititin- ja SOA-pohjaiset arkkitehtuuriratkaisut Sandvikin testiympäristön integraation toteuttamiseksi. Näistä valittiin hub-and-spoke pohjainen reititinarkkitehtuuri, koska se vastasi teorian pohjalta parhainten hajautetun järjestelmän integraatioon asetettuihin tavoitteisiin. Reititin oli myös teorian pohjalta helposti laajennettavissa ja sen aloituskustannukset eivät olleet yhtä suuret kuin SOA:n.

Reititinarkkitehtuuri toteutettiin REST-arkkitehtuurin mukaisesti siten, että toteutettu järjestelmä olisi tulevaisuudessa myös muutettavissa SOA-pohjaiseksi järjestelmäksi. Lisäksi REST-arkkitehtuurin etuna on selaimella toimiva käyttöliittymä sekä http-pohjainen viestintä. Näin pystyttiin hyödyntämään Sandvikilla jo automaatiotestauksessa käytössä olevaa Creanex SITA Gatewayta reitittimen ja simulaattoreiden välisessä

kommunikaatiossa. Tietokannan osalta päädyttiin käyttämään MongoDB-oliotietokantaa, koska oliotietokanta tarjoaa yksinkertaisemman rajapinnan ja tiedon muodosta välittämättömän tietokantaratkaisun, jota voidaan tarvittaessa tulevaisuudessa hajauttaa. Lisäksi MongoDB oli pärjännyt hyvin tietokantojen välisissä suoritusvertailuissa.

Työn lopputuloksena saavutettiin järjestelmä, joka täytti Sandvikin testiympäristön integraatiolle asetetut tavoitteet. Näin pystyttiin myös vastaamaan tutkimuskysymykseen siitä, kuinka toteutettu järjestelmäintegraatio täyttää sille asetetut vaatimukset. Toteutetun ratkaisun ominaisuudet vastasivat myös teoriassa esitettyjä ominaisuuksia. Työn teoriassa pystyttiin vastaamaan työn tutkimuskysymykseen siitä, millä eri tavoilla voidaan toteuttaa hajautetun järjestelmän integraatio. Tähän vastattiin esittelemällä erilaisia vaihtoehtoja hajautetun järjestelmän integraatioon. Lisäksi työn käytännön osiossa pystyttiin esittelemään näistä pipeline-, reititin- ja SOA-arkkitehtuurille käytännön esimerkit. Lopuksi pystyttiin todistamaan, että hub-and-spoke-pohjaisella reititinarkkitehtuuriratkaisulla pystyttiin toteuttamaan hajautetun järjestelmän integraatio, joka vastasi sille asetettuihin vaatimuksiin.

Tutkiessa sitä, olisiko työn käytännön osuuden voinut toteuttaa paremmin jollain muulla arkkitehtuuriratkaisulla voitiin todeta, että nykyisellä arkkitehtuurilla saavutettiin kaikki työn alussa määritellyt tavoitteet. Lisäksi nykyinen järjestelmä myös mahdollistaa jatkokehityksen tulevaisuudessa. Työn teorian ja Sandvikin järjestelmävaatimuksien pohjalta voidaan perustella valittu hub-and-spoke-pohjaisen reitittimen valinta tieteellisesti. Tämä ei kuitenkaan tarkoita sitä, että jollain vaihtoehtoisella arkkitehtuurilla ei olisi voitu päästä yhtä hyvään tai parempaan tulokseen. Tämä johtuu siitä, ettei tämän diplomityön kontekstissa pystytä käsittelemään teoriassa tai käytännössä kaikkia tarjolla olevia integraatioarkkitehtuurin malleja ja niiden kombinaatioita.

LÄHTEET

[1] Jenkins.op, Jenkins Pipeline, 2022. Verkkosivu saatavissa (viitattu 14.4.2023): https://www.jenkins.io/doc/book/pipeline/

[2] Creanex, Simulaattoriratkaisut, 2022. Verkkosivu saatavissa (viitattu 14.4.2023): https://creanex.fi/en/simulaattoriratkaisut/

[3] Pymongo, Collection level operations, 2022. Verkkosivu saatavissa (viitattu 14.4.2023): https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html

[4] G. Schmutz, D. Liebhart, P. Welkenbach, Service-Oriented Architecure: An Integration Blueprint, Packt Publishing, 2010. Saatavissa (viitattu 14.4.2023): https://ebookcentral.proquest.com/lib/tampere/reader.action?docID=950593#

[5] J. Nibert, M. Herniter, Z. Chambers, Model-Based System Design for MIL, SIL and HIL, World electric vehicle journal, 2012. Saatavissa (viitattu 14.4.2023): https://www.mdpi.com/2032-6653/5/4/1121

[6] F. Buschmann et al, Pattern-Oriented Software Architecture, Volume 1, A system of patterns, Wiley, 1996. Saatavissa (viitattu 14.4.2023): https://learning.oreilly.com/library/view/pattern-oriented-software-architecture/9781118725269/?sso_link=yes&sso_link_from=tampere-university

[7] W. Dougal, Oxford, E-business implementation: A guide to web services, EAI, BPI, e-Commerce, Content Management, Portals, and Supporting Technologies, Boston, Buttersworth-Heinemann, 2022. Saatavissa (viitattu 14.4.2023): https://ebookcentral.proquest.com/lib/tampere/reader.action?docID=296798

[8] A. Mili, Software Testing: Concepts and Operations, Wiley, 2015. Saatavissa (viitattu 14.4.2023): https://ebookcentral.proquest.com/lib/tampere/detail.action?docID=7104400&pq-origsite=primo

[9] R. Stephens, Beginning Software Engineering, Wiley, 2015. Saatavissa (viitattu 14.4.2023): https://ebookcentral.proquest.com/lib/tampere/detail.action?pq-origsite=primo&docID=1895174

[10] L. Wiese, Advanced Data Management: For SQL, NoSQL, Cloud and Distributed Databases, Berlin/Boston: Walter de Gruyter GmbH, 2015. Saatavissa (viitattu 14.4.2023): https://andor.tuni.fi/discovery/fulldisplay?docid=cdi_proquest_ebookcentral_EBC4179774&context=PC&vid=358FIN_TAMPO:VU1&lang=fi&search_scope=My_inst_and_CI_extended_search&adaptor=Primo%20Central&tab=Everything&query=any,contains,%5B10%5D%09L.%20Wiese,%20Advanced%20Data%20Management:%20For%20SQL,%20NoSQL,%20Cloud%20and%20Distributed%20Databases,%20Berlin%2FBoston:%20Walter%20de%20Gruyter%20GmbH&offset=0

[11] M. Massé, REST API Design Rulebook, O'Reilly Media Incorporated, 2011. Saatavissa (viitattu 14.4.2023): https://learning.oreilly.com/library/view/rest-api-design/9781449317904/ch01.html#section_introduction_hello_www

[12] N. Busi et al, Choreography and Orchestration: A Synergic Approach for System Design, Heidelberg: Springer Berlin Heidelberg, 2005. Saatavissa (viitattu 14.4.2023): https://link-springer-com.libproxy.tuni.fi/chapter/10.1007/11596141_18

[13] O. Habib, A Performance Analysis of Python WSGI Servers: Part 2, AppDynamics, 2016. Saatavissa (viitattu 14.4.2023): https://www.appdynamics.com/blog/engineering/a-performance-analysis-of-python-wsgi-servers-part-2/

[14] MongoDB, Server Side Public License (SSPL), 2015. Verkkosivu saatavissa (viitattu 14.4.2023): https://www.mongodb.com/licensing/server-side-public-license

[15] PyMongo, PyMongo 4.3.3 Documentation, 2022. Verkkosivu saatavissa (viitattu 14.4.2023): https://pymongo.readthedocs.io/en/stable/

[16] A. Berg, Jenkins continuous integration cookbook over 80 recipes to maintain, secure, communicate, test, build, and improve the software development process with Jenkins, Birmingham: Packt Pub, 2012. Saatavissa (viitattu 14.4.2023): https://andor.tuni.fi/discovery/fulldisplay?docid=cdi_igpublishing_primary_PACKT0003504&context=PC&vid=358FIN_TAMPO:VU1&lang=fi&search_scope=My_inst_and_CI_extended_search&adaptor=Primo%20Central&tab=Everything&query=any,contains,%5B16%5D%09A.%20Berg,%20Jenkins%20continuous%20integration%20cookbook%20over%2080%20recipes%20to%20maintain,%20secure,%20communicate,%20test,%20build,%20and%20improve%20the%20software%20development%20process%20with%20Jenkins,%20Birmingham:%20Packt%20Pub,%202012.

[17] N. Pathania, Learning Continuous Integration with Jenkins: a beginner's guide to implementing continuous integration and continuous delivery using Jenkins 2, Packt Publishing, 2017. Saatavissa (viitattu 14.4.2023): https://learning.oreilly.com/library/view/learning-continuous-integration/9781788479356/cover.xhtml

[18] J. Miller, Waterfall Versus Agile, IEEE software, 2020. Saatavissa (viitattu 14.4.2023): https://ieeexplore.ieee.org/ielx7/52/9121610/09121615.pdf

[19] E. Dustin, Effective Software Testing: 50 Specific Ways to Improve Your Testing, Boston : Addison-Wesley, 2002. Saatavissa (viitattu 14.4.2023): https://learning.oreilly.com/library/view/effective-software-testing/0201794292/ch06.html

[20] S. Asif, Software System Integration Middleware An Overview, International journal of computer applications, 2015. Saatavissa (viitattu 14.4.2023): https://research.ijcaonline.org/volume121/number5/pxc3904547.pdf

[21] M, Ilyas, S. Khan, Software integration in global software development: Challenges for GSD vendors, Wiley, 2017. Saatavissa (viitattu 14.4.2023): https://onlinelibrary-wiley-com.libproxy.tuni.fi/doi/full/10.1002/smr.1875

[22] C. A. Binildas, A. Christudas, Service oriented Jaba business integration enterprise service bus integration solutions for Java developers, Birmingham : Packt, 2008. Saatavissa (viitattu 14.4.2023): https://ebookcentral.proquest.com/lib/tampere/reader.action?docID=979976#

[23] G. Papadopoulos et al, Comparing Architectural Styles for Service-Oriented Architectures – a REST vs SOAP Case Study, Springer, 2009. Saatavissa (viitattu 14.4.2023): https://link-springer-com.libproxy.tuni.fi/book/10.1007/b137171

[24] B. Familiar, Microservices, IoT and Azure Leveraging DevOps and Microservice Architecture to deliver SaaS Solutions, Berkley, 2015. Saatavissa (viitattu 14.4.2023): https://learning.oreilly.com/library/view/microservices-iot-and/9781484212752/9781484212769_FM_0_Cover.xhtml

[25] Software TestingHelp, When To Opt For Automation Testing, 2023. Verkkosivu saatavissa (viitattu 14.4.2023): https://www.softwaretestinghelp.com/software-automation-testing-should-automate-project-testing/

[26] H. Youngsub et al, IEICE transactions on information and systems, Value-Driven V-model: From Requirement Analysis to Acceptance Testing, 2016. Saatavissa (viitattu 14.4.2023): https://www.jstage.jst.go.jp/article/transinf/E99.D/7/E99.D_2015EDP7451/_pdf

[27] B. Woolf, G. Hohpe, Enterprice integration patterns designing, building, and deploying messaging solutions, Boston: Addison-Wesley, 2003. Saatavissa (viitattu 14.4.2023): https://learning.oreilly.com/library/view/enterprise-integration-patterns/0321200683/

[28] E. Modiano, A. Sinha, Optimal Control for Generalized Network-Flow Problems, IEE/ACM transactions on networking, 2018. Saatavissa (viitattu 14.4.2023): https://ieeexplore-ieee-org.libproxy.tuni.fi/stamp/stamp.jsp?tp=&arnumber=8241883

[29] W. Ali et al, Comparsion between SQL and NoSQL Databases and Their Relationship with Big Data Analytics, Asian Journal of Research in Computer Science, 2019. Saatavissa (viitattu 14.4.2023): https://journalajrcos.com/index.php/AJRCOS/article/view/66

[30] L. Yishan, S. Manoharan, A performance comparison of SQL and NoSQL databases, IEE, 2013. Saatavissa (viitattu 14.4.2023): https://andor.tuni.fi/discovery/fulldisplay?docid=cdi_scopus_primary_370389345&context=PC&vid=358FIN_TAMPO:VU1&lang=fi&search_scope=My_inst_and_CI_extended_search&adaptor=Primo%20Central&tab=Everything&query=any,contains,SQL%20NoSQL%20comparison&offset=0

[31] M. Kausar, M. Nasar, A. Soosaimanickam, A Study of Performance and Comparison of NoSQL Databases: MongoDB, Cassandra, and Redis Using YCSB, Indian journal of science and technology, 2022. Saatavissa (viitattu 14.4.2023): https://sciresol.s3.us-east-2.amazonaws.com/IJST/Articles/2022/Issue-31/IJST-2022-1352.pdf