Lappeenrannan teknillinen yliopisto
School of Business and Management
Tietotekniikan koulutusohjelma

Diplomityö

Jani Wunsch

NOPEAN REITITYSVÄLIMUISTIN KEHITTÄMINEN

Työn tarkastajat: Professori Jari Porras
Diplomi-insinööri Jussi Väisänen

Työn ohjaaja: Diplomi-insinööri Jussi Väisänen

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
School of Business and Management
Tietotekniikan koulutusohjelma

Jani Wunsch

Nopean reititysvälimuistin kehittäminen

Diplomityö

2017

91 sivua, 9 kuvaa, 4 taulukkoa, 3 liitettä

Työn tarkastajat: Professori Jari Porras
Diplomi-insinööri Jussi Väisänen

Hakusanat: sovelluskehitys, laskujen reititys, opinnäytetyö
Keywords: software development, invoice routing, academic thesis

Opinnäytetyön tavoitteena oli rakentaa suorituskykyinen laskujen reitityspalvelu, joka toimisi välimuistityyppisesti ja tarjoaisi samat ominaisuudet kuin yrityksessä jo käytössä oleva vanha reitityspalvelu. Puhdas läpilukuvälimuisti tyyppinen palvelu osoittautui valittujen tekniikoiden yhteydessä toteutuskelvottomaksi. Lopulta projektissa päädyttiin toteuttamaan erilaisella tiedontallennusstrategialla toimiva välimuistipalvelu, joka tyydytti yrityksen tarpeet. Palvelun kehitysaikataulu venyi monimutkaisemman toteutuksen vuoksi, mutta ei vaikuttanut siihen liittyvän projektin aikatauluun ollessaan kooltaan ja työmäärältään huomattavasti pienempi projekti kuin uusi sähköisten laskujen välitysalusta. Hankaluuksista huolimatta projekti saatiin päätökseen ja reititysvälimuistipalvelu saatiin tuotantoon riittävällä suorituskyvyllä ja tulevaisuuteen skaalautuvalla arkkitehtuurilla.

ABSTRACT

Lappeenranta University of Technology
School of Business and Management
Degree Program in Computer Science

Jani Wunsch

Developing a high performance routing-cache service

Master's Thesis/Bachelor's Thesis/Seminar Report

91 pages, 9 figures, 4 tables, 3 appendices

Examiners: Professor Jari Porras
Master of Science Jussi Väisänen

Keywords: software development, invoice routing, academic thesis

The goal for this academic thesis work was to develop a high performance invoice routing service, which would serve routing data in read-through type cache manner and support routing features found from existing routing service in the company. Original idea of a read-through cache turned out to be unfeasible to develop with the chosen technologies. Later on in the project a different cache type of data storage model was chosen which enabled us to develop a routing service that met our company's needs. Due to a more complex software architecture requiring more work, the project schedule increased, but did not affect to the schedule of other project needing software from this project, as the related new invoicing platform project was bigger than this project. Despite of all the challenges the routing cache was put into production with high enough performance and with a possibility to scale the performance in future.

ALKUSANAT

Kiitän työnantajaani OpusCapita Oy:tä ja silloista esimiestäni Jussi Väisästä reititysvälimuisti projektista ja sen sallimisesta Diplomityön tekemiseen sekä Diplomityön tarkastamisesta.

Eritoten kiitän saamastani luottamuksesta jo ennen projektin muuttamista varsinaiseksi Diplomityöksi. Yhden miehen sovelluskehitysprojektiksi alkuperäinen ”Proof of Concept” projekti paisui mittasuhteissaan valtavaksi ja on tähän mennessä haastavin ja mielenkiintoisin IT-projektini työelämässä ja sen ulkopuolella.

Kiitän myös työtovereitani ja kavereitani henkisestä sekä käytännön tuesta, avusta ja tiedonjaosta muihin yrityksen tietojärjestelmiin liittyen, joilla on ollut suurta merkitystä Diplomityön tekemisessä ja koko sovelluskehitysprojektin onnistumisessa vaikka alkuperäisen idean mukaiseen lopputulokseen ei päästy.

Kiitän myös tyttöystävääni Piia Rinnettä työn oikoluvusta ja kieliasun tarkistuksesta, uskoisin tuon tehtävän olleen melko raskasta IT-alaa tuntemattomalle.

SISÄLLYSLUETTELO

1. JOHDANTO 4
1.1 Tavoitteet ja sisältö 5
1.2 Ongelman esittely, tavoitteet ja rajaukset 6
1.3 Työn rakenne 7
1.4 Käsitteiden esittely 8
2. TYÖN SUUNNITTELU JA TYÖKALUJEN VALINTA 9
2.1 Web-rajapinnan vaatimukset 9
2.2 Web-palvelin ja REST Web-sovellusalustat Pythonille 10
2.3 Tietovarasto: Redis 12
2.4 Palvelun suunnittelu karkeasti 14
3 REITITYSVÄLIMUISTIN PROTOTYYPIN TOTEUTUS 16
3.1 I –vaihe: Läpilatausvälimuisti 17
3.1.1 Suorituskykytestaus ja ongelmat testiympäristössä 18
3.1.2 Suorituskykytestauksen analysointi ja johtopäätökset 23
3.1.3 Vahinkohuomioita suorituskykytestauksesta 24
3.2 II-vaihe: Palvelun suunnittelu POC –kokemusten pohjalta. 26
3.2.1 Kuluttajareitityksen POC –toteutuksen suunnitelma 28
4 KULUTTAJAREITITYKSEN TOTEUTUS 32
4.1 Tietokantakirjastot ja merkistöongelmat 32
4.2 Kuluttajareittitiedon hallinta (B2CCacheManager) ja kuluttajareititys 33
4.2.1 Testejä ja ongelmat reittien päivityksessä 36
4.3 Askel kohti tuotantomaisuutta: Redis –replikoinnin käyttöönotto 38
4.4 B2CCacheManager muokkaus kahdennetun Redis:in käyttöön. 40
4.4.1 Uusi tietokantaindeksi ja B2C-datan siirto uuteen indeksiin 42
4.5 Yritysreitityksen suunnittelua 43
5 YRITYSLASKUJEN REITITYS 47
5.1 Projektin aikataulu muuttuu 47
5.2 B2CCachemanager -päivitys 48
5.3 Yritysreitityksen toteutus. 51
5.3.1 Yritysreittien latauksen toteutus 52
5.3.2 Yritysreititys 57
5.4 Integraatio NIP:n kanssa 59
5.5 Pakotetun reitityksen toteutus 61
5.6 NIP:n aikataulumuutos, uudet suunnitelmat 63
5.7 Palvelun tehokkuuden optimointi 66
5.8 Palvelun laadun testaus ja tuotteistus 70
6 PALVELUN TUOTTEISTUS 74
6.1 Syksy 2014: Projektin aikataulu muuttuu 74
6.2 Netposti-reititys 76
7 PALVELU TUOTANTOKÄYTÖSSÄ 78
7.1 Käyttöönotto 78
7.2 Palvelukatkot 79
7.3 Sovellusvirheet 80
7.4 Palvelun tuotantoympäristön lyhyt kuvaus 81
8 YHTEENVETO 84
LÄHTEET 86
LIITTEET

SYMBOLI- JA LYHENNELUETTELO

EDI	Electronic Data Interchange
A2A	Application to Application (yrityksen nykyinen laskunvälitysalusta)
SOAP	Simple Object Access Protocol
B2C	Business to Consumers (reitityksen yhteydessä: Yritykseltä kuluttajalle laskutus/reititys)
B2B	Business to Business (reitityksen yhteydessä: Yritysten välinen laskutus/reititys)
REST	Representational State Transfer: Arkkitehtuurimalli HTTP-rajapintojen kehittämiseen
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
POC	Proof of Concept
GIL	Global Interpreter Lock
RHEL	RedHast Enterprise Linux
NIP	New Invoicing Platform (tuotekehitysprojekti ja tuotenimi)
HA	High Availability
FKL	Finanssialan Keskusliitto
SI	SenderInformation (Lähettäjäilmoitus, määrittelijä FKL)
RI	ReceiverInformation (Vastaanottajailmoitus, määrittelijä FKL)
GW	Gateway (Yhdysliikennepiste A2A-alustasta toiselle operaattorille tai pankille)
URL	Uniform Resource Locator
WWW	World Wide Web
WSGI	Web Server GateWay Interface
CGI	Common Gateway Interface
FastCGI	Fast Common Gateway Interface
MVC	Model-View-Controller
TCP/IP	Transmission Control Protocol / Internet Protocol
SSD	Solid State Drive
OVT	Organisaatioiden välinen tiedonsiirto
SOAP	Simple Object Access Protocol
ICT	Information and Communications technology
I/O	Input/Output
CPU	Central Processing Unit
NMON	Nigel's performance Monitor for Linux on POWER
RAID**X**	Redundant Array Of Independent Disks (X = RAID -tason numero)
RPM	Revolutions Per Minute

SCSI	Small Computer System Interface
SAS	Serial Attached SCSI
RLU	Route Look up (Vanha reititysjärjestelmä)
LMC	Local Message Carrier (RLU järjestelmän operaattori määritelmä)
SQL	Structured Query Language
UCS-**X**	**X**-byte Universal Character Set (työssä esiintyy UCS-2 ja UCS-4)
EBA	Electronic Bill Account
BIC	Bank Identifier Code
IBAN	International Bank Account Number
AIX	Advanced Interactive eXecutive -käyttöjärjestelmä
SDU	Software Development Unit
UML	Unified Modeling Language
ISO-8869-1	Latin alphabet no. 1
UTF-8	8-bit Unicode Transformation Format
MVP	Minimum Viable Product
NUMA	Non-Uniform Memory Access
HT	Hyper-Threading
ECC	Error-Correcting Code
DDR**X**	Doube Data Rate (X = versionumero, DDR on keskusmuistimääritelmä)
TTL	Time-To-Live
IM	Instant Messaging
IBM	International Business Machines
SAN	Storage Area Network: Suuren kapasiteetin ja nopeuden levypalveluille pyhitetty verkko
NAS	Network-Attached Storage: Verkkoon kytketty levypalvelu
HDD	HardDisk Drive
TIEKE	Tietoyhteiskunnan kehittämiskeskus ry

1. JOHDANTO

Projektin käynnistyessä elokuussa 2013 yrityksessä oli jo aloitettu kartoitus vanhan laskunvälitysalustan uusimisesta nykyaikaisempaan ja paremmin skaalautuvaan järjestelmään. Nykyinen laskunvälitysalusta perustuu yli 20 vuotta vanhaan EdiServer 3.1 tuotteeseen, johon on ostettu lisenssi omien muutosten tekemiseen. Käytännössä siis yrityksellä on EdiServer 3.1 lähdekoodit, dokumentaatiot, kääntäjäparametrit ynnä muut tuotteen kehittämiseksi. Nykyinen versio EdiServer tuotteesta on myynnissä tuotenimellä Trade Express, jota tuotteen omistaja on kehittänyt omaan haluamaansa suuntaan samanaikaisesti kun yrityksemme on itse kehittänyt omaa versiotaan omiin tarpeisiinsa.

Vanha EdiServer on alun perin suunniteltu EDI (Electronic Data Interchange) -sanomien välitykseen. EdiServer:n ympärille on rakennettu useita järjestelmiä, jotta sen avulla voidaan välittää nykyaikaisia sähköisiä laskuja yrityksille ja kuluttajille. Järjestelmän ollessa kytköksissä moniin muihin järjestelmiin, se on saanut kutsumanimen A2A (Application-to-Application) yrityksemme sisällä.

EdiServer 3.1 on suunnitteluideologialtaan keskustietokoneiden (Main Frame) aikakaudelta, eikä sen toimintojen hajauttaminen useille koneille ole kovin nopeaa tai helppoa eli kustannustehokasta. Itse sovelluksen suunnitteluperiaate on myös vaikuttanut ympäröivien palveluiden arkkitehtuuriin. Uuden järjestelmän arkkitehtuuri on niin erilainen, että ympäröiviä järjestelmiä joudutaan muuttamaan.

Olemassa olevan reitityspalvelun (Route Look Up) nopeus ei riitä uuden alustan mukana määritellyille tehokkuusvaatimuksille, lisäksi uudessa alustassa jokainen sanoma on tarkoitus käsitellä atomisesti eli yksittäin, kun taas vanha reitityspalvelu on suunniteltu eräajo -tyyppiseen käyttötapaukseen. Tästä johtuen aloitimme myös reititysvälimuistipalvelun kehittämisen, jonka on tarkoitus pystyä vastaamaan yksittäisiin kyselyihin vaaditulla nopeudella.

1.1 Tavoitteet ja sisältö

Tässä kappaleessa kuvaillaan lyhyesti reititysvälimuistiprojektin tavoitteet ja vaatimukset, sekä itse diplomityön kirjallisen osuuden sisältö. Lähtökohtaisesti sovelluksen vaatimusmäärittely oli hyvin ylimalkainen, osittain johtuen siitä, että suurin osa RLU (Route Look Up) –osaajista oli ennen projektin aloitusta poistunut talosta, eikä tuotteella ollut varsinaista tuotehallintaa eikä tuotteen omistajaa.

Vaatimukset

- **Nopeus**: 1400 vastattua kyselyä sekunnissa. (1400 reittiä / sekunti)
    - Vertailun vuoksi: RLU -reitityspalvelu pystyi noin 1000-1200 reittiin / minuutti eräajona eräkoon ollessa 2000 kyselyä per SOAP –sanoma. (Simple Object Access Protocol)
- **Skaalautuvuus**: Palvelua pitää pystyä skaalaamaan yksinkertaisesti lisäämällä uusia palvelimia palvelun käyttöön.
- **Varmuus**: Palvelun tulee olla vähintään kahdennettu.
- **Ominaisuudet**: Palvelun tulee toteuttaa RLU -palvelun tarjoamat ominaisuudet. B2C- (Business to Consumer) ja B2B-reititys (Business to Business) sekä molemmille tapauksille lähettäjän pakottama reititys (forced routing).
- **Palvelurajapinta**: REST (Representational State Transfer) HTTP (Hypertext Transfer Protocol), sanoman sisällön muoto JSON (JavaScript Object Notation)
- **Ylläpidettävyys**: Palvelussa tapahtuneet virheet tai reitityspäätökset on oltava todettavissa lokitiedostoista.
- **Ohjelmointikieli**: Ei määritelty. POC –toteutukseen (Proof of Concept) valittiin Python 2.7.
- Kyselyt käsiteltävä yksittäisinä, ei eräajona.

Määritykset tai rajoitteet

- **Tietovarasto**: Esimääritelty arkkitehdin toimesta. Redis [1]
- **Palvelinkäyttöjärjestelmä**: Esimääritelty RHEL 6 -64bit (Redhat Enterprise Linux).
- **Kuormantasaaja**: F5BigIP [2]. Yrityksen yleisesti käyttämä kuormantasaaja.
- **Palvelimet**: Sovelluskehittäjän määriteltävissä.

1.2 Ongelman esittely, tavoitteet ja rajaukset

Reititysvälimuistiprojektin lähtökohtana oli toteuttaa olemassa olevaa reitityspalvelua hyödyntävä välimuistina toimiva palvelu joka olisi riittävän nopea sekä sopisi muuttuneeseen käyttötapaukseen. Projekti on toteutuksellisesti yhden miehen sovelluskehitysprojekti, jota löyhästi ohjastaa NIP (New Invoicing Platform) –projektin tuotehallinta sekä projektipäällikkö.

Edeltävä RLU -palvelu koostui seuraavista alipalveluista tai komponenteista, kaikkia näitä ei ole tarkoitus uusia reititysvälimuistiprojektissa vaan keskitymme yksinomaan reitityspalvelun välimuistittamiseen.

RLU -palvelukokonaisuus rakentuu seuraavista palveluista ja komponenteista.

- **IBM DB2 (International Business Machines Database 2)**: Palvelun tietokanta, jota kaikki komponentit ja palvelut käyttävät tietovarastonaan.
- **WebUI**: WWW-käyttöliittymä palveluun. Sisältäen asiakas- sekä ylläpitokäyttöliittymät, käyttöliittymätyypin valinta tapahtuu käyttäjän oikeusluokan mukaan. WebUI:n avulla asiakas voi hallita omia kuluttaja reittejään, sekä luoda vain omiin asiakasnumeroihin perustuvia yritysreittejä.
- **RI-import**: Vastaanottajaviestien käsittely. Palvelun komponentti joka lukee Suomen pankkiverkosta saapuvia RI-viestejä (ReceiverInformation eli kuluttajien vastaanottosanomia) ja tekee niiden perustella reititystietokantaan muutoksia.
- **Tieke-import**: Palvelu joka noutaa Suomen julkisen B2B –osoitteiston Tieke:n (Tietoyhteiskunnan kehittämiskeskus ry) tietokannasta ja tekee vaaditut lisäykset, päivitykset ja poistot reititystietokantaan. Todellisuudessa projektin alkaessa Tieke-import teki vain lisäykset; automaattiset päivitykset ja poistot implementoitiin projektin aikana.
- **RLU-Routing**: Itse reitityspalvelu johon A2A lähettää reitityspyynnöt.

Työn rajaukset: Reititysvälimuistiprojektin aihe keskittyy listan viimeisen komponentin toiminnallisten ominaisuuksien toistamiseen välimuistipalvelun avulla, siten että reittejä voidaan kysellä yksittäin riittävän nopeasti. Muiden komponenttien tekemät muutokset tulee päivittää reititysvälimuistin tietovarastoon.

1.3 Työn rakenne

Diplomityön luvussa 2 käsitellään ohuesti työhön ja projektiin liittyvää teoriaa ja esitellään valittuja palvelin- ja sovelluskomponentteja sekä perustellaan miksi kyseiset komponentit on valittu. Lisäksi toisessa luvussa esitetään karkea suunnitelma toteutettavasta palvelusta.

Kolmannessa luvussa esitellään reititysvälimuistipalvelun prototyyppitoteutuksen rakentaminen ja siihen liittyvät ongelmat ja mahdolliset ratkaisut sekä vaaditut muutokset toimivan palvelun tuottamiseksi. Kolmannessa luvussa esitetään myös suorituskykymittaukset prototyyppitoteutuksella, joka ei sisällä kaikkea vaadittua liiketoimintalogiikkaa.

Neljännessä luvussa esitellään prototyypin muuttaminen tuotantokelpoista palvelua kohden ja keskitytään kuluttajareitityksen toteuttamiseen. Lisäksi neljännessä luvussa valmistaudutaan HA (High-Availability) palvelin ympäristön rakentamiseen.

Viidennessä luvussa esitellään yritysreitityksen ominaisuuksien toteutus ja palvelun integroiminen uuden laskujenvälitysalustan NIP kanssa sekä projektin aikataulumuutokset. Lisäksi viidennessä luvussa käsitellään liiketoimintalogiikalla varustetun palvelun tehokkuusmittauksia ja useita erilaisia tehokkuusoptimointeja.

Diplomityön kuudennessa luvussa esitellään viimeisten ominaisuuksien rakentaminen sekä sovelluskoodin ylläpidettävyyden parantaminen ja palvelun tuotteistaminen.

Seitsemännessä luvussa esitellään lopullisen palvelun käyttökokemukset ja haasteet tuotannossa sekä esitellään palvelun rakenne tuotantoympäristössä.

1.4 Käsitteiden esittely

Lyhyt esittely työhön liittyvistä käsitteistä ja termistöstä yrityksessä:

- **B2B:** Business To Business, reititystermi, yritysten välisen laskutuksen reititys
- **B2C:** Business To Consumer, reititystermi, yritykseltä kuluttajille lähetettävien laskujen reititys.
- **”Forced Routing”:** Lähettäjän pakottama reititys, lähettäjä reitittää, välittäjän (operaattori) järjestelmä validoi reitin.
- **SI/RI:** Pankki verkon SenderInformation (Lähettäjäilmoitus) / ReceiverInformation (Vastaanottajailmoitus) viestit, viesteillä rekisteröidään lähettäjiä (yritykset) ja vastaanottajia (kuluttajat) pankkiverkkoon.
- **GW / Gateway:** Yrityksen sisäinen yhdysliikennepiste toiselle operaattorille tai pankille.
- **Välittäjäpankki:** Pankki joka tarjoaa välittäjäpankkipalvelua. Välittäjäpankki välittää viestit pankkiverkossa kohdepankille. Lähettäjä voi siis lähettää kaiken materiaalin yhteen pankkiin, jonka tehtävänä on huolehtia materiaalin päätyminen kohdepankkiin.
- **Netposti:** Posti konsernin elektroninen postipalvelu.
- **A2A:** Application-to-Application –kutsumaniminen vanha sanomien- ja laskujenvälitysalusta.
- **NIP:** New Invoicing Platform, rakenteilla oleva uusi laskujenvälitysalusta.

2. TYÖN SUUNNITTELU JA TYÖKALUJEN VALINTA

Reititysvälimuistin POC-vaiheen (Proof of Concept) toteutuskieleksi valittiin Python 2.7, koska kieltä on entuudestaan käytetty yrityksessä ja sen käytöstä on positiivisia kokemuksia. Python mahdollistaa melko nopean kehityksen, koska käyttöönottovaihe on melko kevyt ja Pythonille löytyy monia hyviä avoimen lähdekoodin kirjastoja ja muita ilmaisia työkaluja. Kevyellä käyttöönottovaiheella tarkoitetaan muun muassa sitä, että yksinkertaisia ohjelmia on nopea kirjoittaa ajovalmiuteen asti, vertailun vuoksi esimerkiksi Java –kieltä käytettäessä ajettavan ohjelman tekeminen vaatii hieman enemmän tekemistä, jonka lisäksi luokkien ja olioiden luonti Javassa on koodillisesti raskaampaa.

Pythonin huonona puolena voitaneen pitää sen hitaampaa suoritusta esimerkiksi verrattuna Javaan, joka johtuu ohjelmointikielten ja niiden kääntäjien ja ajotulkkien erilaisesta toteutusideologiasta. Java kieltä käytettäessä ohjelmoijan tulee itse huolehtia esimerkiksi siitä että usean säikeen ohjelmassa vain yksi säie pääsee käsittelemään muuttujaa kerrallaan, eli muuttujat on itse lukittava tai suojattava käsittelyn aikana, Python kielessä kooditulkki tekee saman automaattisesti sisäänrakennetun GIL (global interpreter lock) ominaisuuden ansiosta. [3]

Yrityksen aikaisemmissa sovelluskehitysprojekteissa olemme sisäisessä testauksessa huomanneet GIL –ominaisuuden hidastavan suorituskykyä verrattuna Java toteutukseen, jossa muuttujien suojaus hoidetaan itse.

2.1 Web-rajapinnan vaatimukset

Projektin aloitusvaiheessa kuormantasaaja, käyttöjärjestelmät ja tietovarasto määriteltiin arkkitehdin taholta. Ennen minkäänlaista testausta itse valittuun tietovarastoon liittyen aloitettiin työ tutkimalla erilaisia REST –rajapintakuvausta noudattavia ”WebFrameWork” –toteutuksia jotka olisivat hyödynnettävissä Python 2.7 –ohjelmointikielellä.

Palvelulle vaaditun korkean nopeuden vuoksi päätettiin etsiä mahdollisimman keveitä web-sovellusalustoja, koska emme oikeastaan tarvinneet niiltä kuin kahta ominaisuutta.

Ensinnäkin alustan tuli toimia WebService –ideologian mukaan niin sanottuna ”tilattomana verkkopalveluna”, eli ennen kyselyn lähetystä, kyselyn käsittelyn aikana ja vastauksen antohetkien välillä verkkopalvelulla ei ole minkäänlaista tilaa, jota sisään- tai ulostulodata muuttaisi. Toisin sanoen palvelulle ei ollut tarkoitus voida määritellä mitään erityisiä toimintatiloja tai sen sisäistä tilaa ei ollut tarkoitus pystyä muokkaamaan sisään annettavan datan avulla.

Toinen tärkeä ominaisuus oli REST –rajapintakuvauksen mahdollistama HTTP URL (Uniform Resource Identifier) parametrien tai itse URL:n muutosten avulla eri kyselyiden tai käytettävän funktion muuttaminen. Oleellista REST –rajapintakuvausta tukevissa web-sovellusalustoissa on uusien URL:n tai parametrien sekä niiden hyödyntämisen nopea implementoiminen. Meille selvästi oleellisinta tulisi olemaan uusien URL –polkujen hyödyntäminen, koska sisään tuleva reitityskysely toimitettaisiin JSON –formaatissa palvelulle ja itse kyselyn data sisältäisi kaiken oleellisen tiedon. Yksinkertaisten toiminnallisten vaatimusten lisäksi valitun web-sovellusrajapinnan tuli olla helppokäyttöinen tai ainakin nopeasti käytettävissä oleva, eli käyttöönotoltaan yksinkertainen.

2.2 Web-palvelin ja REST Web-sovellusalustat Pythonille

POC –toteutukseen (Proof of Concept) päätettiin valita yrityksessä tunnettu WWW-palvelinsovellus (World Wide Web), jotta mahdolliset ongelmat saataisiin minimoitua. Yrityksessämme on käytetty paljon Apache2 Web-palvelinta, itselläkin oli ylläpitokokemusta edellisen ”Application Specialist” –roolin (03/2009 – 09/2012) vuoksi kyseisestä palvelimesta paljon ja uskoin Apache2:n osoittautuvan aika ongelmattomaksi. Lisäksi Apache2 on saatavilla lähes jokaisen Linux –käyttöjärjestelmän paketinhallinnasta, joten asennus olisi helppoa ja nopeaa.

Edellä mainittujen vaatimusten valossa päädyttiin tutustumaan seuraaviin web-sovellusrajapintoihin sovelluskehittäjäkavereiden suosituksesta ja Google –hakujen tulosten vuoksi. Muun muassa Django:a suositeltiin huomattavan usein.

- Django [4,5]
- CherryPy [5]
- Flask [4,6]
- Pyramid [4]

Alkuun rajattiin pois CherryPy, koska se on web-sovellusalustan lisäksi myös minimalistinen web-palvelin. CherryPy –sovelluksia voi toki ajaa Apache2:n WSGI- (Web Server GateWay Interface) tai FastCGI (Fast Common Gateway Interface) -moduulin alla, mutta lähtökohtaisesti ei tuntunut kovin järkevältä ajaa web-palvelimen sisällä toista web-palvelinta. Hyvin nopeasti jo teknologioihin tutustumisen yhteydessä kiinnitettiin huomiota verkosta löytyviin opetusmateriaaleihin (myöhemmin nimellä tutoriaali) ja niiden kattavuuteen, sekä esiteltyjen web -sovellusalustojen ominaisuuksiin.

Heti tutustumisen yhteydessä kävi selväksi, että Django ja Pyramid sisältävät suuren määrän ominaisuuksia joita emme tarvitse, toisaalta ne ovat kohtuullisen vanhoja ja maturiteetin saavuttaneita web –sovellusalustoja. Toisaalta molemmissa tarvittiin heti työkaluja projektin pystyttämiseen (web–sovellusalustan omia työkaluja) josta jäi hieman raskas käyttöönottovaikutelma. Molemmat web-sovellusalustat on suunniteltu tukemaan MVC –ajattelua (Model-View-Controller), jota meidän käyttötapauksessa ei tarvittu.

Päädyin valitsemaan POC –toteutukseen web –sovellusalustaksi Flask:n, Flask on suunniteltu pieniä ja kevyitä palveluita varten, eikä sisällä ns. suuria ja hienoja ominaisuuksia. Toisaalta Flask kuitenkin tukee REST –arkkitehtuuria sekä mahdollistaa www-sapluunoiden (Template) käyttämisen. Lisäksi Flask pitää sisällään sovelluskehityskäyttöön tarkoitetun yksinkertaisen www-palvelimen, joka mahdollistaa koodin suoran muokkaamisen ja kokeilemisen. Esimerkiksi valitulla Apache2-palvelimella, täytyy aina kutsua restart- tai vähintään reload –metodia, jotta palvelin etsii ja kääntää koodiin tehdyt muutokset ja ottaa ne käyttöön uudelleen käynnistyksessä.

Lisäksi verkosta löytyi erittäin kattava ”tutoriaali” Flask:n käyttöön, kyseinen ”tutoriaali” pitää sisällään esimerkit mm. ”Templaateille”, WWW-lomakkeille, tietokannoille, käyttäjäprofiileille, käyttäjätunnuksille ja monelle muulle ominaisuudelle joita emme edes tulisi tarvitsemaan. Lyhyesti Flask siis valittiin keveytensä, kehitystyökalujen ja saatavilla olevien ilmaisten ”tutoriaalien” vuoksi. [8]

2.3 Tietovarasto: Redis

Kuten aiemmin mainittu palvelun tietovarastoksi määriteltiin arkkitehdin taholta Redis, joten muita tietovarastoja ei ole tutkittu. Pääpiirteissään Redis on keskusmuistin varassa toimiva tietovarasto (in-memory datastore). Kaikki luku- ja kirjoitusoperaatiot tehdään suoraan keskusmuistissa sijaitsevaan tietovarastoon. Asetuksista riippuen Redis ylläpitää säilyvänä tietovarastona (persistent storage) joko pakattua binäärimuotoista tilannekuvaa (SnapShot) tai tekstimuotoista ”Append-only-file” nimistä (myöhemmin AOF-tiedosto) transaktiolokia josta tietovaraston tila voidaan ladata ja palauttaa keskusmuistiin.

Ennen varsinaisen POC –toteutuksen aloittamista tuli toki kokeilla hieman Redis –palvelinta, tutustua sen ominaisuuksiin ja tehdä jonkinnäköinen testaus Redis:in nopeuteen liittyen. Käytännön pikatestaus tietovaraston riittävän nopeuden varmistamiseksi päätettiin tehdä Redis:in omalla testisovelluksella käyttäen vakiotestiä ilman optimointeja. Testit testisovelluksen (Asiakas, Client) ja Redis-palvelimen (Server) välillä suoritettiin käyttäen TCP/IP –protokollaa (Transmission Control Protocol / Internet Protocol) datan kulkiessa tietokoneella lokaalisti ns. ”loopback” –rajapinnan kautta. Sama testi suoritettiin myös käyttäen Unix Socket –rajapintaa.

Testi ajettiin seuraavanlaisella työasemalla. (Vain Redis:lle oleelliset tiedot).

- Prosessori: 4 Core Intel Xeon E5530 @ 2,4 Ghz with HyperThreading
- Muisti: 8 GB DDR3-ECC 1066 Mhz (3 muistikanavaa).

Redis:in testaamiselle oleelliset informaatiot koneen osilta käytännössä ovat vain suoritin ja keskusmuisti. Testin aikana Redis ei suorita yhtään levyoperaatiota tallennusmediaa vasten, joten nopeus on riippuvainen vain muistikaistasta ja suoritintehosta. Testisovellus

kytkee Redis:in AOF -tiedoston tai ”SnapShot” tallennuksen pois käytöstä testin ajaksi, koska muuten käyttäjän tekemät asetukset asetustiedostoon voisivat hidastaa testiä merkittävästi. Esimerkiksi AOF –tiedostolle voidaan pakottaa kirjoituspuskurin tyhjennys (flush) jokaisen muutoksen jälkeen. On toki ymmärrettävää etteivät kiintolevyt tai ”SSD-massamuistit” (Solid State Disk) kykene samaan nopeuteen kuin keskusmuisti. [9]

Tarkemmat testin tulokset löytyvät liitteistä (liite 1), alla olevasta taulukosta löytyvät tulokset reititysvälimuistin eniten käyttämistä operaatioista GET ja SET.

Redis-benchmark –sovelluksen komennot.

TCP: ./redis-benchmark -q -h 127.0.0.1 -p 50005 -n 100000 -c 50

Unix Socket: ./redis-benchmark -q -n 100000 -c 50 -s /tmp/redis_50005.sock

Oleelliset parametrit:

- n : Pyyntöjen kokonaismäärä
- c : Samanaikaisten rinnakkaisten kutsujen määrä

Operaatio	**TCP**	**Unix Socket**
SET	89686.10 pyyntöä / sekunti	202020.20 pyyntöä / sekunti
GET	128700.12 pyyntöä / sekunti	211416.50 pyyntöä / sekunti

Taulukko 1. Redis-benchmark tulokset.

Tulostaulukosta nähdään selvästi, että tehokkaammin Redis vastaa Unix Socketin kautta tuleviin pyyntöihin, koska TCP:n käyttö aiheuttaa käyttöjärjestelmälle enemmän ylimääräistä työtä TCP –protokollapinon vuoksi. POC –toteutusta lähdettäisiin siis rakentamaan Unix Socketin varaan, vaikka se tietynlaisia ominaisuuspuolen rajoitteita (kommunikaatio) aiheuttaisikin.

2.4 Palvelun suunnittelu karkeasti

Edellä mainitun pikaisen Redis:in suorituskykytestauksen johtopäätöksenä oli selvää, että sekä TCP/IP- että Unix Socket –menetelmällä itse Redis pystyy tarjoamaan riittävän nopean kirjoitus- ja lukunopeuden palvelun vaatimuksiin nähden. Palvelun vaatimus 1400 pyyntöä sekunnissa, Redis:in huonoin suorituskyky 82 000 pyyntöä sekunnissa.

Jo tässä vaiheessa oli kuitenkin oletettavaa, ettei lopullisesta palvelusta tulla saamaan yhdellä palvelimella kymmeniä tuhansia pyyntöjä sekunnissa suorituskyvyksi. Olemassa olevassa RLU -palvelussa on niin kutsuttua bisneslogiikkaa noin 4000 koodirivin edestä, lisäksi itse tietokantakirjastossa (talon sisäisessä kirjastossa, ei IBM (International Business Machines) DB2 (Database 2) kirjastossa) on koodia 2000 koodiriviä, jonka lisäksi itse tietokannan asetuksiin on tallennettu useita laukaisimia (Trigger), jotka reagoivat ennalta määrättyyn tapahtumaan X ja suorittavat operaatiojoukon Y. [10]

Tämän vuoksi päätin toteuttaa palvelun osien loogisen yhteistyön siten, että jokainen pystytettävä palvelin (myöhemmin ”palvelunoodi”) ohjelmistoineen on itsenäisesti (rajaus vain lukuoperaatioille) toimiva kokonaisuus. Rajaus lukuoperaatioihin johtuu Redis:in arkkitehtuurista. Mikäli halutaan olla varmoja kaikkien kirjoitettujen tietojen säilyvyydestä ja saatavuudesta kaikilla noodeilla, tulee ”orjanoodien” (Slave node) ohjata kirjoitukset ”isäntänoodiin” (Master node), koska tietojen toisinnus (Replication) toimii vain ”Master-to-Slave” –suuntaan. Lisäksi Redis suosittelee kirjoituksen estämisen ”orjanoodeilta”. Redis:in tietojen toisinnusstrategiat ovat käyttäjän määriteltävissä muuten, käyttäjä voi rakentaa muun muassa puu-, ympyrä-, jono-, tähti ja verkkotopologioita Redis -noodien välille. [11]

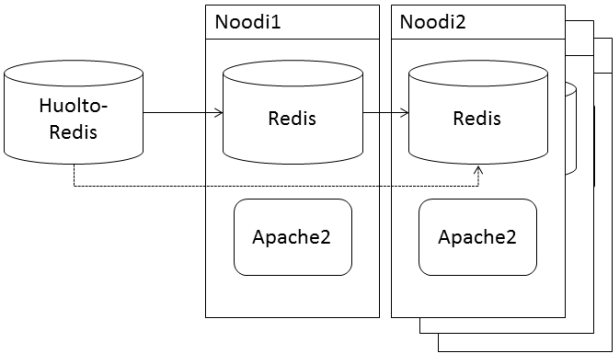
Tässä vaiheessa palvelun vedoksen (Draft) suunnittelua aloin pohtia mahdollisia ylläpidollisia ongelmia. Melkein 4 vuoden ”Application Specialist” –roolin tuoman ylläpitokokemuksen vuoksi ymmärsin välimuistityyppisessä palvelussa jo muutamia mahdollisia ongelmia. Ensinnäkin Redis-topologiassa voi sinänsä olla vain yksi todellinen isäntä –noodi. Toisekseen välimuistia päivittävän komponentin kaatuessa pitäisi joka tapauksessa pystyä palvelemaan asiakkaita (sisäinen asiakas; NIP (New Invoicing Platform)) vaikka vanhentuneella tiedolla sen sijaan, että koko palvelu pysähtyisi.

Tästä syystä päätin luoda vedoksen siten, että jokaisella palvelimella on autonomisesti toimivina seuraavat sovellukset tai komponentit.

- Redis -palvelin (lokaali)
- Apache2 –web-palvelin
- ”Routing-cache” -palvelu (reititysvälimuistin virallinen nimi)

Päivitykset Redis –palvelimen sisällään pitämään dataan ajettaisiin “huoltonoodin” kautta (myöhemmin nimetty uudelleen SuperMaster –noodiksi). Suurien tai riskialttiiden päivitys- tai huoltotoimenpiteiden ajaksi tietojen replikointi huoltonoodilta palvelunoodeille katkaistaisiin, ja mikäli päivitys- tai huolto-operaatio onnistuisi ilman kriittisen tason virheitä, tietojen replikointi käynnistettäisiin uudelleen ja tiedot päivittyisivät palvelua tarjoaville noodeille.

Alla summittainen kaaviokuva, jossa huomioitava ettei replikointistrategiaa ole vielä määritelty (katkoviiva osoittaa vaihtoehtoisen esimerkin).



Kuva 1. Palvelunoodien rakenne-ehdotus (Draft).

3 REITITYSVÄLIMUISTIN PROTOTYYPIN TOTEUTUS

Vaikka ennen varsinaisen ohjelmointityön aloitusta olin pohtinut palvelun skaalautuvuutta ja luotettavuutta, päätin sivuuttaa nämä asiat prototyyppitoteutuksen osilta. Asetin prototyyppitoteutuksen tavoitteeksi vain kuluttajareittien tallentamisen välimuistiin sellaisena kuin ne saataisiin ladattua RLU -palvelusta (tarkennus: RLU-Routing), koska minua kiinnosti enemmän nähdä käytännön nopeus reitityskyselyille käyttötapauksessa jossa kaikki palvelun vaatimat komponentit ovat käytössä. Tällöin nähtäisiin teoreettinen maksiminopeus tilanteessa jossa vain tietovaraston nopeuden testaamisen sijasta mukana olisivat www –palvelin, WSGI –moduuli ja itse Pythonilla toteutettu verkkopalvelu.

Tällä tavoin rakennettu POC –toteutus ei tulisi vastaamaan lopullista nopeutta tai tehokkuutta valmiiseen palveluun nähden, koska reititysvälimuistikin tulisi tarvitsemaan ainakin jonkin verran nk. bisneslogiikkaa. Minimissään suomalainen kuluttajalaskutus vaatisi niin sanotun välittäjäpankki –mallin (Intermediator Bank) vaatiman logiikan implementoimista koodiin, tai vaihtoehtoisesti välimuistiin pitäisi tallentaa jokaisesta kuluttajareitistä useampi versio jolloin tallennustilan tarve kasvaisi.

Menemättä turhiin yksityiskohtiin välittäjäpankki –malli toimii siten että lähettäjäasiakas (yritys) tekee välityssopimuksen valitsemansa pankin kanssa, jolloin välittäjäpankki hoitaa laskujen edelleen lähetyksen muihin pankkeihin mikäli kuluttaja ei ole kyseisen välittäjäpankin suora asiakas. Vaihtoehtoisesti lähettäjä voisi tehdä suoran sähköisen laskutuksen sopimuksen jokaisen Suomessa toimivan pankin kanssa joka tarjoaa kuluttajapalveluita. Käytännössä valtaosa nykyisistä asiakkaistamme hyödyntää välittäjäpankki –mallia, koska se on sopimusteknisesti helpompi toteuttaa.

3.1 I –vaihe: Läpilatausvälimuisti

Noin kahden viikon työpanoksella sain asennettua työasemaani Apache2 WWW–palvelimen, Redis-tietovaraston sekä luotua hyvin yksinkertaisen web-sovelluksen joka tuki vain yhtä käyttötapausta. Palvelu pystyi parsimaan sille lähetetystä HTTP -pyynnöstä JSON –formaatissa olevan reitityskyselyn ja edelleen lähettämään samaisen kyselyn SOAP -sanomassa RLU -reitityspalvelulle. RLU -palvelun antama vastaus parsittiin SOAP -sanomasta ja tallennettiin Redis -tietovarastoon.

Palvelu tuki vain reitityskyselyn oleellisimpia (käytännössä pakollisia) parametreja, joita ovat lähettäjäyrityksen OVT -tunniste (Organisaatioiden välinen tiedonsiirto), lähettäjäyrityksen sisäinen tunniste laskuaiheelle (Invoice subject) sekä kuluttajavastaanottajan asiakasnumero, nimi ja postiosoitetiedot. Huomioin jo tässä vaiheessa palveluiden toimintaperiaatteiden erilaisuuden ja loin keinotekoisen tavan rajoittaa yhtäaikaisten SOAP -kyselyiden (Simple Object Access Protocol) määrän halutulle tasolle, koska oletin RLU -palvelun kohtaavan tehokkuusongelmia yksittäisten kyselyiden käyttötapauksissa.

RLU -palvelu on optimoitu eräajotyyppisiä kyselyitä varten jossa yksi lähettäjä kysyy tuhansia reittejä yhdellä SOAP -kyselyllä, tällaisessa käyttötapauksessa pystytään optimoimaan tietokantakyselyitä, koska lähettäjä on sama jokaiselle haettavalle reitille. Kuluttajalaskutuksessa ”lähettäjä-vastaanottaja” -pari on aina ns. uniikki reittitieto, oikeastaan voitaisiin puhua ”lähettäjä-laskutusaihe-vastaanottaja” -monikosta (tuple), koska jokaisella laskutusaiheella on uniikki asiakasnumeroavaruus.

Reititysvälimuistin sovelluslogiikka toteutettiin hyvin yksinkertaiseksi tässä vaiheessa, kun sisään tulleesta kyselystä parsittiin lähettäjän ja vastaanottajan tiedot näistä luotiin hakuavain (myöhemmin ”Redis key”) ja avaimella haettiin tietoa Redis tietovarastosta. Redis:in palauttaessa vastauksen siitä luotiin reititysvastaus ja palautettiin se kyselyn lähettäjälle, mikäli Redis ei antanut vastausta muunnettiin JSON -sanoma RLU -palvelun SOAP -sanomaksi ja suoritettiin kysely RLU -palveluun. RLU -palvelun reittivastaus tallennettiin Redis-tietovarastoon sellaisenaan, ottamatta huomioon reitin tyyppiä tai muuta

sisällöllistä merkitystä joka oikeassa käyttötapauksessa tulisi huomioida.

Hyvänä esimerkkinä käyttötapauksen huomioimisesta toiminee niin sanottu ”fallback” –reitti. Käytännössä tämä tarkoittaa sitä, että mikäli kuluttajalle ei löydy mitään sähköistä reittiä pankkiin tai Netposti –palveluun, palautetaan viimeisenä vaihtoehtona ”paperikanavan” reitti (myöhemmin IPOST). Käytännössä tämä tarkoittaa laskun tulostamista paperimuotoon ja toimitusta kirjepostina. Ymmärrettävistä syistä IPOST –reittien tallentaminen ei ole viisasta koska se voidaan tarvittaessa luoda muutamalla koodirivillä, olettaen että reitityskyselystä löytyy vastaanottajan postiosoite ja vastaanottajan nimi. Lisäksi kuluttajien postiosoitteiden varastoimiseen liittyy lakiteknisiä ongelmia tai selvitettäviä asioita (henkilötietorekisteri).

3.1.1 Suorituskykytestaus ja ongelmat testiympäristössä

Jotta oikeaa suorituskykytestausta päästäisiin kokeilemaan oli välimuistiin ladattava testaukseen sopivaa reittimateriaalia. Tätä varten latasin RLU -palvelun Stage -ympäristön tietokannasta satunnaisesti valitulta lähettäjältä listan kuluttajien asiakasnumeroista joilla oli voimassa oleva sähköisen laskutuksen pankkireitti. Kuluttaja-asiakkaiden asiakasnumeroita poimin 20 000 kappaletta joiden pohjalta loin valmiit JSON –kyselyt reititysvälimuistipalvelua varten ja jätin asiakassovelluksen lataamaan reittejä RLU -palvelusta Stage –ympäristössä välimuistipalvelun lävitse.

Seuraavana päivänä asensin Windows –kannettavaani Apache Jmeter –testausohjelmiston ja asetin sinne 10 satunnaisesti valittua reitityskyselyä ja yritin ajaa tehokkuustestejä työasemallani pyörivään reititysvälimuistiin. Valitettavasti sovelluskehitystiimin toimistoverkon kytkin ja toimistoverkon reititys ei kestänyt testitilannetta, kyselyiden lähetysnopeuden ollessa noin 200-300 kyselyä / sekunti koko sovelluskehitystiimin verkko alkoi katkoa yhteyksiä konesaleihin, internetiin tai viereisille työasemille. Noin 20% lähetetyistä kyselyistä päätyi reititysvälimuisti –palveluun.

Tässä vaiheessa oli selvää että toimistoverkon varassa kyseistä testausta ei voitu suorittaa. Valitettavasti toimistolla ei varastoitu ylimääräisiä verkkolaitteita ja kaapeleita, joiden

avulla olisi voitu rakentaa muusta toimistoverkosta erotettu oma lähiverkko testauksessa käytettävien koneiden välille. Koska mitään välittömästi toteutettavaa ratkaisua ei ollut saatavilla, päätin ajaa testejä lokaalisti työasemallani siten, että sekä asiakas- että palvelusovellus ajettiin samalla koneella.

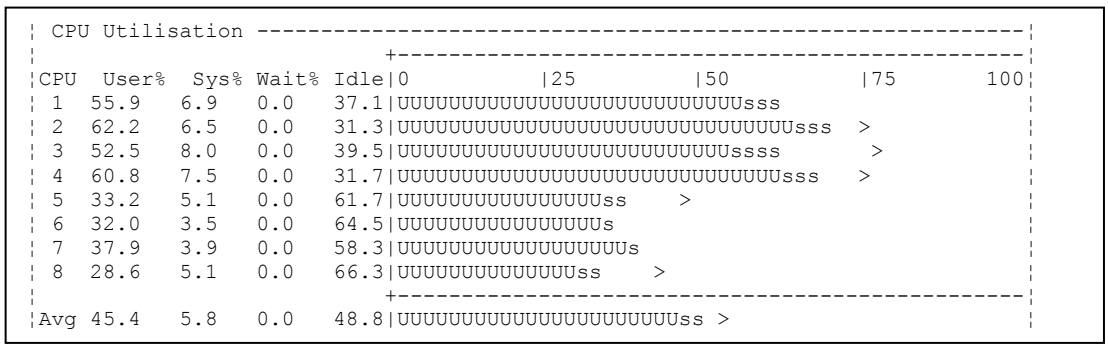
Työasemakoneeni on varustettu Intel Xeon 4-ydinsuorittimella joka tukee ”Hyper-Threading” –ominaisuutta (myöhemmin HT). Käytännössä käyttöjärjestelmä näkee suorittimella 8 loogista ydintä tietäen että niistä vain 4 ovat fyysisiä ytimiä. Menemättä hirveän tarkasti Intelin ”Hyper-Threading” –teknologian määritelmiin voidaan lyhyesti HT:n toimintaa kuvailla seuraavasti: Kun jollakin fyysisellä ytimellä suorituksessa oleva ohjelmakoodi pysähtyy johonkin operaatioon joka vaatii tiedon lataamista keskusmuistista tai kiintolevyltä, kyseinen ohjelmasäie jätetään taustalle odottamaan ja fyysinen suoritinydin siirtyy suorittamaan loogisella (ns. ”ylimääräisellä”) ytimellä jonottavaa sovellusta. Huomioitavaa tässä on että Intel puhuu säikeestä suorittimen näkökulmasta, eli toinen säie voi käyttöjärjestelmätasolla olla toinen prosessi, eikä vain saman prosessin toinen säie. [12]

Useimmissa tapauksissa HT -ominaisuutta tukeva suoritin pystyy suorittamaan toista sovellusta samalla suoritinytimellä kun alkuperäinen sovellus odottaa I/O-operaation tai muun odottavan operaation päättymistä. Useissa käyttötapauksissa koneen yleinen suorituskyky kasvaa, suorituskykyparannuksia osoittavia testitapauksia löytyy niin hyötyohjelma- kuin videopelikäyttötapauksissa. Olen myös itse jo aiemmin huomannut HT –ominaisuuden toimivan mm. Pifast –testiohjelman käyttötapauksessa hyvin, jota olen käyttänyt ylikellotetun tietokoneen toiminnan testaamiseen. [13] [14]

Koska testissä tultaisiin ajamaan sekä WWW –asiakassovellusta että palvelinsovellusta samalla tietokoneella päättelin, että HT –ominaisuudesta luultavasti olisi hyötyä, sillä molemmat ohjelmat joutuisivat käynnistyksen tai alustuksen jälkeen suorittamaan nk. estäviä I/O –operaatioita, joten ensimmäiseen testiajoon määrittelin 4 synkronisesti toimivaa Apache Jmeter –asiakassovellussäiettä jolloin periaatteessa WWW –palvelin tarvitsisi vähintään yhden ja enintään 4 palveluprosessia tai –säiettä suoritukseen. Tällöin suorittimen yliallokointi (CPU overcommit ratio) olisi maksimissaan 2:1, eli jokaista

fyysistä suoritinydintä kohden olisi ajossa enintään 2 prosessia tai säiettä. Yliallokointi on yleisesti tunnettu ominaisuus muun muassa palvelimien virtualisoinnissa, mutta sama periaate toimii fyysisen tietokoneen suorittimen ja suoritinydinten kohdalla HT –ominaisuudelle. [15]

Ensimmäisen testin tulokseksi saatiin noin 900 kyselyä sekunnissa käyttäen 4 lähettävää asiakassovellusta synkronisessa tilassa, eli sovellus ei lähetä uutta kyselyä ennen kuin edellisen HTTP -pyynnön vastaus on palautunut. Apache2 WWW -palvelin toimi vakioasetuksillaan, asetuksista säätämällä palveluprosessien ja prosessien sisäisten säikeiden määrää suoritustehoa olisi luultavasti voinut optimoida. Ensimmäisen testiajon aikana oli jo huomattavissa, ettei kaikkia tietokoneen resursseja käytetty hyväksi. Tämä näkyy myös alla olevasta NMON –sovelluksen (Nigel's performance Monitor) suoritingraafista, vain noin 50% saatavilla olevista resursseista on käytössä. Lisähuomiona voidaan sanoa että käyttötapauksen kuormitus jakautuu varsin hyvässä suhteessa käyttäjä- ja käyttöjärjestelmäresurssien kesken, sovelluksen aiheuttamasta kuormituksesta vain 11% aiheuttaa kuormaa käyttöjärjestelmälle (Graafissa ”system space” eli Sys%).



Kuva 2. NMON –suoritinkäytön kuvaaja.

Seuraavaksi päätin keskittyä sopivan suorittimen yliallokointisuhteen hakemiseen, jotta saisin mitattua suurimman saavutettavan nopeuden sekä ymmärtäisin paremmin kuinka sovellus skaalautuu. Alla olevassa taulukossa listataan asiakassovellussäikeiden määrä, suorittimen yliallokointisuhde, ajetun testitilanteen suoritinajan tyhjäkäyntiprosentti (Idle %) sekä saavutettu nopeus (vastausta sekunnissa). Suurin yliallokointisuhde on laskettu oletuksella että Apache2 WWW-palvelin käynnistää yhden palveluprosessin jokaista asiakasta kohden, näin ei toki aina tapahdu.

Asiakassovellusten lkm.	**Yliallokointi suhde (maksimi)**	**Tyhjäkäynti % (keskiarvo)**	**Suorituskyky (kyselyä / s)**
4	2	48,8	907
6	3	38,7	1387
8	4	23,8	1871
12	6	13,8	2188
16	8	5,4	2436
24	12	4,2	2513
32	16	1,6	2542
64	32	7,9*	2461
128	64	13,5*	2178

Taulukko 2. Reititysvälimuistin prototyypin suorituskykytestin tulokset.

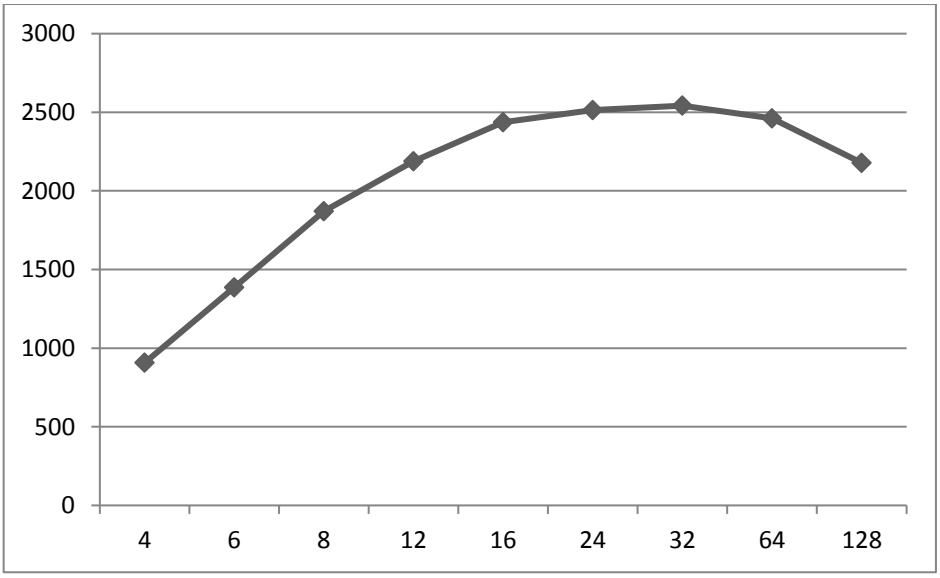
Huomioitavaa yllä olevassa taulukossa on mm. kaksi viimeistä riviä joissa tyhjäkäyntiprosentin perään on merkitty tähti (*). Näissä tapauksissa tyhjäkäyntiprosentin keskiarvo nousi rajusti, suoritingraafista oli tällöin havaittavissa huomattavan suuria I/O-operaatioiden odotusaikoja (IO-Wait %). Samalla huomasin työaseman RAID1 –levyosion (Redundant Array of Independent Disks) ylikuormittuvan palvelun lokitiedostojen kirjoittamisesta. Levyn I/O on testitapauksessa yksinomaan lokitietojen kirjoitusta, alustuksen ja käynnistyksen jälkeen asiakas- ja palvelinohjelmat eivät suorita lukuoperaatioita koska tarpeellinen data on saatavissa tietokoneen keskusmuistista.

Alla toinen NMON –sovelluksen suoritinkuvaaja sekä samaisen sovelluksen ”Disk I/O” –kuvaaja, josta tilanne (taulukko 2) on nähtävissä. Kuvaaja on otettu testaustilanteesta jossa lähettäviä asiakassovelluksia ajettiin 128 kappaletta yhtäaikaisesti.

```
 CPU Utilisation ---------------------------------------------------------
                        +-------------------------------------------------
CPU  User%  Sys% Wait% Idle|0          |25         |50          |75       100
 1  63.1  10.6   1.0   25.3|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUsssss
 2  66.5   9.0   7.5   17.0|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUsssWWW
 3  65.7   9.3  18.1    6.9|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUsssWWWWWWWWW   s
 4  64.8  10.1   0.0   25.1|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUsssss
 5  76.6   9.5   0.0   13.9|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUssss
 6  66.5   8.5   0.0   25.0|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUssss
 7  67.2   7.5   0.0   25.4|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUsss            s
 8  66.8   8.0   0.0   25.1|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUssss
                        +-------------------------------------------------
Avg 67.1   9.1   3.3   20.5|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUssssW
                        +-------------------------------------------------
 Disk I/O --/proc/diskstats----mostly in KB/s-----Warning:contains duplicates
DiskName Busy  Read WriteKB|0          |25         |50          |75       100
sda1       0%    0.0    0.0|>
sdb1      79%    0.0 9528.3|WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW>
sdc1      87%    0.0 9528.3|WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW>
md0        0%    0.0 9527.8|>disk busy not available
```

Kuva 3. Nmon –suoritinkäytön ja levy IO:n kuvaaja.

Tehokkuustestauksen tulosta on helpompi ymmärtää graafisessa muodossa. Alla kuvaaja jossa yksiköt; vaaka-akselilla (X-akseli) ”lähettävää asiakasta”, pystyakselilla (Y-akseli) ”kyselyä/sekunnissa”, käyrän kuvatessa saavutettua tulosta.



Kuva 4. Suorituskykytestin tulos 2d-kuvaajana.

3.1.2 Suorituskykytestauksen analysointi ja johtopäätökset

Yksinkertaisesti toteutetun suorituskykytestauksen tuloksista nähdään että asetettu suorituskykytavoite, 1400 vastattua kyselyä sekuntia kohden olisi saavutettavissa muutaman palvelinnoodin voimin. Testitilanne ei tietenkään täysin vastaa valmiin palvelun tilannetta, koska sovelluksesta puuttuu tässä vaiheessa kaikki liiketoimintalogiikka. Tässä käyttötapauksessa yksinkertainen suorituskykytestaus osoitti, että hyvin nopealla kyselyiden lähetystahdilla suoritin kohtaa melkoisen paljon nk. estäviä I/O-operaatioita (Blocking I/O-operation), koska suorittimen yliallokointi paransi kokonaissuorituskykyä. Testi ei kuitenkaan mahdollista yksittäisten kyselyiden läpimenoaikojen analysointia, koska palvelinpuolelle ei rakennettu yksittäisten kyselyiden läpimenoaikojen lokitietojen keräämistä.

Testitilanteesta piirretystä kuvaajasta (edellinen sivu, kuva 4) nähdään nopeuden kasvavan suhteellisen lineaarisesti 8 lähettävään prosessiin asti, tämän jälkeen prosessien lisäämisen ohella suorituskyky nousee hitaammin. 16 lähettävän prosessin jälkeen kasvu hidastuu merkittävästi ja taitekohta jonka jälkeen suorituskyky alkaa laskea on kuvaajan mukaan 32 lähettävän prosessin kohdalla. Taitekohta osuu jonnekin 32 ja 64 lähettävän prosessin välimaastoon, mutta nk. optimaalista kohtaa ei ruvettu tarkemmin hakemaan, tarkoitus oli saada lähinnä suuntaa antavia tuloksia.

Tässä kohden on kuitenkin huomioitava että testitapaus ei vastaa tuotannon käyttötapausta, koska tuotantoa varten liiketoimintalogiikka on oltava käytössä. Lisäksi ylläpitokelpoinen palvelu vaatisi huomattavasti enemmän lokitietojen keräämistä, jotta mahdollisia sovellusvirheitä tai virheellisiä reitityspäätöksiä voitaisiin tarvittaessa jälkikäteen tutkia.

Johtopäätöksenä tästä esitän seuraavan oletuksen konseptitoteutuksen ja tuotantovalmiin palvelun eroavaisuuksista. Oletettavasti täydellä liiketoimintalogiikalla varustetun palvelun yksittäisen kyselyn vaatima suorittimen laskentamäärä tulee olemaan suurempi kuin konseptitoteutuksessa, jonka vuoksi optimaalinen suorittimen yliallokointisuhde luultavasti tulee olemaan pienempi kuin konseptitoteutuksessa, johtuen siitä yksinkertaisesta seikasta että laskentaoperaatioiden määrä suhteessa estäviin I/O-operaatioihin kasvaa. Lisäksi

voidaan olettaa, että kokonaissuorituskyky samoilla resursseilla tulee laskemaan havaittavasti valmiissa sovelluksessa.

Testissä havaittu lokilevyjen ylikuormittuminen on mahdollinen skenaario myös tuotantokäytössä ja varmasti toteutuisi ainakin riittävillä suoritinresursseilla, en kuitenkaan oleta tästä seuraavan ongelmia oikeassa palvelinympäristössä. Oikeassa palvelinympäristössä käytetään havaittavasti nopeampia levyjä kuin työasemassani jonka kiintolevyt ovat tavallisia 7200 RPM (Revolutions per minute) kuluttajalevyjä, konesalin palvelimissa käytetään joko lokaaleja 15000 RPM SAS (Serial Attached SCSI (Small Computer System Interface)) kiintolevyjä. SAS-kiintolevyt ovat havaittavasti nopeampia sekä lukunopeuden että hakunopeuden osilta kuin kuluttajaluokan kiintolevyt.

Lisäksi tarpeen vaatiessa lokitiedostojen kirjoitus voidaan puskuroida jolloin I/O-operaatioiden määrä laskee, koska puskuriin kerättäisiin enemmän tietoa ennen jokaista kirjoitusoperaatiota. Lokitietojen kohdalla puskurointi on kuitenkin sinänsä vaarallista varsinkin tuotantokäytössä, että sovelluksen tai palvelimen kaatuessa kaikki puskurissa oleva tieto kadotetaan.

3.1.3 Vahinkohuomioita suorituskykytestauksesta

Suorituskykytestauksen valmistelussa sattuneesta inhimillisestä virheestä johtuen havaitsin läpilatausvälimuistisovelluksen käyttötapauksessa kriittisen luokan ongelmia sovelluksen kehityksen kannalta. Ensimmäisellä yrityksellä ladata reittejä välimuistipalvelun läpi RLU -palvelun stage –ympäristöstä en rajoittanut kyselynopeutta ja ajoin useampaa asiakassovellusta rinnakkain. Reittien latausyrityksen jälkeen havaitsin, että yli puolet kyselyistä oli saanut RLU –palvelusta virheviestejä. RLU -palvelun tietokantakirjasto ilmoitti, ettei vapaita tietokantayhteyksiä ollut (”Unable to allocate free database connector object from connection pool”), jonka vuoksi reittejä ei voitu antaa.

Läpilatausvälimuisti -tyyppinen ratkaisu ei siis tulisi sellaisenaan toimimaan, koska taustalla oleva palvelu hyytyisi välittömästi välimuistipalvelun kyselyiden käsittelynopeuteen. Taustalla olevaan palveluun ei olisi kovin helppoa rakentaa

minkäänlaista yksinkertaista jonotusjärjestelmää, koska sekin toimii HTTP –kutsuilla joille aina määritellään jonkinlainen aikakatkaisuarvo (Timeout value). Lisäksi tuotantopalvelussa on vaarallista jonouttaa tietoa palveluprosessin sisällä, varsinkin kun palvelulla ei ole mahdollisuuksia jononhallintaan, jolloin periaatteessa on mahdollista aiheuttaa tapaus jossa jono syö kaiken keskusmuistin palvelintietokoneelta. Käytännössä meidän palvelinympäristössä sovellus voisi varata kaiken käyttäjälle sallitun keskusmuistin, jota sallitaan palvelukäyttäjälle vähemmän kuin mitä käyttöjärjestelmällä on käytettävissä.

Yksi vaihtoehto olisi ottaa välimuistipalvelussa ja RLU -palvelussa käyttöön niin sanottu jonohallitsijasovellus. Yrityksessämme on valmiiksi käytössä IBM WebSphere –jononhallintapalvelu, lisäksi NIP (New Invoicing Platform) –projektissa käytetään RabbitMQ -jononhallintapalvelua. Päätimme kuitenkin arkkitehdin kanssa että emme lähde kehittämään RLU -palveluun tukea jononhallintapalvelun käytölle, koska sen käyttöönotto vanhassa A2A -järjestelmässä olisi erittäin haastavaa ja kasvaisi kokoluokassa omaksi projektikseen. Käytännössä olisi hyvin vaikeaa varmistaa että RLU WebService-asiakasohjelman päivitys tuhansille olemassa oleville asiakkaille toimisi ongelmitta, koska useissa asiakastoteutuksissa on asiakkaalle kustomoituja ominaisuuksia joita olisi mahdoton ottaa huomioon päivityksen yhteydessä, suuri osa näistä kustomoinneista ei ilmene asiakasdokumentaatiosta.

Lisäksi pohdittuamme välimuistin ylläpitoa läpilatausvälimuistin kohdalla olisi erittäin haastavaa ja kirjanpitotyöllisesti raskasta ylläpitää välimuistin oikeaa tilaa (Cache Coherency). Normaali kuluttajareitti muuttuu muun muassa seuraavissa tilanteissa, joista jokainen tulee vieläpä erillisestä palvelusta tai sovelluksesta.

- Asiakas muuttaa reititystietoja verkkopankista, pankki lähettää RI –viestin jonka RI-import –palvelu käsittelee ja päivittää tietokantaan.
- Asiakas on yhteydessä laskuttajaan ongelmatapauksessa, ja laskuttaja muuttaa asiakkaan reittiä RLU WebUI –palvelusta.
- Service Desk –osaston työntekijä muuttaa reititystietoja yksittäin tai eräajona komentorivityökaluilla A2A –ympäristöstä.
- 3. tason tuki (sovelluskehitys) suorittaa ylläpitotoimia suoraan tietokantaan.

Mainituissa käyttötapauksissa RI-import -palvelu sekä RLU WebUI -palvelu kirjaavat välimuistin ylläpidon suhteen riittävät tiedot alla olevan tietokantatoteutuksen historia- ja muutostauluihin (Change –table, History –table), mutta komentorivityökalut eivät aina. Sovelluskehityksen tukitoimien yhteydessä ei olisi mitään takeita että sovelluskehittäjä muistaisi kirjoittaa ylläpitotoimia suorittavaan koodiin mukaan historia- ja muutostiedot tietokantaan kirjoittavat ominaisuudet, usein tämän tyyppisten huoltotöiden taustalla on kriittinen sovellusvirhe joka on saatava korjattua välittömästi.

Vaihtoehtoisesti kaikkiin palveluihin voisi rakentaa jonkinlaisen ilmoitusviestimenettelyn, jolloin muutoksen tapahtuessa palvelu voisi lähettää HTTP –kyselyssä ilmoituksen tiedon päivittymisestä esimerkiksi välimuistipalvelun HTTP Rest –rajapintaan. Valitettavasti tämäkään ratkaisu ei auta huoltotoimenpiteisiin joissa tietokantaa muutetaan olemassa olevien ohjelmistojen ohi, ja tämän tyyppiset huoltotoimenpiteet olisivatkin hyvä sekä suunnitella, organisoida että dokumentoida huolellisesti ongelmien ja erityisesti tuotantokatkojen välttämiseksi.

3.2 II-vaihe: Palvelun suunnittelu POC –kokemusten pohjalta.

Ensimmäisen prototyyppitoteutuksen yhteydessä havaitun läpilatausvälimuistin ongelmista johtuen aloitettiin tutkimalla kuluttajareitityksen normaaliksi katsottavia päivitysviiveitä, yrityksen sisäisen päivityssyklin viiveitä sekä miettimällä oikean käyttötapauksen kannalta näiden merkitystä. Tässä vaiheessa konsultoin talossa vielä jäljellä olevaa RLU -osaajaa Ari Mannista joka on ollut suunnittelemassa, kehittämässä ja ylläpitämässä RLU palvelukokonaisuutta.

Keskustelussamme ilmeni että kuluttajien laskureittien päivityssykli pankeilta operaattoreille ei ole mitenkään reaaliaikainen, keskimäärin viive asiakkaan tekemästä muutoksesta verkkopankissa ja pankilta operaattorille viestityksessä oli noin yksi pankkipäivä. Käytännössä normaaliviikolla jolla ei ole erillisiä pyhäpäiviä ”tänään” tehdyt päivitykset tulisivat meille seuraavana päivänä, pois lukien perjantaina tehdyt muutokset jotka tulisivat käsittelyyn vasta maanantaina. Kaikki pankit eivät kuitenkaan lähetä RI –

sanomia (ReceiverInformation) yhdessä erässä vaan ne voivat tulla pitkin päivää isoissa tai pienissä erissä.

Yrityksen RI-import –sovellus, jolla RI-viestit käsitellään ja muutokset tehdään tietokantaan ajettiin tuolloin kahden tunnin välein, mutta aikaisemmin sitä oli ajettu vain kerran päivässä kunnes asiakaskunta oli toivonut tiheämpää päivityssykliä. Tästä johtuen tulin johtopäätökseen että ainakaan juuri nyt välimuistin päivityssyklin ei tarvitse olla kovin reaaliaikainen, mutta mielellään kuitenkin syklin toistojen aikaväli tulisi olla muokattavissa asetustiedostoista tai vastaavasti muualta siten ettei sovelluskoodiin tarvitsisi tehdä muutoksia.

Päivityksen pitäisi ainakin alussa osata ladata laskuttajan ja sen välisten kuluttajien reittitiedot kokonaisuudessaan yhdellä kertaa, tämä ei kuitenkaan olisi kovin tehokas tapa päivittää muutoksia koska erittäin isojen asiakkaiden tapauksissa kuluttajareittitietoja pitäisi ladata satojatuhansia kappaleita kerrallaan. Eli sinänsä päivitysmetodin pitäisi myös osata tehdä vain lisäyspäivityksiä (Incremental update), jolloin ladattaisiin vain muuttuneet (muutos reitissä, asiakastiedossa, tai reitin poisto) ja uudet reitit.

Onneksi RLU -palvelun tietokantarakenteessa oli RI -viestejä varten oma taulu (Table FI_RI) jossa ylläpidettiin jokaiselle asiakasnumerolle onnistuneesti kohdistetun viimeisen RI –viestin tiedot ja aikaleima. Samaisilla RI –viestin tiedoilla vanhempien RI –viestien tiedot olivat saatavilla historiataulusta (FI_RI_H). Toisin sanoen sekä uusien reittien että muuttuneiden reittien haku onnistuisi aikaleiman perusteella, mikäli välimuisti ylläpitäisi tietoa lähettäjäkohtaisesti viimeisen päivityksen ajankohdasta tai vaihtoehtoisesti viimeisen päivityksen suurimmasta muutostunnisteesta (ChangeID).

Muutostunnisteen käyttö sinänsä olisi hieman monimutkaisempaa, koska muutostunnistetta ei käytetä RI –viestien taulussa vaan itse reitin kertovassa taulussa, jolloin olisi pakko tutkia molempien taulujen sisältöä. Toisaalta reittitaulua ei tarvitsisi vielä POC -toteutuksessa tutkia, koska kohdepankki jonne lasku olisi lähetettävä löytyy jo RI –viestien tiedoista, ja pankin tunnisteella voidaan tehdä myöhemmin oikea reitityspäätös tarkistamalla paikallisten viestinvälittäjien (Local Message Carrier, myöhemmin LMC)

asetuksista ulosmenokanavan yhdysliikennepiste (Gateway, myöhemmin GW).

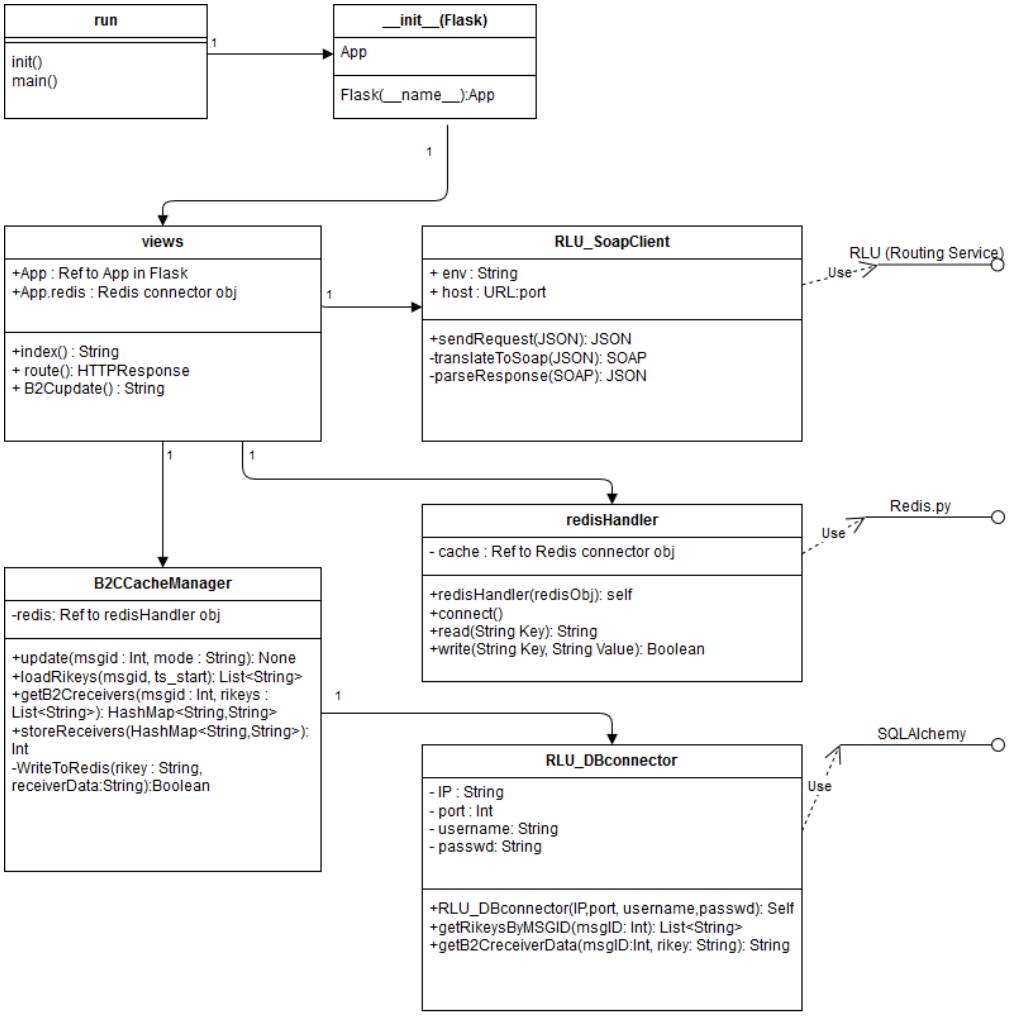
Selvästi siis pelkän reittitiedon lisäksi välimuistin olisi ylläpidettävä LMC –listauksen kopiota ja kytkettävä reittien päätepiste sisäiseen reititykseen LMC –listauksen avulla. Käytännössä LMC –listauksessa kerrotaan ulosmenevän operaattorin tunniste, kanava, kanavalle tarkoitetun laskun tietomuoto (viestiformaatti, esim Finvoice) ja sisäinen sähköisen laskun välitystili EBA (Electronic Billing Account).

Päästäkseni nopeasti alkuun oleellisen reititysongelman kanssa, päätin kovakoodata (Hard Coding) reititysvälimuistipalvelun sisäiseen tietorakenteeseen kuluttajalaskutukseen liittyvät LMC –tiedot. Käytännössä tämä tarkoitti 8 pankkia ja niille 2 kanavaa, jotka olivat suora laskuntoimituskanava ”f2c” (Finvoice to Consumer) ja jakajapankkikanava ”f2d” (Finvoice to Distributor).

3.2.1 Kuluttajareitityksen POC –toteutuksen suunnitelma

Päätin toteuttaa kuluttajareittien päivityksen toteuttavan koodin omana itsenäisenä moduulinaan ja kytkeä sen aluksi toimimaan itse WWW –sovelluksesta ja vaikka tämä ei tuotannossa olisi käyttökelpoinen ratkaisu niin kehitysvaiheessa se välttäisi. Mielestäni tämä oli nopein tapa päästä ongelman kanssa eteenpäin, Flask –WWW-sovellusalustan avulla olisi helppo lisätä uusi URL, jonka kautta moduuli käynnistettäisiin. Myöhemmin moduuli voitaisiin siirtää WWW –sovelluksen alta pois ja rakentaa sen ympärille moduulin käynnistävä ohjelma.

Tuleva väliaikainen sovellusrakenne olisi siis seuraavanlainen kuvattuna UML kaaviona.



Kuva 5. Suunniteltu sovelluksen väliaikainen rakenne UML kaaviona.

Yllä olevassa UML kaaviossa on kuvattu vain itse tehdyt luokat. Mikäli luokan nimessä on sulut ja toinen nimi, kerrotaan sillä mikä on jonkin toisen kirjaston tarjoama luokka josta toiminnallisuus on peritty ja sitä on muokattu. Esimerkiksi ”views” on oikeasti Flask – WWW-sovellusalustan luokka joka tarjoaa mm. mahdollisuudet määritellä virtuaalisia URL -polkuja ja mahdollistaa niille toiminnollisuuksien rakentamisen Python -koodilla. Esimerkin vuoksi, alla koodilistaus index() –function luomisesta, jonka URL:t olisivat ”http://localhost/” ja ”http://localhost/index”.

```
@app.route('/')
@app.route('/index')
def index():
    app.logger_debug.debug( 'views.index() : Received HTTP request')
    return "Hello World!"
```

Käytännössä siis annotaatiolla ”@app.route()” määritellään URL:n loppuosa ja alle kirjoitetaan itse toiminnallinen koodi, esimerkkinä siis jokaiselta WWW -sovellukselta oletettu ”index” sivu.

Luokka ”run” on WWW –sovelluksen käynnistysluokka, joka tekee vaadittavat alustukset sovelluksen käynnistyessä. Tuohon luokkaan on tarkoitus koodata mm. kovakoodattu kuluttajapankkien LMC -listaus. ”__views__” luokka on Flask:lle pakollinen initialisointiluokka, jossa käytännössä Flask –kirjastolle annetaan vain referenssi omaan sovelluskoodiin. Kokonaisuudessaan se sisältää 2 riviä koodia, Flask kirjaston import -lauseen ja sovelluksen viitteen sijoituksen.

Rajapintakuvaukset tietovarastojen kohdalla tarkoittavat käytettyä tietovarastokohtaista kirjastoa, esimerkiksi redis.py on Pythonilla luotu kirjasto Redis –palvelimen käyttöä varten, samoin kuin SQLAlchemy on kirjasto useiden erilaisten SQL (Strutured Query Language) –tietokantojen käyttöön. Toimiakseen SQLAlchemy vaatii ainakin IBM DB2:n kanssa tietokannan asiakassovelluksen sekä sen mukana toimitettavien kirjastojen asentamista koneelle.

”B2CCacheManager” –luokka tulisi toteuttamaan kuluttajareittien päivityksen, sillä olisi kaksi toimintatilaa jotka voitaisiin valita kutsussa annettuina parametreina. Tilat olisivat täysipäivitys eli lähettäjäkohtaisen reittitiedon haku kokonaan tai lisäyspäivitys joka hakisi edellisen päivitysajankohdan jälkeen tapahtuneet muutokset. Tässä vaiheessa tietojen latauksen käynnistys vaatisi vielä lähettäjäasiakkaan viestitunnisteen (myöhemmin MSGID) antamista parametrina päivitysmetodille.

Pseudokoodina toiminnallisuus toteutettaisiin seuraavasti:

```
update( msgid, moodd ) : - Käynnistää päivityksen halutussa moodissa
IF moodi == 'partial':
    IF timestamp
       - load_rikeys( msgid, ts_start ) : Ladataan RI –avaimet aikaleimasta
       eteenpäin.
    ELSE:
        - load_rikeys( msgid, None ): Ladataan kaikki RI -avaimet
ELSE:
   -   load_rikeys( msgid, None ): Ladataan kaikki RI -avaimet
FOREACH rikey
   -   getB2CreceiverData( msgid, rikey) : Ladataan jokaiselle RI-avaimelle
       reittitieto
   -   Tallennetaan jokainen reittitieto ("route") väliaikaisesti sovelluksen
       muistiin
FOREACH "route"
   -   storeB2Croute( routeHashMap ) : Tallennetaan jokaisen reitin avain-arvo
       pari Redis tietovarastoon
return
```

4 KULUTTAJAREITITYKSEN TOTEUTUS

Kappaleessa käsitellään kuluttajareitityksen tuotantoversion toteutusta ja toteutuksen aikana havaittuja ja korjattuja ongelmia. Palvelusovellukseen toteutetaan myös HA-ympäristötuki ja yritysreititystä suunnitellessa havaitaan alkuperäisen yritysreitityksen idean olevan mahdoton toteuttaa yksikertaisena läpilatausvälimuistia.

4.1 Tietokantakirjastot ja merkistöongelmat

Ennen varsinaisen kuluttajareitityksen ja kuluttajareitityslogiikan toteutusta täytyi tietysti ”B2CCacheManager” –komponentti saada valmiiksi. Prototyyppitoteutuksen luomiseen meni noin kolme viikkoa aikaa. Tuosta ajasta yli viikko meni hukkaan erilaisten tietokantakirjaston ja ajurin välisten ongelmien vuoksi, ongelmaksi osoittautui Python -pakettina asennettu IBM_DB -Python kirjasto, joka toimii SQLAclhemy -kirjaston ja IBM DB2 -ajurin välissä. Paketista asennettu kirjasto on esikäännetty kahden tavun ”unicode” merkistölle (2 byte unicode characters (UCS2)), työasemassa oleva Ubuntu Linux kuitenkin käyttää sisäisesti 4 tavun ”unicode” (UCS4) merkistöä. [16]

IBM_DB -kirjaston manuaalisen uudelleen kääntämisen (Compile) jälkeen esiintyi erittäin kummallinen ongelma. Jokaisen tietokantahaun yhteydessä ensimmäisestä palautetusta sarakkeesta saatiin vain kaksi ensimmäistä merkkiä Python -koodiin asti, ongelma toistui kaikilla testatuilla sarakkeiden tietotyypeillä. Esimerkiksi teksti- ja numerotietotyypit toimivat samalla tavalla väärin. Kahden päivän ongelmanselvitysyrityksen jälkeen luovutin, poistin koko Python -asennuksen palvelun alta ja asensin uudestaan, tällä kertaa esikääntäen IBM_DB -kirjaston ja asentaen itse käännetyn version manuaalisesti jonka jälkeen asensin muut kirjastot normaalisti paketinhallinnan kautta.

Varsinaisen ongelman lopullinen syy jäi selvittämättä, mutta uudelleenasennuksen jälkeen käytössä olevat kirjastot ja tietokanta tuntuivat toimivan oikein. Ongelmasta aiheutuneen huolestumisen vuoksi päätin tutkia asiaa RHEL -Linux jakelun suhteen, koska tuotantopalvelimissa käyttäisimme RHEL6 -Linuxia. Asensin Python 2.7:n ja kaikki palvelun kirjastot eräälle testausalustan RHEL6 -palvelimelle konesalissa ja kokeilin yhteyttä RLU -stage palvelun tietokantaan, samaa ongelmaa ei onneksi esiintynyt.

4.2 Kuluttajareittitiedon hallinta (B2CCacheManager) ja kuluttajareititys

Tietokannan käyttöön liittyvien ongelmien selvittämisen jälkeen täytyi suunnitella, miten tieto tallennettaisiin Redis -tietovarastoon ja kuinka reitityskyselystä saatavilla tiedoilla se löydettäisiin. RLU -palvelun toiminnasta ja vaatimuksista johtuen reitityspalvelun olisi saatava kyselyssä seuraavat tiedot: (Koko reitityskyselyn JSON -esimerkki löytyy liitteestä numero 2, jossa nämä tiedot esiintyvät.)

- Lähettäjän tunniste (SenderID)
- Lähettäjän tunnisteen tyyppi (SenderIDtype)
- Lähettäjän lähetysaihe (MessageSubject)
- Lähettäjän palvelutyyppi (ServiceCode : ”invoice” tai ”order”)
- Vastaanottajan tunniste (ReceiverID)
- Vastaanottajan tunnisteen tyyppi (ReceiverIDtype)

Kappaleessa 3.1 mainittiin että jokaisella lähettäjällä ja lähetysaiheeseen liittyvällä asiakaskunnalla on uniikki osoiteavaruus. Yllä olevasta listauksesta nähdään tiedot joiden avulla osoiteavaruus tunnistetaan. Tiedot SenderID, SenderIDtype, ServiceCode ja MessageSubject määrittävät mistä osoiteavaruudesta reittitietoa haetaan, näillä siis määritellään laskuja lähettävä yritys ja tuote jota he laskuttavat. Valitettavasti näillä tiedoilla ei suoraan tiedetä mitä osoitteita tietokannasta olisi haettava, jotta reitit voitaisiin ladata Redis tietovarastoon.

Finanssialan keskusliiton RI -viestimääritelmässä RI -sanoma sisältää kentän, johon jokainen operaattori määrittelee oman sisäisen tunnisteensa asiakasyritykselle ja kyseessä olevalle laskutusaiheelle. RLU -palvelussa ja sen tietokantarakenteessa tämä arvo on ”msgID”, valitettavasti reitityskyselyn tietosisällön ja ilmoitussanoman ”msgID” -arvon yhdistäminen ei onnistu tietovarastosta aivan yksinkertaisesti, kuten alla olevasta SQL – lauseesta nähdään. [17]

```
"select K.key, M.service, M.subjectcode, A.label
from Msg M
join Site S on (M.siteid = S.siteid)
join Key K on (K.rowid = S.siteid)
join Attr A on (K.attrid = A.attrid)
where M.msgid = ? and K.table = 'SITE';"
```

Oleellisinta reititysvälimuistipalvelun suorituskyvylle on tietenkin kyky vastata reitityskyselyihin nopeasti, siitä johtuen Redis -hakuavaimen tulisi koostua reitityskyselyssä välitettävistä tiedoista. Eli vaikka päivitysrutiini vaatii toimiakseen ”msgID” -arvon, niin tietovarastoon pitäisi kirjoittaa sen sijaan lähettäjäyrityksen ”SenderID”, ”SenderIDtype”, ”MessageSubject” ja ”ServiceCode”. Lisäksi avaimen tulisi sisältää laskun vastaanottajan asiakasnumero ja asiakasnumeron tyyppi.

Avaimen takana olevan Redis -arvon tulisi sisältää yrityksemme sisäiseen reititykseen liittyvien tietojen lisäksi pankkireititykseen liittyvät tiedot, käytännössä siis samat tiedot mitä RLU -palvelu palauttaa. Alla ko. tiedot listattuna.

- Yhdysliikennepisteen (GW) käyttämä sähköisten laskujen tili (EBA)
- Kohdeoperaattorin (pankki) BIC -tunniste (Business Identifier Code)
- Kohdekanavan laskudatan tietomuoto (message format)
- Vastaanottajan IBAN -tilinumero (International Bank Account Number)

Reititysvastaus tulisi olemaan tietomuodoltaan samanlainen kuin RLU -palvelun reititysvastaus, tieto vain esitettäisiin SOAP -sanoman sijasta JSON -sanomana. Alla esimerkki jonka tiedot eivät ole oikeita tietoja, esimerkki sisältää JSON -sanomasta vain yhden ”route” elementin sisällön. JSON sanomien sisällöt on kuvattu kokonaisuudessaan liittessä 2.

```
{
	"eba": "BELR990017587L",
	"operator": "NDEAFIHH",
	"format": "f2c",
	"eaddr": {
		"addr": "FI12340000123400",
		"operator": "NDEAFIHH",
		"format": "f2c",
		"envaddr": "FI12340000123400@NDEAFIHH/f2c"
	}
}
```

Lyhyesti yllä olevan reittitiedon käyttötapaus on seuraava: EBA -tilinarvo kertoo mille sisäiselle ”lähetystilille” lasku tulee tallentaa, operaattorikentän ja formaatin tiedot ovat alla olevassa EBS Repository -järjestelmässä (Electronic Billing System) sidottu tiliin mutta ne ovat vastauksessa näkyvissä ylläpitosyistä; Service Desk -osaston on helpompi

hoitaa tukitoimia kun reittitiedosta näkee suoraan kohdeoperaattorin. ”EADDR” -elementti pitää sisällään pankkiverkon käyttämät reititystiedot joilla lasku kohdennetaan oikealle vastaanottajalle.

Välittäjäpankkimallin (Intermediator bank) käyttötapauksessa ”EADDR” -elementti pysyy samanlaisena, mutta sisäisen reitityksen tiedoissa eba, operator ja format -kenttien arvo muuttuisi osoittamaan välittäjäpankin yhdysliikennepisteen tietoja. Tällöin siis lasku toimitettaisiin toiselle pankille, joka sitten hoitaa viimeisen toimituksen EADDR -elementin tietojen perusteella.

Halusin minimoida reitityspalvelun laskentakuormaa reitityskyselyn aikana, joten päätin tallentaa reititysvastauksen tiedot valmiiksi lasketussa muodossa. Määrittelin Redis avain-arvoparin kuluttajareititykselle seuraavasti, ja esimerkin arvojen nimet ovat muuten ns. itsestään selviä paitsi seuraavat lyhenteet fOP (Final Operator), fFmt (Final Format). Tilasyistä Value -kenttä on parametrisoitu osiin ”ir” (InternalRouting) ja ”er” (ExternalRouting).

```
Key =
"SenderID!SenderIDtype!SubjectCode!ServiceCode!RecvID!ReceiverIDtype"
Value = "ir||er"
, jossa:
ir = "eba=EBA!op=operator!fmt=format"
er = "addr=IBAN!op=fOP!fmt=fFmt!envaddr=IBAN@fOP/fFmt"
```

Reittitiedon tallentamisen lisäksi reititysvälimuistipalveluun piti luoda yksinkertainen reitityslogiikka. Tässä vaiheessa loin vedosmuotoisen logiikan ”views” -luokkaan.

Pseudokoodina logiikka oli seuraava:

```
Nouda avaimella arvo Redis:stä
IF arvo
      IF intermediator määritelty reitityskyselyssä
            Muokkaa sisäisen reitityksen EBA, Operator ja format
            tiedot vastaamaan välittäjäpankin tietoja
      ELSE
            Reititä Redis arvon tiedoilla
ELSE
      Muodosta ”fallback” –reitti, eli paperikanavan reitti.
```

Tässä yksinkertaisessa reitityslogiikassa ei oteta huomioon vielä muita käyttötapauksia kuin tavallinen sähköinen lasku. Suoraveloituksen korvanneen suoramaksun kohdalla välittäjäpankkimallin logiikka toimii samoin kuin sähköisellä laskulla, mutta suoramaksu vaatii sähköisen laskun kanavan lisäksi ilmoituskanavan. Yleisesti ilmoituskanava on paperiposti, mutta emoyhtiömme tarjotessa Netposti –palvelua tilanne on monimutkaisempi.

Kuluttaja voi lähettäjäkohtaisesti valita Netposti –palvelusta että haluaako hän materiaalin vain Netpostiin vai vaihtoehtoisesti paperina sekä kopion Netpostiin. Eli käyttötapauksessa suoramaksu ja kopio Netpostiin palvelu tulisi enimmillään palauttamaan 3 reittiä ja vähintään 2 reittiä, ja lisäksi palvelun täytyisi tarkistaa löytyykö sähköisen reitin lisäksi käyttökelpoinen ilmoituskanava. Mikäli käyttökelpoista ilmoituskanavaa ei olisi, tulisi palvelun palauttaa virhe. Tässä vaiheessa päätin jättää tuotantokelpoisen logiikkakoodin rakentamisen tulevaisuuden ongelmaksi.

Tuotantokelpoinen logiikka vaatisi myös pakotetun reitityksen käyttötapauksen toteutuksen, käytännössä pakotetun reitityksen tapauksessa laskuja lähettävä taho hoitaa reitityksen itse ja kertoo meidän palvelulle vain reitin avaintiedot, meillä palvelun on tarkoitus tarkistaa että annettu reittitieto sisältää olemassa olevat operaattorit ja että niiden kanavat ovat käytettävissä.

4.2.1 Testejä ja ongelmat reittien päivityksessä

Ennen varsinaisen reitityslogiikan toimintanopeuden testaamista piti palvelun tietovarastoon saada ladattua reittejä. Ensimmäinen ongelma tuli Apache2:n vakion aikakatkaisuarvon (timeout) kanssa, vakioasetuksilla HTTP –pyynnön vastausta odotetaan 15 sekuntia jonka jälkeen pyyntö hylätään epäonnistuneena ja palveluprosessi käynnistetään uudelleen. Prosessin uudelleenkäynnistys johtuu siitä, että mod_fcgid –moduuli (FastCGI -moduuli) ei tiedä jumiutuiko koko pyyntöä hoitava palveluprosessi vai onko suoritus kesken. Timeout -arvoa kasvattamalla sain ladattua pienemmän asiakkaan reititystiedot palveluun, mutta ensimmäisissä testeissä käytetyn isomman asiakkaan tietojen lataus olisi kestänyt huomattavan pitkään.

Valmistelin uuden testimateriaalin pienemmän asiakkaan reititystiedoille ja ajelin aikaisemmin esiteltyä suorituskykytestiä vastaavan testin, tällä kertaa tosin vain yksillä asetuksilla. Nyt samanaikaisesti pyyntöjä lähettäviä asiakasprosesseja oli 16. Edellisessä testitilanteessa samalla asetuksella tulos oli 2436 vastattua kyselyä sekunnissa, nyt tulos oli 1520 eli noin 62,4% edellisestä suorituskyvystä. Suurimmat suorituskykyä heikentävät, suoritinaikaa vaativat muutokset edelliseen läpilatausvälimuistin testitilanteeseen ovat mm.

- HTTP –kehyksen validointi
- JSON –viestin validointi sekä vaadittujen kenttien täytön validointi
- Uudet hakukyselyt Redis –tietovarastoon
- Erilaiset merkkijonojen kopiointi- ja muutosoperaatiot datan käsittelyssä
- Reitityslogiikka
- HTTP –standardin mukaisen vastauksen luominen ja palautus (edellinen versio palautti vain muotoilemattoman merkkijonon).
- Aikaisempaa kattavampi lokikirjaus (palveluloki + debugloki)

Vaikka suorituskykytestin tulos romahti paljon edellisestä testistä, ei asia minua sinänsä huolettanut. Ensimmäinen testitilanne ei millään muotoa vastannut oikeaa käyttötapausta ja testin tarkoituskin oli vain varmistaa, että valituilla teknologioilla haluttu suorituskyky olisi mahdollista saavuttaa. Edelleen oletan että tämänkään testin suorituskyky ei tule vastaamaan tuotantovalmiin palvelun suorituskykyä, koska koodiin olisi tulossa vielä hyvin paljon lisää validointia ja jonkun verran lisää myös liiketoimintalogiikkaa. Tällä hetkellä koodista puuttuu merkistöjen koodausten tarkastukset ja mahdollisesti tarvittavat uudelleenkoodaukset sekä paljon muita tarkistuksia ja ongelma- tai virhetilanteisiin varautumisia joita tuotannossa on pakko olla.

Vaikka suorituskyky romahtaisi vielä 50% tämän testin suorituskyvystä se olisi tuotantokäytön kannalta vieläkin hyväksyttävää, minimissäänkin palvelimia tultaisiin pystyttämään kaksi kappaletta. Olimme myös alustavasti käyneet läpi nykyisiä teknologiavalintoja ja keskustelleet kehitys- sekä stage –ympäristön pystyttämisestä. Eräältä projektilta oli jäänyt konesaliin kaksi testiympäristön palvelinta, joita ei ollut vielä poistettu käytöstä vaikka palvelimilla ajetut sovellukset olikin jo ajettu alas. Päätimme käyttää uudelleen nämä palvelimet jotta meidän ei tarvitsisi odotella konesalitoimittajalta

uusien palvelinten pystyttämistä.

Tällöin varmistui että stage –ympäristöön saataisiin fyysiset palvelimet, olimme jo alkuun pohtineet ettei käyttötapauksemme luultavasti ole kovin hyvä virtualisoiduilla palvelimilla. Jo edellisessä roolissani 2.- ja 3.-tason tuessa olin törmännyt tilanteeseen, jossa virtualisoitu web-palvelinklusterimme suorituskyky rajoittui alla olevan fyysisen koneen muistikaistan tukkeutumiseen. Konesalitoimintojen uudistuksen yhteydessä siirsimme kyseisten palvelimien palvelut fyysisille koneille ja ongelma poistui. On myös tutkimustietoa joka osoittaa saman ongelman mm. erilaisten tiedontallennusjärjestelmien kuormituksen kanssa virtualisoidussa ympäristössä. [18]

Stage –ympäristön palvelimet olivat seuraavanlaiset koneet joita konesalitoimittaja kutsui tuotenimellä ”Level 2 Power Blade”.

- 2 kpl 4-ydin Intel Xeon E5-2609 @ 2,4Ghz -suorittimia
- 32 Gigatavua muistia
- SAN-High kiintolevyt (toimittajan nopeimmat SAN -levyt)

Stage –ympäristöön siis tulisi kaksi palvelinta, joista kumpikin numeroiden valossa olisi noin 2 kertaa tehokkaampia kuin työasemakoneeni. Tosin Intel Xeon E5-2609 suoritin ei tue Hyper-Threading -ominaisuutta joka mahdollisesti vaikuttaisi suorituskykyyn heikentävästi verrattuna siihen, että myös palvelinten suorittimet tukisivat HT –ominaisuutta. Toisaalta konesalin palvelinten suorittimet tukivat 4 muistikaistaa työasemani 3 muistikaistan sijaan, konesalin raudassa muistikaistaa oli noin 33% enemmän (34,1 GB/s vs. 25,6GB/s).

4.3 Askel kohti tuotantomaisuutta: Redis –replikoinnin käyttöönotto.

Seuraavaksi haasteekseni otin palvelun prototyypin muokkaamisen tietovaraston osilta tuotantoa muistuttavaan asetelmaan. Työasemani toki rajoitti minut testaamaan ominaisuutta saman koneen piirissä, mutta saisin kuitenkin käytännön kokemuksia Redis –palvelinten synkronoinnin toiminnasta sekä mahdollisesti löytäisin sovelluksesta siihen liittyviä ongelmia joita en ole osannut ajatella aikaisemmin. Loin toiselle Redis –

instanssille asetustiedoston, käynnistysskriptit ja asetin sen edellisen Redis -asennuksen orjaksi (Slave node). Muutin sovelluksen asetuksia siten että se käyttäisi tietolähteenään uutta Redis –orjanoodia.

Käynnistyksen jälkeen orjanoodi otti automaattisesti yhteyttä isäntänoodiin ja aloitti tiedon synkronoinnin. 20 MB:n tietovaraston synkronointi tapahtui hyvin nopeasti lokaalin TCP/IP –yhteyden kautta. Oikeastaan liiankin nopeasti, koska en kerennyt ”leikkiä” Redis:llä kokeillakseni miten orjanoodi vastaa kesken synkronointioperaation. Niinpä generoin Redis:iin noin 500 MB satunnaisdataa kirjoittamalla satunnaisia avaimia ja arvoja tietovarastoon. Tyhjensin orjanoodin tietokannan ja käynnistin synkronoinnin uudestaan. Toistaessani ensimmäisen testin uudelleen, Redis vastasi lukuyritykseen kesken synkronoinnin ”BusyLoadingError” –poikkeuksella.

Tutkittuani Redis:in dokumentaatiota ja eri synkronointimetodeja ilmeni, että täyden tietokantalatauksen jälkeen uuden datan käyttöönottohetkellä Redis ei voi palvella asiakassovellusta ja tämä tilanne tulisi asiakassovelluksessa huomioida. Normaalisti Redis pyrkii tekemään päivitykset yksittäisinä atomisina operaatioina, mutta käynnistystilanteessa tai pitempien verkkokatkojen jälkeen Redis suorittaisi täyden tietokantalatauksen isäntänoodista. WWW –sovellukseen olisi siis rakennettava ominaisuus joka osaisi odottaa Redis:in palvelukyvyn palautumista, mikäli Redis ilmoittaisi ettei voi palvella. [11]

Tutkin Redis -palvelinten asetustiedostoja ja huomasin, että vakiona Redis:iin on asetettu ”sisäiseksi toimintataajuudeksi” 10hz, Redis pyrkii suorittamaan nk. taustaoperaatiota tuon tahdistuksen mukaan. Päätin toteuttaa palvelukyvyn palautumista odottavan logiikkakoodin pätkän puolta nopeammalla tahdilla. Rakensin perinteisen aikakatkaisulla varustetun silmukkarakenteen joka yrittää lukea Redis:stä (olematonta) avainta 20hz taajuudella. Mikäli vastaus on sama ”BusyLoadingError” –poikkeus, odottaa silmukka 0,05 sekuntia ja yrittää uudelleen kunnes suurin aikaraja (120 sekuntia) tulee vastaan tai Redis –kirjaston ”get()” metodi palauttaa arvon ”None”. Arvoksi palautuu ”None”, koska kyseisellä hakuavaimella ei löydy dataa. Tämän jälkeen sovellus jatkaa suoritusta kuten aikaisemmin.

Toistin alkuperäisen testitilanteen uudelleen, eli pakotin Redis –orjanoodin täyden synkronoinnin tilanteeseen ja lähetin palvelulle reitityskyselyn. Hieman alle 10 sekunnin odottelun jälkeen palvelu palautti onnistuneen vastauksen. Ominaisuudessa oli puutteena, ettei vastauksesta ilmennyt miksi sen käsittely kesti niin pitkään. Lisäsin merkinnän puuttuvasta vastauksesta tehtävälistaan, jotta myöhemmin muistettaisiin lisätä vastauksen lokitietoihin syy käsittelyn hitaudelle.

4.4 B2CCacheManager muokkaus kahdennetun Redis:in käyttöön.

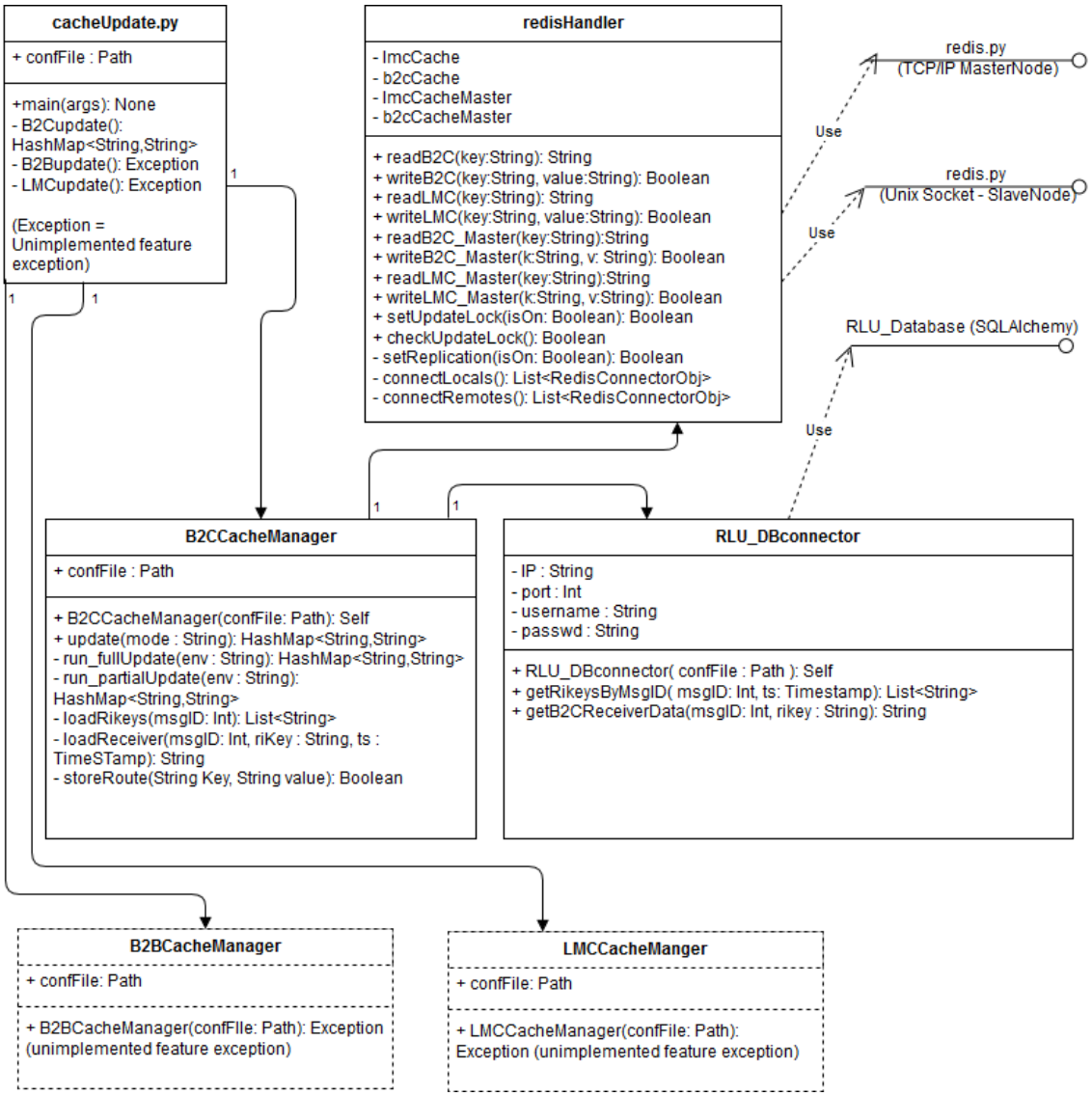
Kuluttajalaskutuksen osoitteiden päivitysmoduuli piti muokata tukemaan uutta asetelmaa, jossa ajossa oli kaksi Redis palvelinta. Ensimmäisenä lisäsin tuen kahden eri Redis -palvelimen käyttöön siten, että moduuli pystyisi lukemaan tietoa ’palvelun Redis noodista’ Unix Socket –rajapinnan kautta kuitenkin tehden kaikki muutosoperaatiot isäntänoodin kautta käyttäen TCP/IP -rajapintaa. Samalla lisäsin pikaisesti tietojen noutoon uuden ominaisuuden joka hakisi pankkien kuluttajakanavien LMC –tiedot (Local Message Carrier) Redis -tietovarastoon, jotta voisin poistaa kovakoodatun LMC -listauksen palvelusta.

Lisättyäni kahta erilaista dataa Redis:iin käyttäen samaa Redis:in ”tietokantaindeksiä” havaitsin, ettei kahden erilaisen tiedon noutaminen tai ylläpitäminen ollut kaikissa tapauksissa kovin tehokasta. Eritoten havaitsin ongelman listatessani satunnaisluotuja avaimia tietokannasta jotka olisi ollut tarkoitus poistaa. Olin laittanut jokaiselle satunnaisluodulle avaimelle etuliitteen ”RANDOM=” jolloin avaimen tunnistaminen olisi helppoa. Listatessani näitä avaimia ”redis-cli” –asiakassovelluksesta havaitsin koneen suoritinkäytön ”redis-server” –palvelinprosessilla nousevan hetkellisesti täyteen 100% kun avaimia pyydettiin listamaan komennolla ”KEYS RANDOM=*” , jossa *-merkki tarkoittaa sitä, että avain voi sisältää ko. merkin jälkeen mitä vain dataa. Lukiessani ”KEYS”-komennon dokumentaatiota havaitsin että tällaisten hakujen tekoa tulisi välttää tuotantokäytössä tehokkuussyistä. [19]

Tässä vaiheessa päätin eritellä tietueet käyttötapauksen tai tietotyypin mukaan omiin

tietokantaindekseihinsä Redis –tietovaraston sisällä. Redis:in tietokantaindeksi ei ole normaalin SQL –tietokannan mukainen rivien indeksointilistaus, vaan käytännössä se on oma tietosäiliö jolla on oma avain-arvoparisisältönsä, tieto on siis loogisesti erillään muista indekseistä. Tämä näkyy myös siinä, että Redis:in ”flushdb()” –komento vaikuttaa vain valittuun indeksiin. (Flushdb –komento tyhjentää koko indeksin kaikki avaimet ja arvot.)

Kuvassa 6 esitellään välimuistinhallintasovelluksen suunniteltu rakenne. Katkoviivoin merkityt luokat toteutettaisiin myöhemmin.



Kuva 6. Välimuistinhallintasovelluksen suunniteltu arkkitehtuuri.

4.4.1 Uusi tietokantaindeksi ja B2C-datan siirto uuteen indeksiin

Valitettavasti heti alkuun havaitsin, että ”redis-py” –kirjasto ei tukenut kuin yhtä tietokantaindeksiä jokaista yhteyttä kohden, joten muutin pikaisesti omaa Redis –luokkaani siten että jokaiselle tietokantaindeksille luotaisiin oma yhteytensä, lisäksi lisäsin luokkaan tuen isäntänoodin yhteyksille. Kirjastossa oli nyt siis kaksi erillistä Unix Socket –yhteyttä indekseihin 0 ja 1 sekä samaiset yhteydet käyttäen TCP/IP –protokollaa, jotta samaisella luokalla tarvittaessa voitaisiin ottaa yhteyttä myös isäntänoodiin. Toteutus oli hieman ruma, mutta tehokkuuden kannalta parempi kuin toteutus yhdellä yhteydellä jolla täytyisi kutsua disconnect()- ja connect() –kutsuja joka kerta kun olisi tarvetta käyttää toista tietokantaindeksiä.

Seuraavaksi poistin kaiken satunnaisluodun datan ja siirsin kaiken kuluttajareititysdatan indeksiin 1, jolloin indeksi 0 jäisi LMC –listaukselle. Samalla loin ominaisuuden pankkien LMC –tietojen lataamiselle ja tallensin nämä tiedot indeksiin 0. Seuraavaksi päivitin kuluttajalaskujen reittitiedot Redis:ssä olevien muutaman lähettäjäasiakkaan osilta. Ihmettelin päivityksen lokitiedostosta miksi osalle asiakkaista ei ollut tullut yhtään uutta reittiä tai päivityksiä edellisiin reitteihin. Tutkittuani asiaa hieman perusteellisemmin ja katsoessani lokitiedoista edellisiä päivityksiä, havaitsin ongelman edellisen päivityksen aikaleiman suhteen.

Aikaisemman päivitysrutiinin toteutuksen yhteydessä olin käyttänyt virheellisesti yleistä aikaleimaa kuvaamaan edellisen päivityksen päivityshetkeä ja koska kaikilla päivityssykleillä ei ollut päivitetty kaikkia laskutusaiheita, oli käynyt niin että osalta laskutusaiheista puuttui osa päivityksistä eikä niitä ladattu osittaisen päivityksen yhteydessä. Päätin muuttaa aikaleimakäytännön siten, että Redis:iin tallennettaisiin jokaiselle laskutusaiheelle oma aikaleimansa, jolloin osittaisen päivityksen pitäisi toimia kaikilla lähettäjäasiakkailla vaikka päivitys jostain syystä suoritettaisiin vain osalle esimerkiksi käsin tehtynä huoltoajona.

Odottaessani koko reititystiedon päivittymistä rupesin miettimään palvelun ylläpidettävyyttä tämän virheen johdosta ja tulin lopputulokseen että olisi ehkä hyvä toteuttaa täysi päivitys siten että se voitaisiin suorittaa isäntänoodille niin kutsutussa

”offline” –tilassa, jolloin reitityspalvelu voisi palvella reitityskyselyjä ns. vanhasta datasta joka olisi saatavilla palvelun omasta noodista. Päivityksen valmistuessa voitaisiin tiedot synkronoida palvelun käyttämään Redis –orjanoodiin. Tutkiessani Redis:in dokumentaatiota havaitsin komennon ”SLAVEOF” jolla voitiin suorituksen aikana muuttaa Redis:in replikoinnin toimintaa, samaten komennolla voidaan replikointi poistaa käytöstä. [20]

4.5 Yritysreitityksen suunnittelua

Yleisin käyttötapaus yritysreititykselle on sinänsä yksinkertainen ja toimii seuraavasti; katsotaan vastaanottavan yrityksen OVT –tunnisteeseen kytketty voimassa oleva reititystieto ja palautetaan reitti. Käytännössä reititys ei kuitenkaan toimi näin yksinkertaisesti vaan RLU -palvelu mahdollistaa lähettäjäyrityksen määrittelemään omia yritysreitityksiä usealla tavalla. Lähettäjä voi ylikirjoittaa jonkin laskuaiheen (message subject) reitityksen lähetyskanavan, ylikirjoitus vakioreititykselle voidaan tehdä myös jollekin osoitteelle. Näiden kanava- ja osoitekohtaisten reittien lisäksi lähettäjäyritys voi määritellä omaan yksityiseen osoiteavaruuteen omia asiakasnumeroita yrityksille sen sijaan, että käyttäisivät Tieke –palvelusta saatavaa yrityksen OVT –tunnistetta. Käytännössä siis reittitieto muodostetaan vähintään lähettäjän ja vastaanottajan OVT-tunnuksen perusteella.

Tuotehallinnan ehdottama tapa toteuttaa kaikkien näiden ominaisuuksien toiminta olisi ristiinkartoittaa jokainen osoitepari erikseen. Valitettavasti tämä ei käytännössä olisi mitenkään mahdollista, koska se vaatisi erittäin suuria määriä keskusmuistia kaikkien reittien varastoimiseen. Lisäksi reittitiedon päivittäminen ei onnistuisi yleisesti vain vastaanottajan OVT:lle, koska osoitetieto on sidottu lähettäjä-vastaanottajapariin.

Muistivaatimuksia on helppoa havainnollistaa esimerkillä. Kuvitellaan ristiinkartoitus (crossmapping) kolmelle yritykselle jolloin ristiinkartoitusmerkintöjä tulisi 6, mikäli yrityksiä olisi neljä, tulisi ristiinkartoitusmerkintöjä tehdä 12, 5 yrityksellä 20 ja niin edelleen.

Taulukossa (3) on esimerkin mukaisesti ristiinkartoitettu kolmen kuvitteellisen yrityksen reittitieto vain lähettäjän ja vastaanottajan tunnisteiden mukaan, esimerkistä on jätetty mahdollinen laskuaihe kolmantena tunnisteena pois.

Lähetystunniste	**Vastaanottajan tunniste**	**Reitti**
OVT1	OVT2	Reitti1
OVT1	OVT3	Reitti2
OVT2	OVT1	Reitti3
OVT2	OVT3	Reitti4
OVT3	OVT1	Reitti5
OVT3	OVT2	Reitti6

Taulukko 3. Esimerkki 3 yrityksen reittitaulun ristiinkartoituksesta (crossmapping).

Näyttäisi siis siltä, että vaadittavien merkintöjen määrä noudattaa kaavaa (1).

$$S(n) = n^2 - n \tag{1}$$

Jokaista merkintää kohden pienin tallennettava datamäärä (Redis avain=arvo –parina), huomioimatta Redis:in sisäisiä tietorakenteita ja niiden mahdollisia lisämerkkejä olisi käytännössä 4 kpl lyhimpiä madollisia OVT-tunnisteita sekä kaksi merkkiä kanava formaatin kertovalle tiedolle ja yksi merkki operaattorin ja yrityksen osoitteen erottamiseen. Esimerkki:

- Redis avain: ”003700000001#003700000002”
- Redis arvo: ”003700000002@003799999999/ta”
    - Poistaen datasta heittomerkit, joita ei tarvita oikeasti saadaan tallennettavan datan määräksi 53 merkkiä (ts. tavua, byte).

Todellinen tallennettavan B2B-reititystiedon koko siis noudattaisi kaavaa (2).

$$S(n) = (n^2 - n) * 53 \tag{2}$$

Kirjoitushetkellä RLU:n tietokannassa on 533063 kappaletta B2B-reititysavaimia asiakkaillemme Suomessa, Norjassa ja Ruotsissa. Yllä olevaan kaavaan sijoittaen tämän luvuksi n, saadaan tulokseksi noin 14 026 Gt (Gigatavua). Todellisuudessa Redis-tietovaraston sisäinen kirjanpito lisää muistinkulutusta joitakin prosentteja, lisäksi esimerkki laskettiin pienimmän mahdollisen tiedon määrällä. Näillä suureilla ei kuitenkaan ole kokoluokan kannalta mitään merkitystä, koska kirjattavien tietojen määrä skaalautuu toisen potenssin mukaan, kun taas muut mainitut suureet ovat ensimmäisessä potenssissa. Joka tapauksessa laskutoimitus osoittaa, ettei reittien tallentaminen tällä tavoin ole käytännöllistä eikä millään tavalla palvelinkustannuksiltaan järkevää.

Päätin keskustella reititystiedon tallentamisen erilaisista vaihtoehdoista myös RLU arkkitehdin kanssa, koska hänellä oli hieman parempi käsitys RLU -palvelun tietokannan datasta vain reitityksen osilta. Tietokannassa on paljon muutakin dataa, joka ei ole reititysvälimuistille oleellista. Lisäksi jollakin lailla rakenteellisesti relaatioilla tai muilla sidoksilla rakennettuun dataan olisi huomattavasti helpompi tehdä päivityksiä ja poistoja kuin yksinkertaiseen ja tilaa tuhlaavaan ritstiinkartoitustauluun tai muuhun hyvin simppeliin rakenteeseen.

Arkkitehti ehdotti, että lataisimme välimuistiin tiedot samassa loogisessa muodossa kuin RLU -palvelu niitä käyttää reitityskyselyjä tehdessään. Tämä lähestymistapa vaatisi sovelluslogiikkatasolla koodillisesti samojen ominaisuuksien toteuttamista kuin mitä SQL –tietokannat sisäisesti tekevät JOIN –komentojen osilta; koska Redis ei tue minkäänlaisia relaatioita eikä myöskään avainten tai arvojen yhdistämistä JOIN –komennon tyyliin, päätimme luoda Redis:in välimuistiin seuraavat loogiset tietokokonaisuudet.

- Operaattoreiden kanavatiedot ja yhdysliikennepistelistaus (LMC)
    - Osittain luotu jo B2C –käyttötapaukselle.
- Lähettäjien laskuaihetiedot, jotta asiakkaan lähetysluvan voimassaolo voitaisiin varmistaa
- Vastaanottajan tiedot ja ”mappaus” (mapping) reittiin
- Poikkeusreititystlistaus, jossa tiedot yhdistävänä tekijänä laskuaiheentunniste (msgID) ja palveluosoitteen tunniste (SvcAddrID).

Yritysreitityksen reititystietojen osittainen päivitys onnistuisi tietokannasta löytyvän muutostunnisteen (changeID) avulla. RLU -tietokannassa jokaiselle palveluosoiteriville (taulu SVCADDR) tallennettiin jokaisen muutoksen muutostunniste, joka on automaattisesti generoitu uniikki juokseva tunniste. Jos siis ensimmäisellä täyspäivitysajolla tallennettaisiin Redis:iin suurin tunnettu palveluosoitetaulun muutostunniste, seuraavalla osapäivityksellä riittäisi kun ladattaisiin kaikki rivit joilla on suurempi muutostunniste kuin Redis:in tiedoissa on. Päivityksen jälkeen suurin löydetty muutostunniste tallennettaisiin Redis:iin ja voitaisiin taas käyttää seuraavalla päivitysajolla.

Valitettavasti toteutuksen osilta tässä vaiheessa ei ehditty kuin hieman luonnostella sovelluskoodia ennen kuin projekti keskeytyi. Organisaatiossamme kärsittiin tiedon kulkemiseen liittyvistä ongelmista. Norjan maayhtiömme oli jo huomattavasti aikaisemmin tehnyt sopimuksen erään asiakkaan kanssa joka vaatisi uuden yhdysliikennepisteen rakentamista, lisäksi myytyyn tuotteeseen kuului sähköisen reitityksen ylläpito meidän puolesta.

Tiedonkulun ongelmista johtuen SDU –yksikkö (Software Development Unit) ei ollut saanut tästä tietoa. Kuultuamme asiasta meillä oli 3 viikkoa aikaa päivittää olemassa olevaa reitityspalvelua ja rakentaa uusi yhdysliikennepiste. Reitityspalvelun päivitys vaati myös norjalaisen ”enrollment” –viestien käsittelyn toteuttamista RI-import -palveluun joka ei tuolloin tukenut kuin suomalaisen pankkiverkon RI-viestien käsittelyä.

Valitettavasti aikataulut venyivät niin kehityksen kuin varsinaisen käyttöönoton suhteen ja reititysvälimuistiprojekti oli pysähdyksissä 10 viikkoa. Norjan projektin aikana sain kuulla että reititysvälimuistin prototyypistä tultaisiin rakentamaan NIP -alustan virallinen reitityspalvelu ja minut siirrettiin ”NIP Release 1” -projektiin lokakuun lopussa 2013.

5 YRITYSLASKUJEN REITITYS

Palvelun ominaisuuksien osalta kappaleessa 5 keskitytään kuluttajareitityksen laadun parantamiseen ja yritysreitityksen toteuttamiseen. Sovelluskehitysprojekti kohtaa aikataulumuutoksia NIP -projektista johtuen, joiden vuoksi töitä priorisoidaan, palvelun tehokkuutta optimoidaan ja palvelun laatua parannetaan.

5.1 Projektin aikataulu muuttuu

Projektin uudelleenkäynnistys tammikuun alussa alkoi masentavasti, tuotehallinnalla oli reititysvälimuistista hyvin erilainen käsitys kuin projektin alussa annettu POC –toteutuksen idea. Lisäksi palvelu olisi tarkoitus tuotteistaa ja ottaa käyttöön huhtikuussa 2014. Tuotehallinta olisi kaivannut reitityspalveluun sellaista muutosta että asiakkaalle olisi voitu tallentaa yhden reititysavaimen taakse lista useista ulosmenevistä reiteistä ja soveltuvista reitityskanavista, joista myöhemmin tehtäisiin kanavavalinta kanavankäyttökustannusten ja asiakasmääritelmien mukaan.

Muutaman palaverin jälkeen pääsimme yhteisymmärrykseen, että emme ehtisi toteuttaa niin suurta muutosta alla olevaan reitityspalveluun ja testata sen toimimista ennen huhtikuuta. Palvelulla oli jo valmiiksi useita tuhansia olemassa olevia asiakastoteutuksia joita ei saisi rikkoa, lisäksi A2A –järjestelmän muuttaminen useita reittejä ymmärtäväksi ei olisi realistista tuossa aikataulussa. Tällöin vähintäänkin palvelun olisi pitänyt osata toimia kahdessa eri tilassa; vanhassa yhden reitin ’legacy tilassa’ sekä uudessa monien reittien tilassa. Käytännössä siis kohtuullisen yksinkertainen muutos vaatisi huomattavan määrän muutostöitä ympäröivissä järjestelmissä, jonka lisäksi vielä moni kustomoitu asiakastoteutus olisi pitänyt jotenkin pystyä todentamaan toimivaksi muutosten jälkeen.

Lisäksi ominaisuuden tukeminen reititysvälimuistissa vaatisi muutoksia alkuperäiseen suunnitelmaan, eikä yrityspuolen reitityksestä ollut edes vedosmuotoista koodipohjaa valmiina. Lopulta päädyimme siihen että yritämme luoda reititysvälimuistiin samat ominaisuudet kuin RLU -palvelussa on sisäiselle reititykselle, ja saada ne valmiiksi huhtikuuhun mennessä. Tämä siis tarkoittaisi että ulkoista Netposti –palvelua ei vielä tuossa julkaisussa tuettaisi.

Lisäksi kuluttajareittien latausominaisuuteen pitäisi tehdä muutos, Norjan projektin yhteydessä tein suuren päivityksen RI-import palveluun jolloin havaitsin, että emme olleet muistaneet implementoida muutamalla asiakkaalla käytössä olevaa asiakasnumeron poiminta- ja tarkistusominaisuutta. Lyhyesti kuvattuna ominaisuus toimii siten, että sisään tulevasta RI-sanomasta poimitaan laskun viitenumero, josta poimitaan asiakasnumero sääntölausekkeen (Regular Expression) perusteella. Poimittu asiakasnumero ja viitenumeroyhdistelmä sen jälkeen tarkistettaisiin toisella sääntölausekkeella ja poimittu asiakasnumero tallennettaisiin reititysavaimeksi.

Jo tässä vaiheessa ilman B2CCacheManager –moduulin tarkistamista tiesin ettei tämä ominaisuus toiminut oikein reititysvälimuistissa, koska asiakasnumero poimittiin FI_RI –tietokantataulusta sillä oletuksella että reititysavain olisi sama kuin RI –sanomassa, kuten se valtaosassa käyttötapauksia oli. Tätä kirjoittaessa poiminta- ja tarkistusominaisuutta käytti 6 laskutusaihetta, kun tietokannasta löytyi yhteensä 2000 aktiivista laskutusaihetta. Tässä vaiheessa lukijan kannattaa tutustua liitteeseen 3 ”RLU -tietokantakuvaus”.

5.2 B2CCachemanager -päivitys

Ennen varsinaista yritysreititystä päätin korjata B2CCacheManager –komponentin toiminnan sääntölausekkeen (regular expression) perusteella tehtävän poiminta- ja tarkistusominaisuuden. Norjan projektin yhteydessä eräälle suomalaiselle asiakkaalle tehtiin manuaalisia ylläpitotoimenpiteitä, joita välimuistin päivitys ei ymmärtänyt, koska kuluttajareittejä luettiin saapuneiden RI –sanomien tiedoista, kun taas kyseinen päivitys oli tehty suoraan tietokantaan skriptillä joka muokkasi palveluosoitetaulun (SVCADDR) viittauksia toisiin tauluihin.

Käytännössä tietylle ennalta määritellylle asiakasjoukolle vaihdettiin kanavaformaatti suoramaksusta tavalliseen sähköiseen laskuun, RI –sanomataulun sisältöä ei muokattu koska se olisi vaikuttanut seuraaviin samoille asiakkaille kohdistuneisiin RI –viestien käsittelyyn ja olisi luultavasti vaatinut tietyissä tapauksissa taas manuaalisia huoltotoimia.

Johtopäätöksenä tästä tietokantahakua pitäisi normaalissakin käyttötapauksessa laajentaa

ainakin sen verran, että palveluosoitetaulusta (SVCADDR) haettaisiin viittaus kanavatauluun (CHANNEL) ja kanavan formaattitauluun (CHANNELFMT) joiden avulla haettaisiin kohdekanava ja sen viestimuoto. Sääntölausekkeeseen (regular expression) perustuva poiminta- ja tarkistusominaisuus vaatisi lisäksi vielä varsinaisen reititysavaimen hakemista avaintaulusta (KEY).

Päätin luoda molempiin tapauksiin omat SQL –lauseet ja kokeilla niiden käytännön suorituskykyä noutamalla 1000 uusinta reittiä tietokannasta. Valitettavasti tietokannan KEY –taulun indeksoinnissa oli vikaa, koska seuraavista SQL –lausekkeista jälkimmäinen vaati yli kymmenenkertaisen suoritusajan.

```
--; Get route for normal case
 select F.rikey, F.BIC, F.iban, S.gweba, C.chnlcode, CF.format,
 SV.enabled, SV.svcaddrid
 from FI_RI F
 join SvcAddr SV on (F.svcaddrid = SV.svcaddrid )
 join SERVICE S on (SV.SERVICEID = S.SERVICEID )
 join CHANNELFMT CF on (S.LMCID = CF.LMCID)
 join CHANNEL C on (CF.CHNLID = C.CHNLID)
 where F.MSGID = ? and F.rikey = ? ;

--; Get route for regex pickuprule
 select F.rikey, K.KEY, F.BIC, F.iban, S.gweba, C.chnlcode, CF.format,
 SV.enabled, SV.svcaddrid
 from FI_RI F
 join SvcAddr SV on (F.svcaddrid = SV.svcaddrid )
 join SERVICE S on (SV.SERVICEID = S.SERVICEID )
 join CHANNELFMT CF on (S.LMCID = CF.LMCID)
 join CHANNEL C on (CF.CHNLID = C.CHNLID)
 join KEY K on (SV.CONSID = K.ROWID AND K.TABLE = 'CONSUMER')
 where F.MSGID = ? and F.rikey = ? ;
```

Tässä vaiheessa lähetin konesalipalveluiden ja tietokantapalveluiden toimittajalle tukipyynnön ja pyysin kuvauksen KEY –taulun indeksoinnista, toimitin testissä käytetyt SQL –kyselyt tukipyynnön mukana ja kerroin ongelman yleispiirteet. Päätin ainakin väliaikaisesti toteuttaa tuen molemmille päivitystapauksille B2CCacheManager –komponenttiin. Päivitys toimi nyt siten että mikäli asiakkaalla oli käytössä sääntölauseke, niin reittitiedot ladattiin yllä olevan listauksen jälkimmäisellä (ja hitaammalla) SQL –kyselyllä, muutoin käytettäisiin nopeampaa kyselyä.

Tässä vaiheessa päätin toteuttaa aiemmin kappaleessa 4.3.1 pohditun ”offline” –

päivitysmenetelmän kuluttajareittien latauksseen. Helpomman koodin ylläpidettävyyden vuoksi jaoin alkuperäisen UML –kaavion (Unified Modeling Language) suunnitelman update() –metodin siten, että moodista riippuen kutsuttaisiin metodeja ”partialUpdate() ” tai ”fullUpdate()”. Tämän jälkeen muokkasin täyden päivityksen metodiin niin kutsutun offline –ominaisuuden.

Käytännössä ominaisuus toimii seuraavasti.

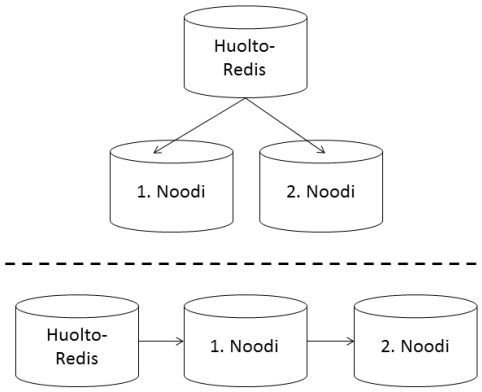
1. Otetaan yhteys Redis orjanoodiin ja annetaan komento ”SLAVEOF NO ONE”
    a. Redis –replikointi keskeytyy isäntänoodista kyseiselle orjanoodille
2. Ladataan lista päivitettävistä laskuaiheista
3. FOREACH laskuaihe
    a. Ladataan lista päivitettävien asiakkaiden RI –viestin tunnisteista (rikey)
    b. FOREACH rikey
        i. IF !regexInUse
            1. Käytetään kevyempää SQL –lausetta reitin hakuun
        ii. ELSE:
            1. Käytetään raskaampaa SQL –lausetta reitin hakuun
    c. FOREACH route
        i. Tallennetaan tai päivitetään reitti Redis tietovarastoon
4. Otetaan uusi yhteys Redis-orjanoodiin ja annetaan komento ”SLAVEOF IP PORT”, missä IP = Redis isäntänoodin IP-osoite ja PORT kyseisen noodin TCP -portti.

Neljännen loogisen vaiheen komennon (SLAVEOF IP PORT) jälkeen Redis orjanoodi aloittaa täyden tietokantareplikoinnin isäntänoodilta. Tämän replikoinnin aikana reititysvälimuisti ei pysty vastaamaan reitityspyyntöihin vaan käytännössä kaikki pyynnöt osaavat odottaa Redis:in toiminnan palaamista jo aiemmin toteutetun ”BusyLoadingError” –poikkeuskäsittelyn tuen vuoksi (katso kappale 4.2).

5.3 Yritysreitityksen toteutus.

Kuluttajareitityksen latausmuutostöiden aikana olin saanut sähköpostia sekä stage- että testiympäristöstä. Palvelimet olivat valmiit käytettäväksi molemmissa ympäristöissä ja ainakin testipuolen ympäristö pitäisi saada otettua käyttöön, jotta NIP -projekti voisi aloittaa tuotteen integroinnin reititysvälimuistiin. Tässä vaiheessa käytin pari päivää aikaa molempien ympäristöjen asentamiseen sekä testiympäristöön sopivan datan luomiseksi. Käytännössä minun oli muutettava kaikki sellainen tieto anonyymiksi jota käytettäisiin testiympäristössä, koska ympäristöksi oli valittu Microsoft:in tarjoama Azure – pilvipalvelu. Asennuksen jälkeen lähetin pikaisen dokumentaation palvelun käytöstä kummassakin ympäristössä projektitiimille.

Samalla testasin useamman Redis –noodin käyttöä oikeasti hajautetussa ympäristössä, jossa minulla oli käytettävissä kaksi fyysistä palvelinta. Kuluttajareittien latauksen muutoksessa en ollut huomannut ajatella sitä, että mikäli kaikki orjanoodit olivat suoraan kytköksissä Redis isäntänoodiin, niin niille kaikille tulisi lähettää replikoinnin katkaiseva ”SLAVEOF NO ONE” –komento. Pikakorjauksena päätin muuttaa Redis -asetustiedostoja siten, että Redis -replikointi muodostaisi puurakenteen sijasta jonon. Kuvassa 7 on katkoviivan yllä tilanne ennen muutosta ja alla muutoksen jälkeen.



Kuva 7. Redis –replikointi Stage –ympäristössä.

Todensin muutoksen toimintaa nopeasti sillä tavalla, että pakotin kuluttajalaskujen

latauksen ”offline” –tilassa yhdelle asiakkaalle ja käynnistin reitityspyyntöjä lähettävän testisovelluksen lähettämään reitityspyyntöjä palvelun kuormantasaajan virtuaaliseen WWW –palvelimeen, josta kuormantasaaja jakaa pyyntöjä taustanoodeille vuorotellen (Round Robin).

Palveluiden lokitiedoista näkyi kuinka palvelun 1. Noodi lakkasi palvelemasta pyyntöjä noin 11 sekunnin ajaksi, jonka jälkeen palvelun 2. Noodi lakkasi palvelemasta noin 14 sekunniksi. Yhtään reitityskyselyä ei jäänyt vastaamatta eikä yhdessäkään ollut virhettä. Replikointistrategia siis näytti ihan toimivalta. Redis noodien päivitysnopeuden ero selittyy sillä seikalla, että Redis huoltonoodi ja 1. Redis palvelunoodi sijaitsevat fyysisesti samalla palvelimella, eli tieto siirretään TCP/IP:llä ns. lokaaliyhteyden yli, kun taas 2. palvelunoodi joutuu lataamaan tiedot 1Gbit ethernet -verkon ylitse 1. palvelunoodista. Käytetty 1Gbit/s ethernet –yhteys on huomattavasti hitaampi kuin palvelinten SAN –High –luokan (tuotenimi) levyosiot.

5.3.1 Yritysreittien latauksen toteutus

Valitettavasti projektin keskeytyessä edellisenä marraskuuna olin unohtanut ns. puhtaaksikirjoittaa palaverimuistiinpanot ja jouduin käyttämään pari ylimääräistä päivää RLU:n tietokannan sisällön tutkimiseen ja pohtimaan miten tietokannan relaatiot oli rakennettava sovelluskoodiin. Pääsin kuitenkin kohtuullisen hyvin tehtävässä eteenpäin ja suunnitelmani alkoi muodostua.

Kuten aikaisemmin oli mainittu, päivityksessä ladattaisiin seuraavat tiedot. Kanavatietojen listaus (LMC), lähettäjän laskutusaiheen tiedot lähetysluvan varmistamiseksi, vastaanottajan tiedot, reititystiedot ja reitityspoikkeuksien tiedot. Rakenteellisesti suunnittelin B2BCacheManager –komponentin vastaavaksi kuin B2CCacheManager –komponentti aikaisemmin kuvattiin UML -kaaviossa kappaleessa 3.2.1.

Alla olevasta erittelystä löytyy tietojen lataukseen käytetyt SQL –kyselyt sekä kuvataan miten tieto mallinnettiin Redis tietovaraston avain-arvopareiksi.

Kanavatiedot (channels): tietokenttien välimerkkinä on käytetty huutomerkkiä (!):

```
select S.ServiceID, S.Service, S.Env, S.GWEBA, L.ChnlCode, L.Format
from Service S
join Lmc L on (L.LmcID=S.LmcID)
where S.serviceID = ? ;
```

RedisKey = ”ServiceID=NNNNN”, missä N = numeerinen serviceID

RedisValue = ”Service!Env!EBA!ChnlCode!format”

Esimerkki ”anonymisoidulla” datalla:

RedisKey = "ServiceID=1014"

RedisValue = "invoice!P!BELRFI0000123A!003709999999!ta"

Lähettäjän tietojen noutaminen ja tallentaminen tapahtuisi seuraavasti:

```
select K.Key, M.Service, M.SubjectCode, M.MsgID
from Key K
join Msg M on (K.Table='SITE' and K.RowID=M.SiteID);
```

RedisKey = "Key!Service!SubjectCode"

RedisValue = ”MSGID”

Esimerkki ”anonymisoidulla” datalla (OVT –tunniste ja laskuaihe vaihdettu):

RedisKey = "00370999999999!invoice!PUHELINLASKU"

RedisValue = ”77”

Vastaanottajien tietojen lataus:

```
select S.serviceID, S.Service, S.Env, K.Key, A.SvcAddrID,
       COALESCE(A.EBA, A.EAddrStr) EAddr,
       COALESCE(A.EBA, S.GWEBA) EBA, S.LmcID,
       A.Enabled, A.ValidFrom, A.ValidTo
from Key K
join SvcAddr A on (K.Table='SITE' and K.RowID=A.SiteID)
join Service S on (S.ServiceID=A.ServiceID)
where K.key = ? ;
```

RedisKey = "Service!Env!Key"

RedisValue = "ServiceID!SvcaddrID!EAddr!EBA!LmcID!Enabled!ValidTo"

Esimerkki josta arvo ”KEY” on anonymisoitu arvoon ”00370999999999”:

```
RedisKey = "invoice!P!00370999999999"
RedisValue = "1014!7574!00370999999999!BELRFI0000123A!12!E!None"
```

Tiedon rakenteessa huomattavaa on, että kanavatiedoissa ja vastaanottajan tiedoissa saattaa mahdollisesti toistua sama tietue ”S.GWEBA”. Tämä on valitettavasti pakollista johtuen RLU -palvelun tavasta käsitellä sisäisiä ja ulkoisia operaattoreita sovelluslogiikallisesti eri tavalla. Tämä johtuu tavasta jolla B2B –reititystieto on alun perin tallennettu RLU palvelua edeltäneeseen MSD (Master Site Directory) -palveluun. Aiheeseen palataan tarkemmin seuraavassa kappaleessa joka tulee käsittelemään reititystä ja tiedon käyttöä reitityksessä.

Asiakkaan itse määrittelemille poikkeusreiteille tieto haettiin seuraavasti:

```
select X.MsgID, X.SvcAddrID, X.Enabled
from Msg2Addr X;
RedisKey= “MsgID!SvcAddrID”
RedisValue = “Enabled”
```

Poikkeusreiteistä ei tosin tallennettaisi kuin ne tapaukset joiden Msg2Addr –taulun Enabled solun arvo olisi “E” (Enabled). Poistettuja tai epäaktiivisia reittejä ei tallennettaisi (tilat: E = Enabled, A = Available tai D = Disabled). Poikkeusreitityksessä vain ”Enabled” –tilassa olevilla reitin ylikirjoitusmääreillä on käyttöä. Muualla reitityspalvelussa ei yleisesti ”Available” –tilan reittejä käytetä vaikka asiakas voi halutessaan pyytää niiden käytön sallimista, käytännön tilanteessa en kuitenkaan ominaisuudelle ole koskaan tarvetta tai käyttötapausta nähnyt.

Kokeiltuani tietojen latauksen onnistumista tein pikaisen kopion B2CCacheManger –luokasta ja muokkasin sen tietojen haku- ja tallennusosuudet käyttötapaukselle soveltuvaksi. Osittaispäivityksen toimimiseksi vaihdoin viimeisen viestiaiheen päivityksen aikaleiman tilalle tietokannasta löytyneen suurimman muutos-ID:n ja tallensin sen Redis:iin. Päätin tallentaa koko B2B –reititystiedon omaan tietokantaindeksiin kaksi (2), jolloin voitaisiin täyslatauksen kohdalla tehdä samoin kuin B2CCacheManagerissa, eli huolto -Redis:in ja ensimmäisen palvelu -Redis:in välinen replikointi voitaisiin pysäyttää

päivityksen ajaksi.

Päätin koittaa koko B2B –reititystiedon lataamista RLU -stagepalvelusta, mutta se ei onnistunut WWW-palvelimen 5 minuutin aikakatkaisun vuoksi. Olin myöhään illalla vielä toimistolla kun sain sovelluskoodillisesti toiminnallisuuden valmiiksi, joten tarkistin mitä tuotannossa tapahtuu. Tuotannon kuorma normaalista työajan aikaisesta korkeasta kuormasta oli jo laskenut ja A2A –järjestelmä sekä RLU -tuotantotietokanta molemmat käyttivät vain osan saatavilla olevista resursseista (CPU ja muisti).

Tästä johtuen muutin pikaisesti B2BCacheManagerin ottamaan yhteyttä tuotannon reititystietokantaan ja koitin ladata B2B –reittitiedot sieltä. Ensimmäinen lataus epäonnistui merkistökoodausten vuoksi. Tietokannasta tuli pääasiassa UTF-8 (8-bit Unicode Transformation Format) koodattua tietoa, valitettavasti sieltä tuli myös muutamia ISO-8859-1 koodattuja tietoja jotka kaatoivat sovelluskoodin. Kiersin ongelman pikaisesti tekemällä poikkeuskäsittelyn, joka hylkäsi ISO-8859-1 muodossa olevat tiedot. Nyt päivitys meni läpi ja aikaa kului noin 2,5 minuuttia.

Seuraavana aamuna aloittaessani työt ajattelin ensimmäisenä tehdä oikean korjauksen ISO-8859-1 muotoisten tietojen käsittelyyn, siten että Python koodissa yritettäisiin uudelleen koodata (encode) nuo tiedot UTF-8 muotoon. Päätin yrittää ladata tiedot uudestaan, mutta päivitys epäonnistui jälleen WWW-palvelimen aikakatkaisuun, jonka jälkeen tutkin sovelluskoodia ja havaitsin käyttäväni edelleen tuotantotietokantaa. Merkistöongelma oli siis saatu korjattua, mutta nyt tuotannossa oli jotain ongelmaa, jonka vuoksi lataus ei onnistunut.

Päätin tarkistaa tuotannon tilanteen ja huolestuin; jaetun tietokantaympäristön palvelin käytti kaikki olemassa olevat CPU –resurssit, jonka lisäksi tietokantaympäristön SAN –levyosioiden kuormitusaste oli huolestuttavan korkea. Päätin ajaa samaisen testi-reitityskyselyn SoapUI –työkalulla tuotantoreitityspalveluun, jolla kappaleessa 1.1 mainittu RLU -palvelun 1000-1200 reittiä minuutissa nopeus oli mitattu ennen projektin aloittamista. Palvelun nopeus oli romahtanut noin 400 reittiin minuutissa. Kaivelin pikaisesti tikettijärjestelmästämme edellisen tukipyynnön liittyen tuotanto- ja stage -

ympäristön KEY -taulujen indeksoinnin eroihin ja pyysin tutkimaan asiaa nopeasti, koska tuotanto oli hidastunut huomattavasti.

Seuraavaksi siirryin kirjoittelemaan tuotannon hidastumisesta eskalointisähköposteja tuotehallinnalle sekä tuotevastaaville ja ilmoitin, että tuotannon tietokantapalvelimelle täytyisi lisätä resursseja viipymättä tai siirtää RLU -palvelun tietokanta omalle palvelimelleen. Varoitin tuotehallintaa että isojen reitityskyselyiden kohdalla tulisi ongelmia tuotannossa, mikäli palvelu hidastuisi vielä lisää. Tuotannossa A2A -ympäristön RLU-asiakassovelluksessa nimittäin on kovakoodattuna maksimina 10 minuutin aikakatkaisu, aikakatkaisun tapahtuessa A2A -järjestelmä käsittelee tilanteen virheenä ja koko laskuerä jouduttaisiin käsittelemään uudelleen manuaalisesti.

Eskalointisähköpostien kirjoittelun aikana sain konesalipalveluiden toimittajan tietokantaylläpidolta vastauksen KEY –taulun indeksointiin. Jostain syystä stage-ympäristöstä puuttui kokonaan KEY -taulun indeksointi. Pyysin rakentamaan stage-tietokantaan samanlaisen indeksoinnin KEY –taululle kuin tuotannossa.

Loppupäivän käytin kirjoittaessani käynnistysskriptiä sekä ns. Python ”main” luokkaa jonka avulla B2BCacheManager- ja B2CCacheManager –luokat voitaisiin ajaa komentoriviltä, koska käytössäni ei ollut yhtäkään tietokantaympäristöä josta saisin varmasti ladattua yritysreititystiedot ennen WWW –palvelimen aikakatkaisua. Seuraavana päivänä pääsin kokeilemaan latausta, stage –ympäristön tietokannasta B2B –reititystiedot sai ladattua noin 9 minuutissa, tuotannosta lataus kesti 6 minuuttia.

5.3.2 Yritysreititys

Saatuani lopulta ladattua reititystiedot Redis:iin aloin toteuttamaan yritysreitityksen logiikkaa. Kiireestä johtuen koodasin ”proof-of-concept” –tyyliset reititysrutiinit RedisHandler –luokan sisään, josta ne voitaisiin myöhemmin siirtää omaan reitityslogiikkaluokkaansa. Elettiin helmikuun puoliväliä ja mikäli jotain käyttökelpoista pitäisi saada huhtikuuksi, oli pakko oikoa toteutuksessa ja jättää hienostelu myöhemmäksi.

Tein yksinkertaisen muutoksen views –luokkaan joka vieläkin piti sisällään kuluttajareitityksen POC –toteutuksen, mikäli reitityspyynnön tyyppi oli B2B niin kysyisi views –luokka reittiä ”redisHandler” –luokan ”routeB2Bpoc()” –metodilta. Kyseisen metodin alle aloin toteuttaa pikaisesti B2B -reititystä seuraavasti, mutta heti toteuttamaan lähtiessä havaitsin suunnitteluvirheen reititystietojen lataamisessa. Poikkeusreititys pystyttiin tunnistamaan, muttei toteuttamaan.

Alustavasti reititys toteutettiin seuraavasti:

1. Reitityskyselystä poimitaan lähettäjän tiedot: **SenderID**, **ServiceCode** ja **SubjectCode** joista muodostetaan Redis hakuavain "Key!Service!SubjectCode". Mikäli avaimella saataisiin numeerinen vastaus (**msgID**), olisi lähettäjän lähetyslupa voimassa.
2. Reitityskyselystä poimitaan vastaanottajan avain **receiverID** ja ympäristöparametri **Env** joista muodostetaan Redis -hakuavain käyttäen edellisen vaiheen **ServiceCode** arvoa. Hakuavain: ”Service!Env!Key". Mikäli Redis palauttaisi vastauksen muotoa ”ServiceID!SvcaddrID!EAddr!EBA!LmcID!Enabled!ValidTo" olisi vastaanottajalle sähköinen reitti olemassa.
    a. Jos vaihe 2 palauttaisi arvon **None** (null –arvo Python kielessä) vastaanottajalle ei olisi sähköistä reittiä ja jos taas postiosoite löytyisi täydellisessä muodossaan reitityskyselystä palautettaisiin paperikanava; muuten virheilmoitus ettei voida reitittää.

3. Vaiheen 2 vastaustiedoista poimitaan **SvcaddrID** ja se yhdistettäisiin lähetysluvan **msgID** kanssa Redis hakuavaimeksi ”MsgID!SvcaddrID”.
    a. Jos vastaus olisi None, ”D” tai ”A” jatkettaisiin reitin rakentamiseen (vaihe 4)
    b. Jos vastaus olisi ”E”, palautettaisiin virhe ”Exception Routing not implemented”
4. Kohdan 2 vastauksen **ServiceID** arvosta rakennetulla hakuavaimella ”ServiceID=NNNN” , (jossa N = numeerinen arvo) haettaisiin kanavatiedot muotoa ”Service!Env!Eba!ChnlCode!format"

4. kohdan latauksen jälkeen kaikki tarvittava tieto reititysvastauksen reitin rakentamiseksi oli kerätty kasaan, ja täytyi vain muodostaa vastaus. Alla esitellään JSON –muodossa vastausesimerkki, käyttäen edellisen listauksen muuttujanimiä. Rakenteessa on toistuvaa dataa, koska välittäjäpankkimallissa ylemmän tason eba, operator ja format -tiedot muunnetaan välittäjäpankin tiedoiksi ja eaddr kentän vastaaviin arvoihin laitetaan viimeisen kohdepankin tiedot.

```
{
	"eba": "Eba",
	"operator": "ChnlCode",
	"format": "format",
	"eaddr": {
		"addr": "receiverID",
		"operator": "ChnlCode",
		"format": "format",
		"envaddr": "receiverID@ChnlCode/format"
	}
}
```

Seuraavaksi testasin reitityslogiikan ja reititysvastausten oikeellisuutta ajamalla esivalittuja reitityskyselyitä RLU -palveluun ja reititysvälimuistiin. Käytännössä reittivastaukset olivat oikein mutta vastauksen ulkomuoto oli väärin mikäli kanavalistassa (LMC) ei ollut määritelty viestimuotoa (format), pikaisen korjauksen jälkeen ”envaddr” parametri oli kunnossa molemmissa tapauksissa.

Seuraavaksi rakensin 20 000 satunnaisen B2B –kyselyn tiedoston tehokkuustestausta varten ja ajoin työasemakoneellani vastaavan tehokkuustestin kuin B2C –reitityksen

kohdalla. Tulos oli noin 740 reittiä sekunnissa, verrokiksi yksinkertainen B2C –reititystoteutus antoi tulokseksi noin 1320 reittiä sekunnissa. Edellisellä B2C –testauksella vastaava luku oli 1520. B2C –reitityksen lievää hidastumista selittää yksinkertaisesti jo se seikka, että olin näiden kehitystehtävien aikana parantanut reitityskyselyn HTTP –kehyksen validointia, sekä lisännyt pakollisten tietojen tarkistuksen JSON –tiedolle.

Tutkin B2B –reitityksen hitautta ja se näytti johtuvan pääasiassa virheellisestä yhteyden otosta Redis:in B2B –tietokantaindeksiin, käytännössä ennen jokaista lukuoperaatiota koodi loi uuden yhteyden tuohon tietokantaindeksiin joka mitä luultavimmin hidasti toteutusta havaittavasti, yhteyden luonnin lisäksi joka kerta initialisoitiin ”redis-py” -kirjastoluokasta uusi olio, joka hylättiin kyselyn jälkeen. Merkitsin omiin muistiinpanoihin ja koodiin ”TODO” –merkinnät asian korjaamiseksi. NIP -projektin suunnalta oli kova paine päästä integroimaan NIP -alusta reititysvälimuistipalveluun ja minun täytyi keskittyä välillä siihen.

5.4 Integraatio NIP:n kanssa

Aloitimme NIP -alustan integraation reitityspalveluun Azure –testiympäristössämme, saimme itse tiedonsiirron kannalta integraation toimimaan varsin nopeasti muutamassa tunnissa, jonka jälkeen loppupäivän NIP -tiimi toteutti reitityskyselyn HTTP –kehyksen ja JSON –tietokuorman lähettämistä NIP -alustaan. Minä puolestani kirjoittelin anonymisointikoodin jolla saatiin anonymisoitua Redis -tietovaraston sisältö Azure-ympäristöä varten. Jo seuraavana päivänä NIP -tiimi lähetti tietosisällöllisesti oikean reitityskyselyn reititysvälimuistiin käyttäen yksinkertaista B2B –reitityskyselytyyppiä.

Määrittelimme yhdessä reitityskyselyn ja -vastauksen tietosisällöt seuraaville käyttötapauksille, vaikkakaan niistä ei ollut vielä toteutettu kuin perus B2B- ja B2C –reititys.

- B2B –reititys
    - Normaali reititys
    - Asiakkaan pakottama reitti (nk. Forced routing)
        - Ulkoinen reititys (ulosmenokanava toinen operaattori tai pankki)
        - Sisäinen reititys (ulosmenokanava yrityksen asiakkaan sisäinen tili)
- B2C –reititys
    - Normaali reititys
        - Sähköinen lasku
        - Suoramaksu
    - Asiakkaan pakottama reitti
        - Sähköinen lasku
        - Suoramaksu
    - Netposti vastauksen muoto ja toimitustapavalinnan tuki (tuleva ominaisuus)
        - Toimitustapavalinnat ovat Netposti tai Netposti -kopio

Integraatio Stage –ympäristössä vie aikaa kaksi viikkoa johtuen monimutkaisemmasta palvelinten, verkkojen ja sovellusten ylläpitokuvioista sekä vastuiden hajautumisesta useille tahoille, mukaan lukien ulkoistuskumppanit. Integraation jälkeen Stage –ympäristössä ilmeni ongelmia joista NIP -projektin testaajat raportoivat, että reititysvälimuisti kaatuu satunnaisesti osalla reitityskyselyitä palauttaen vain virhekoodia ”HTTP500 – Internal Server Error”.

Lähemmän tutkimisen jälkeen ongelma osoittautui merkistöongelmiksi. NIP:n lähettävässä päässä ei ollut merkistökoodauksen tarkistusta tai pakotusta ja merkistöksi sattui jotain mitä testaajan koneen työkalut käyttivät kun he loivat testimateriaalia, satunnaisuus taas johtui siitä, että osa materiaalista oli luotu Windows –käyttöjärjestelmässä ja osa Linux –käyttöjärjestelmässä jolloin merkistöt vaihtelivat Win1251 ja UTF-8 merkistöjen välillä. Arkkitehti oli määritellyt tiedonsiirron JSON –määritelmän mukaan, jossa määritellään tekstin koodaukseksi UTF-8 merkistö. Merkistöongelmaa ei ilmennyt Azure -ympäristössä,

koska testaajat loivat materiaalia käyttäen Azuren palvelinten työkaluja, jotka käyttivät ympäristön vakiomerkistöä UTF-8.

Käytännössä kyseinen ongelma ei siis vaatisi minulta toimenpiteitä, mutta loin reitityspalvelun views –luokan HTTP –kehyksen ja JSON –tiedon tarkistukseen uuden rutiinin joka osasi ilmoittaa väärästä merkistökoodauksesta; arvelin ettei tämä olisi ensimmäinen eikä viimeinen vastaava virheraportti. Virheen sattuessa palvelu vastaisi "HTTP400 – Bad Request – Incorrect encoding, expecting UTF-8".

Parantelin myös JSON –tiedon osilta virheentunnistusta ja raportointia; mikäli JSON –tietoa ei pystyttäisi kääntämään Python -olioksi vastaisi palvelu samaisella HTTP –koodilla ja laittaisi virheviestiksi "Unable to Parse JSON" ja mahdolliset virheilmoitukset Flask –kirjaston sisältämältä JSON –parsintakirjastolta. Yleisesti ottaen JSON –kirjasto raportoi millä rivillä oli virhe, joskus vain monesko merkki JSON –tiedostosta aiheutti parsintavirheen. Samalla lisäsin JSON –tarkistukseen virheilmoituksen palauttamisen mikäli pakollisia tietokenttiä ei ollut täytetty.

5.5 Pakotetun reitityksen toteutus

Lähettäjän pakottamien reittien käyttötapaus on sinänsä melko yksinkertainen, lähettäjä kertoo vastaanottajan osoitteen ja vastaanottajan käyttämän operaattorin ja kanavan (viestiformaatti), eli sinänsä reitityspäätös on tehty jo ennen reitityskyselyä. Ajamme kuitenkin kaikki pakotetun reitityksen kyselyt reitityspalvelusta läpi jolloin kohde operaattorin olemassaolo voidaan varmistaa, esimerkiksi virheellisen syöttötiedon vuoksi. Lisäksi tällä tavoin voimme muuttaa sisäistä reititystä yhdysliikennepisteiden (GW) suhteen, esimerkiksi vaihtamalla yhdysliikennepisteen sisäistä verkkolaskutiliä (EBA).

Tämän lisäksi reitityspalvelussa on niin sanottu "fallback" –toiminnallisuus, joka aktivoituu mikäli lähettäjä ei ole määritellyt operaattorin kanavaa. Pääasiassa muilla sähköisen laskutuksen operaattoreilla on vain yksi kanava, mutta pankeilla on monia vanhoja (legacy) tuotteita sekä uudempia tuotteita jotka erotetaan kanavatiedon avulla. Mikäli kanavaa ei määritellä reitityskyselyssä, valitsee reitityspalvelu oletusarvoisen

kanavan ja ohjaa materiaalin sinne.

Kuluttajalaskutus tuo tähän vielä lisänä suoramaksukanavan (Direct Payment) jonka vaatimuksissa määritellään että pankkiin lähetettävän sähköisen laskun lisäksi asiakkaalle tulee lähettää ilmoituskirje. Suoramaksun tapauksessa reitityspalvelun on siis varmistettava, että molempien kanavien reitit löytyvät tai palvelun on palautettava virhe. Reitityskyselyt ovat nähtävissä liitteestä 2. [17]

Pakotetun reitityksen käyttötapaus vaatisi täyden operaattori ja kanavalistauksen (LMC) lataamista Redis:in tietovarastoon, ja päätin tallentaa tämän datan aiemmin vapautuneeseen tietokantaindeksiin nolla (0). Käytännössä LMC –listaus tapahtuisi seuraavalla SQL lauseella.

```
select L.chnlcode, L.format, S.gweba, L.defmt, L.type
from service S
join eivc.lmc L on (S.LmcID=L.LmcID)
where L.enabled = 'E';
```

Tämän jälkeen tietoa käsiteltiin sovelluksessa; jokaisen operaattorin kohdalla rivistä jossa L.defmt -sarakkeen arvo oli ’Y’, luotiin erillinen lisärivi Redis:iin, alla kuvatussa esimerkissä normaalit LMC –listauksen avainarvoparit, sekä viimeisenä *-merkillä merkittynä oletuskanavan avain ja arvo.

```
RedisKey = ”OPER=!003705925424!ta!P”
RedisValue = “BELRFI0000123A”
*RedisKey = “OPER=!003705925424!DEFAULT!P”
*RedisValue = “BELRFI0000123B!scn”
```

Tapauksessa jossa kyselyssä tiedettäisiin operaattori ja käytettävä kanava, vastaus sisältäisi siis vain ulosmenevän operaattorin yhdysliikennepisteen sähköisten laskujen tilin. Mikäli kanavaa ei olisi määritelty palvelu hakisi ”DEFAULT” –määreellä olevan tiedon joka sisältää sähköisten laskujen tilin ja oletuskanavan formaatin. Avaimiin on myös määritelty RLU:n tietokannan LMC listauksen L.TYPE –sarakkeen arvo, joka näkyy avaimen etuliitteenä. Etuliitteen merkitys on reitityslogiikan valinnassa, koska sisäisiin, ulkoisiin ja pankkioperaattoreihin jouduttaisiin soveltamaan erilaista reitityslogiikkaa.

LMC -listauksen latausta varten loin uuden välimuistin hallintaluokan ”UtilCacheManager” ja nimesin luokan näin, koska ajattelin LMC -latauksen yhteyteen siirtää muutamia kirjoitussuuntaisia välimuistin ylläpitotehtäviä, joita ei kuluttajalaskutukseen liittyen ollut toteutettu eikä vielä tässä vaiheessa toteutettaisikaan. FKL:n määritelmä kuluttajalaskutuksesta vaatii operaattoria ylläpitämään tietoa viimeisen lähetetyn laskun viitenumerosta, jota tulee verrata saapuvan RI -sanoman sisältämään viitenumeroon. Sanoma on hylättävä jos viitteet eivät ole samat molemmissa. [17]

Pakotetun reitityksen toteutus keskeytyi kun tehtävästä noin 2/3 –osaa oli saatu toteutettua, tuotannon reitityspalvelu alkoi ylikuormittua päivisin kun A2A –alustalla oli vilkkain aika menossa. Jouduimme pitämään muutaman hätäpalaverin ja teimme ns. ”emergency tilauksen” uuden tietokantapalvelimen rakentamiseksi. Lopulta saimme uuden palvelimen pelkästään RLU:n tietokannalle ja siirsimme tietokannan sinne, jonka jälkeen palvelu alkoi toimia normaalisti. RLU -palvelun nopeus nousi noin 2000 reittiin minuutissa ja palvelimen CPU -resurssien käyttö suurimman kuorman aikaan oli noin 30-50% riippuen siitä, montako samanaikaista reitityskyselyä oli ajossa.

Kaikkineen tuotannon ongelmien ratkaisemiseen ja siitä syntyneiden asiakkaiden materiaaliongelmien käsittelystä johtuen sovelluskehitys ei edistynyt kolmeen viikkoon oikeastaan yhtään. Onnekseni sain epävirallista kautta kuulla, ettei huhtikuussa voitaisi kuitenkaan julkaista mitään tuotantokelpoista NIP -projektin osilta, joten sinänsä välitöntä aikataulupainetta ei ollut. Viimeistelin kappaleessa aiemmin esitetyt toiminnollisuudet muuten, mutta jätin vielä viitenumeroiden tallentamisen toteuttamatta.

5.6 NIP:n aikataulumuutos, uudet suunnitelmat

Vuoden 2013 lopulla yrityksen ohjausryhmä oli päättänyt nopeuttaa NIP -alustan tuotteistusta, ensimmäiset pilottiasiakkaat oli tarkoitus saada pilottituotantoon kesäkuun 2014 alussa. Tammikuussa 2014 kokoonnuimme NIP:n kehitystiimin ja tuotehallinnan kanssa kokoushotelliin kolmen päivän suunnittelureissulle. Määrittelimme NIP:lle niin sanotun MVP -toteutuksen (Minimum Viable Product) vaatimukset ja teimme

suunnitelmat NIP:n vähimmäisvaatimuksille, joilla nykyisen laskunvälitystuotteen ominaisuudet voitaisiin tarjota NIP -alustalla. Reititysvälimuisti päätettiin tuotteistaa olemassa olevien ominaisuuksien varaan mahdollisimman nopeasti, koska pilottiasiakkailla oli käytössään hyvin yksinkertaiset sähköisen laskutuksen tuotteet. Muita ominaisuuksia voitaisiin ottaa mukaan tai kehittää mikäli ylimääräistä aikaa jäisi.

Merkitsimme projektisuunnitelmaan huomautuksen, että tällöin meillä ei välttämättä olisi valmiutta reitittää asiakkaiden materiaalia Netposti palveluun jolloin sähköisen postipalvelun sijaan kirjeet ja laskut ohjautuisivat välttämättä paperipostiin Netposti asiakkaiden osilta. Päätimme toteuttaa Netpostin tukemiseen vaadittavat ominaisuudet vasta sen jälkeen kuin reititysvälimuistin tuotteistus olisi muilta osiltaan valmis.

Käytännössä minulle annettiin vapaat kädet oman projektini jatkamisen suhteen ja tehtävien priorisointiin. Päätin alkaa tuotteistamaan reititysvälimuistia seuraavan strategian mukaisesti. Tuotteistusosuuden tekeminen aloitettiin vuoden 2014 tammikuun lopussa.

1. Korjataan ohjelmistovirheet olemassa olevien ominaisuuksien osalta.
2. Korjataan palvelun palvelinympäristö ja korjataan koodissa olevat virheet HA-ympäristöön liittyen ja konfiguroidaan HA-ympäristö käyttökuntoon.
3. Optimoidaan palvelun tehokkuutta ja muutetaan palvelun kaikki kirjoituskäskyt kirjoittamaan tiedot Redis-huoltonoodiin sekä estetään kirjoitukset orjanoodeihin.
4. Testaus, viimeiset sovellusvirheiden korjaukset

Huom. Kappaleessa 5.2 testattiin Stage –ympäristöä jo HA-ympäristössä, mutta nykyisin se ei enää toiminut. Yrityksemme oli tehnyt muutoksia konesalien verkoissa ja palomuuri-konfiguraatioissa. Näiden muutosten yhteydessä HA-ympäristö ei enää toiminut, vain ensimmäinen palvelunoodi sai tehtäviä kuormantasaajalta.

Aloitin palvelun tuotteistamisen ohjelmistotason virheiden korjaamisesta sekä NIP:n integraatiorajapinnan virheiden korjauksesta, jotta NIP -projektin kannalta reitityspalvelu olisi mahdollisimman lähellä lopullista toiminnollisuutta vaikka toimisikin kehityksen aikana huomattavasti hitaammin. NIP -projektin kannalta palvelun nopeus ei ollut vielä

oleellista vaan lähinnä palvelusta saatavan datan oikeellisuus. Listauksen ensimmäisen vaiheen korjaukset sain toteutettua maaliskuun alkuun mennessä.

Maaliskuun aikana saimme vihdoin palvelinympäristöt kuntoon ja HA-ominaisuudet käyttöön. Suurin osa maaliskuusta tuhrautui nk. yritysbyrokratian merkeissä; tilailin paljon palomuuriavauksia ja asetusmuutoksia F5-kuormantasaajaan, ja koska palomuurien sekä kuormantasaajan ylläpito oli ulkoistettu eri toimittajille kommunikaatio sekä muutosten toteutus oli varsin hidasta. Käytin paljon työaikaa viritellessäni erilaisia palomuurien kiertoja, jotta pystyin työskentelemään HA-ominaisuuksiin liittyvien sovelluskoodin osien kanssa.

Sovelluskoodin osilta kaikki kirjoitusoperaatiot ohjattiin lokaalin Redis -orjanoodin sijaan Redis –huoltonoodiin ja sovelluskoodiin kirjoitettiin poikkeuskäsittelyt siltä varalta ettei kirjoitusoperaatio onnistuisi jostain syystä. Käytännössä huoltonoodiin kirjoitettiin jokaisen laskun vastaanottajan viimeisimmät tiedot kuten viimeisen laskun päivämäärä ja viitenumeroita joita tuli verrata RI –viestien sisäänluvussa RI-viestin sisältämään viitenumeroon. Jos viitteet eivät täsmänneet tuli kuluttajareitin muutos hylätä. ”CacheManager” komponenttiin loin ominaisuuden tämän tiedon siirtämiseen välimuistista RLU -palvelun tietokantaan. Listauksen toisen kohdan muutokset ja korjaukset saatiin valmiiksi maaliskuun lopulla.

Kevään 2014 huhtikuun käytin palvelun tehokkuuden optimointiin. Onneksi palvelussa oli paljon kehitystason ”debug lokia” ja jokaisella lokirivillä aikaleimat (timestamp). Kattavan lokitiedon avulla oli helppoa analysoida mitkä sovellusosat tai käyttötapaukset olivat hitaita. Hitausongelmat kohdentuivat Redis tietovarastoa käyttävään kirjastoon, oma sovellustason Redis kirjastoni mm. otti jokaisen B2B -reitityspyynnön yhteydessä uuden yhteyden Redis -tietovarastoon ja katkaisi yhteyden sen jälkeen. Vaikka kyseessä olikin Linux -käyttöjärjestelmän lokaali “socket” yhteys, se hidasti suoritusta, lisäksi sovelluskoodissa oli turhia päällekkäisiä osuuksia jotka aiheuttivat käyttötapauksia joissa sama tieto luettiin yhden reitityskyselyn aikana useita kertoja tietovarastosta.

Suorituskykyanalysoinnin perustella päädyin sellaiseen ratkaisuun, että aloitan suuremman

luokan koodin uudelleenkirjoituksen (code refactor). Ennen sovelluskoodin osittaista uudelleenkirjoitusta päätin toistaa suorituskykymittaukset kehitysympäristössä ja palvelinsalissa olevassa ”Stage” –ympäristössä. Suorituskykymittausten tulokset numeerisina tuloksina olivat varsin masentavat, mutta toisaalta varsin ymmärrettävät johtuen koodin huonosta tasosta ja optimointien puutteesta.

Suorituskyky ennen optimoinnin aloittamista:

- Kehitysympäristö:
  - 8 lähettävää asiakassovellusta: 207 vastausta/sekunti
  - 16 lähettävää asiakassovellusta: 293 vastausta/sekunti
- ”Stage” –ympäristö:
  - 8 lähettävää asiakassovellusta: 234 vastausta/sekunti
  - 16 lähettävää asiakassovellusta: 434 vastausta/sekunti

Ajettuani suorituskykytestit analysoin palvelun lokitietoja uudestaan ja tein johtopäätöksen että palvelinympäristön erilaisuus aiheutti jossakin kohdin ongelmia, eikä Stage -ympäristön raudan yli kaksinkertaisen suorituskyvyn (per palvelin) pitäisi tuottaa näin huonoja tuloksia pelkän sovelluslogiikan osilta. Päätin silti aluksi aloittaa sovelluskoodin korjaamisella ja tutkia ympäristöön liittyviä ongelmia vasta sen jälkeen.

5.7 Palvelun tehokkuuden optimointi

Aloitin sovelluskoodin uudelleenkirjoituksen korjaamalla sovelluksen logiikkaosuuden joka tekisi reitityspäivitykset. Paloittelin koodin käyttötapauksiltaan yrityslaskutukseen (B2B) sekä kuluttajalaskutukseen (B2C). Näiden haarautumien alle paloittelin vielä asiakkaan käyttötapaukset kokonaan omiksi loogisiksi osuuksiksi.
Käyttötapaukset:

- Käyttötapaukset yritys- ja kuluttajalaskutukselle
  - OpusCapita reitittää: Asiakas kertoo lähettäjätunnuksen ja vastaanottajan tunnisteen, palvelu etsii vastaavan reittitiedon.
  - Lähettäjäasiakas reitittää: Asiakas kertoo vastaanottajan osoitteen ja operaattorin, OpusCapita varmistaa tiedon oikeellisuuden ja lähetysluvan

ko. operaattorille. (Jotkin ulkomaiset operaattorit laskuttavat viestien välityksestä, näille tarvitaan erillinen lähetys- ja laskutuslupa lähettäjäyritykseltä).

Aloitin uudelleenkirjoituksen analysoimalla molempien laskutustapausten ja niiden käyttötapausten tietokannan tarvitsemien lukuoperaatioiden osalta. Jaottelin lukuoperaatiot vaiheisiin joita sovelluslogiikka tarvitsisi. Esimerkiksi pakotetun reitityksen tapauksessa ensimmäiseksi suoritettiin lukuoperaatiot asiakkaan lähetysluvan tarkistamiseksi sekä kohdeoperaattorin olemassaolon sekä käyttöluvan tarkistamiseksi. Mikäli sovelluskoodi hyväksyisi reitityskyselyn, mahdolliset muut tiedot ladattaisiin vasta sen jälkeen, jolloin tietokantaa ei turhaan kuormitettaisi.

Lisäksi sovelluskoodista poistettiin kaikki mahdolliset luokkien uudelleen initialisoinnit ja mikäli jokin tietoluokka tarvitsisi nk. puhdistaa ennen seuraavaa reitityskyselyä loin sille puhdistusmetodit. Koodista siivottiin myös kaikkien ei enää käytettyjen kirjastojen import –komennot ja ylimääräiset tai hyödyttömät ”debug” –viestit siivottiin pois.

Sovelluskoodin osittaisen uudelleenkirjoituksen ansiosta palvelun suorituskyky parani oleellisesti, testitapauksesta riippuen 20-30%. Sekä Stage- että kehitysympäristössä havaittiin 25% suorituskykyloikka 8 lähetyssovelluksen tapauksessa ja noin 30% parannus 16 lähetyssovelluksen tapauksessa.

Kehitysympäristö: 1 työasema (1x Intel Xeon E5530 @ 2.4Ghz 4C/8T)

- 8 lähetyssovellusta: 207 req/s → 258req/s (+25%)
- 16 lähetyssovellusta: 293 req/s →375req/s (+28%)

Stage -ympäristö: 2 palvelinta (2x Intel Xeon E5-2609 @ 2.4Ghz 4C/4T)

- 8 lähetyssovellusta: 234 req/s → 278req/s (+20%)
- 16 lähetyssovellusta: 434 req/s →564 (+30%)

Testauksen yhteydessä ympäristöjen välillä oli havaittavia eroja CPU –resurssien käytössä. Kehitysympäristön työasema käytti CPU –ajasta noin 12% käyttöjärjestelmän ”system space” –aikaan kun Stage –ympäristössä vastaava luku oli 35-40% (8-16 lähettäjää). Johtopäätöksenä ensimmäisestä suorituskyvyn optimointiyrityksestä totesin, että sovelluskoodin uudelleenkirjoittaminen vaikutti suuresti suorituskykyyn, mutta Stage- ja kehitysympäristöjen välisellä erolla oli välttämättä jokin Stage – ympäristöä haittaava ero, olihan Stage -ympäristössä laskennallisesti noin 4 kertaa enemmän tehoa kuin kehitysympäristössä.

Tässä vaiheessa päätin kuitenkin vielä keskittyä helposti havaittaviin optimointikohteisiin. Kummassakin ympäristössä saavutettiin testitapauksesta riippuen 17-20% suorituskykyparannus vaihtamalla Web-palvelin Apache2 ja FastCGI -moduulista CherryPy nimiseen Python web -palvelimeen. Lisäksi debug –tason lokituksen poiskytkeminen palvelun konfiguraatiotiedostosta lisäsi suorituskykyä noin 15%. Päätin kuitenkin tässä vaiheessa jättää molemmat optimoinnit tulevaisuuden varalle ja käyttää edelleen Apache2 -palvelinta ja pitää debug –lokit päällä. Tällä tavoin ympäristön käyttäytymistä olisi helpompi ymmärtää ja ongelmatapauksia selvittää lokitiedoista. [21]

Miettiessäni suorituskykyongelmaa sovelluskehitysympäristön ja Stage –ympäristön välillä tulin ajatelleeksi kuinka ympäristöt on rakennettu fyysisesti. Molemmat ympäristöt käyttävät sen verran nykyaikaista x86 –laitteistoa, että jokainen prosessori sisältää oman muistiohjaimen josta on suora yhteys muistikampoihin (tai muistipiireihin). Toisin sanoen Stage –ympäristössä olisi mahdollista, että NUMA (Non-Uniform Memory Access) voisi aiheuttaa ongelmia, mikäli suoritin 1 joutuisi esimerkiksi hakemaan tietoa suorittimen 2

hallitsemista muisteista.

Päätin ajaa tehokkuustestejä uudelleen ja seurata ”redis-server” prosessin käyttäytymistä. Täydellä kuormituksella testattaessa eroja ei havainnut, mutta ajaessa testejä vain 1-2 lähettäjäsovelluksella havaitsin Stage –ympäristön käyttöjärjestelmän siirtävän Redis:in palveluprosessia fyysiseltä prosessorilta toiselle. Päätin vielä tutustua Intelin dokumentaatioon NUMA:sta ja oletukseni ongelman ymmärtämiseksi vahvistuivat. [22], [23]

Päätin kokeilla tehokkuustestejä pakottamalla Stage –ympäristön palvelimella ”redis-server” –prosessia eri suorittimille ja suoritinytimille. Päivän testaamisen jälkeen tulin tulokseen, että oikeastaan mikä tahansa muu suoritinydin paitsi ydin numero 0 soveltuu parhaiten korjaukseksi, myös ydin 0 soveltuisi, mutta ei tarjoasi parasta mahdollista suorituskykyä. Ilmeisesti Linux käyttöjärjestelmän kernel varaa joitain toimia itselleen suoritinydin numero 0:lle tai vaihtoehtoisesti suosii kyseistä suoritinydintä ajanjaossa. Tässä vaiheessa on syytä huomioida, että Redis –palvelinprosessi suorittaa tehtäviä vain yhdellä säikeellä, eikä siis voisi kuormittaa muita kuin pakotettua suoritinydintä tai useita sallittuja ytimiä samalla fyysisellä suorittimella.

Päätin pakottaa Stage -ympäristössä 1. palvelinnoodin Redis -huoltonoodin ensimmäisen suorittimen viimeiselle ytimelle (numero 3) ja Redis -orjanoodin toiseksi viimeiselle suoritinytimelle (numero 2) ja 2. Palvelinnoodilla ainoan Redis -prosessin ytimelle numero 2. Tällöin käynnistysskriptiin asetettu ”taskset” -komento toimisi sekä yhden että kahden suorittimen palvelimissa, ja koska kaikki palvelimemme ovat vähintään 4-ydin suorittimilla varustettuja, niin ytimet 0-3 löytyisivät jokaiselta palvelimelta aina.

Redis -prosessin pakottaminen tietylle prosessorille paransi palvelun suorituskykyä testitapauksesta riippuen 30-35% Stage –ympäristössä, kehitysympäristössä suorituskyvyssä ei ollut eroa. Tulokset on esitettynä taulukossa 3 optimointien eri vaiheissa sekä saavutettu suorituskyky vaihtoehtoisella palvelinohjelmalla (CherryPy) ja kevyemmällä lokituksella. Tulokset kuvattu kumulatiivisesti siten, että alemman rivin uusi optimointi on yhdistetty ylemmän rivin optimoinnin kanssa.

Asiakassovelusten lukumäärä	Optimointi (kumulatiivinen, alempi rivi sisältää ylemmän rivin optimoinnin)	Suorituskyky (kyselyä / sekunti)	Parannus edelliseen
8	Ei optimointia	234	-
16	Ei optimointia	434	-
8	Koodin uudelleenkirjoitus	283	21%
16	Koodin uudelleenkirjoitus	564	29%
8	Redis ”taskset”	365	29%
16	Redis ”taskset”	761	35%
8	CherryPy	427	17%
16	CherryPy	905	19%
8	Debugloki kytketty pois	496	16%
16	Debugloki kytketty pois	1097	21%

Taulukko 4. Optimointien tehokkuusparannukset asteittain.

Kaikilla käytössä olevilla optimoinneilla ei kuitenkaan vieläkään saavutettu alunperin vaadittua nopeutta 1400 kyselyä/sekunti. En ollut kuitenkaan huolestunut suorituskyvystä, koska optimointia tehdessäni olin havainnut koodissa vielä useita parannettavia kohtia. En kuitenkaan jatkanut optimointia pidemmälle tässä vaiheessa koska tuotteistuksen aikaraja lähestyi. Projektin taholta ei myöskään suoranaisesti tiputettu suorituskykyvaatimuksia, mutta suorituskyvyn nostosta ei enää puhuttu yhtä aktiivisesti palavereissa. Samaan aikaan NIP -alusta pystyi nykyisillä resursseilla käsittelemään noin 150 laskua sekunnissa, joka suoraan tarkoittaa ettei reitityskyselyitä voinut tulla kuin maksimissaan 150 kyselyä/sekunti.

5.8 Palvelun laadun testaus ja tuotteistus

Palvelun laadun ja vastausten oikeellisuuden testaamisen ja todentamisen avuksi päätin luoda työkalun jolle annettaisiin esimääriteltyjä reitityskyselyitä, testaustyökalu lähettäisi kyselyt sekä reititysvälimuistipalveluun että tuotannossa olevaan RLU -reitityspalveluun ja vertailisi tuloksia. Työkalun tulisi ilmoittaa kaikista poikkeavuuksista jokaisen

reitityskyselyn tapauksessa, lisäksi työkaluun tulisi kevyt logiikka tiettyjen oletettujen virhetapausten tulkitsemiseen jolloin testejä ajavan henkilön olisi helpompi tulkita virheitä.

Päätin rakentaa testaustyökalun rungon melko yksinkertaisesti, työkalun pääkomponentit olivat seuraavat:

- Testaustiedoston lukija: Pystyi lukemaan tiedostosta testitapauksen nimen ja kuvauksen, sekä JSON muotoisen valmiin reitityskyselyn.
- Reititysvälimuistin koodista perityt Python objektit joilla voitiin muodostaa JSON reitityskyselystä ”dataolio” sekä ”dataoliosta” JSON reititysvastaus.
- Reititysvälimuistin POC koodista peritty SOAP kääntäjä joka osasi kääntää Python ”dataoliot” RLU -palvelun reitityskyselyiksi ja -vastauksiksi.
- Yksinkertainen HTTP -sanoman lähetys- ja vastaanottoluokka, joka osasi lisätä HTTP -kyselyyn merkkijonomuotoisen kuorman (payload).
- compareResponses –luokka joka osasi verrata kummankin palvelun vastauksia toisiinsa, huomioiden myös mahdolliset poikkeustilanteet.
- testManager –luokka joka lähetti jokaisen tiedostosta ladatun kyselyn molemmille palveluille ja parsi vastaukset ”dataolioksi” ja pyysi compareResponses –luokalta kyselyiden vertailun tulosta (tulos tai lista todetuista virheistä).

Esimääriteltyyn testitiedostoon kerättiin pakotetun reitityksen sekä normaalin reitityksen tapauksista tunnetut käyttötapaukset:

- Molemmat käyttötapaukset:
    - B2B
        - Reititys OpusCapitan asiakkaalle
        - Reititys asiakkaalle jonka operaattorina Pankki
        - Reititys asiakkaalle jonka operaattorina toinen operaattori
        - Reititys paperikanavaan (IPOST)
    - B2C
        - Kuluttajareititys Pankkiin (Finvoice)
        - Kuluttajareititys Pankkiin (DirectPayment)
        - Kuluttajareititys Netposti palveluun
        - Kuluttajareititys paperikanavaan

Normaalin reitityksen käyttötapauksen kohdalla huomioitavaa oli se, ettei reittien säilymistä voitu taata koska reititystieto päivittyi useita kertoja päivässä, käytännössä vastaanottajan operaattori tai vastaanottokanava voisi muuttua. Testaustyökaluun luotiin yksinkertainen logiikka joka hyväksyi operaattorin tai kanavan muutoksen jos molemmat reitit muuten pysyivät samana, kyseisessä tapauksessa testityökalu kuitenkin antoi varoituksen että reittitieto poikkesi oletetusta.

Testaustyökalun ajo 20-30 reitillä vei aikaa vain noin puoli minuuttia ja sen listauksesta näki nopeasti virheiden määrän. Tämän vuoksi työkalulla oli myös helppo varmistaa etteivät koodimuutokset aiheuttaneet virheitä reititykseen.

Alla esimerkkilistaus työkalun luomasta raportista.

```
Cache vs. RLU matching results -------------
 Queries loaded                     : 25
 Sender not found from DB           : 0
 Invalid input query                : 0
 Output is error message            : 1
 Matching results                   : 18
 Errors (routes dont match)         : 1
 Unexpected output operator/channel : 5
-----------------------------------------------
Found 2 Error(s) and 5 Warning(s)

Test failed!
```

Testityökalu helpotti virheiden löytämistä jonkun verran, mutta suurempi hyöty työkalusta saatiin kun palveluun tehtiin muutamia NIP -alustalle tarkoitettuja kustomointeja, kuten paperikanavan (IPOST) palautus aina kun reitityskyselyssä oli riittävän tarkat postiosoitetiedot. Testityökaluun luotiin lisää logiikkaa joka osasi havaita ja hyväksyä tilanteen jossa reititysvälimuisti palautti elektronisen reitin sekä paperikanavan reitin, vaikka alkuperäinen reitityspalvelu palautti vain elektronisen reitin.

Lisäksi palveluun luotiin toiminnallinen tuki Netposti reititykselle jonka toiminta pystyttiin varmentamaan testityökalun avulla. Tässä vaiheessa palvelu siis teknisesti tuki Netposti-reittien tunnistamista ja palauttamista, mutta tietokannassa ei oikeita Netposti-reittejä ollut, eikä palvelulla ollut vielä komponenttia joka olisi osannut Netposti-reittejä ladata Netposti:n palvelusta.

Testityökalu ja itse reitityspalvelu olivat tuotteistusvalmiita toukokuun lopussa 2014, tässä vaiheessa NIP -alustan ”Release 2” oli valmiina pilottiasiakkaiden käyttöönottoon, mutta käyttöönotto päätettiin siirtää kesälomakauden yli syksyyn. Kesäkauden aikana lisäilin reititysvälimuistipalveluun ominaisuuksia jotka eivät alkuperäiseen MVP-ajatteluun kuuluneet (Minimum Viable Product). Kesän aikana implementoin reititysvälimuistipalveluun sekä ”CacheManager” –komponenttiin tuen operaattoreiden vaihtoehtoisille kanavakoodeille. Yleisesti ottaen operaattorilla on vain yksi tunniste (esim. OVT) mutta osalle operaattoreista on määritelty useampia tunnisteita, käytännössä palvelu vain yhdisti sekundaarisen tunnisteen operaattorin primääritunnisteeseen ja sen jälkeen teki reitityskyselyn käyttäen primääritunnistetta, eli ylikirjoitti reitityskyselyssä olevan sekundaarisen tunnisteen tietokannasta haetulla primääritunnisteella.

Lisäksi kesälomakauden aikana toteutin POC (Proof-of-Concept) –tasoisen Netposti-reittien lataus- ja tallennusmekanismin. Tuotannossa olevista reitityspalvelun kysely- ja vastauspareista hain sopivaa testidataa jonka avulla kokeilin reititysvälimuistin Netposti-reittien latausta, tallennusta ja reitityskyselyiden toimintaa. Periaatteessa Netposti reititys oli toimivassa kunnossa kesälomakauden lopussa, mutta ei varsinaisesti vielä laadultaan tuotannon vaatimaa tasoa.

6 PALVELUN TUOTTEISTUS

Kappaleessa kuvaillaan sovelluskehitysprojektin aikataulun muuttumista uudelleen NIP -alustan arkkitehtuurin muutosten vuoksi ja reititysvälimuistipalvelun projektille saadaan lisää aikaa. Aikataulumuutoksista johtuen Netposti -palvelu saadaan tuotteistettua ennen palvelun käyttöönottoa.

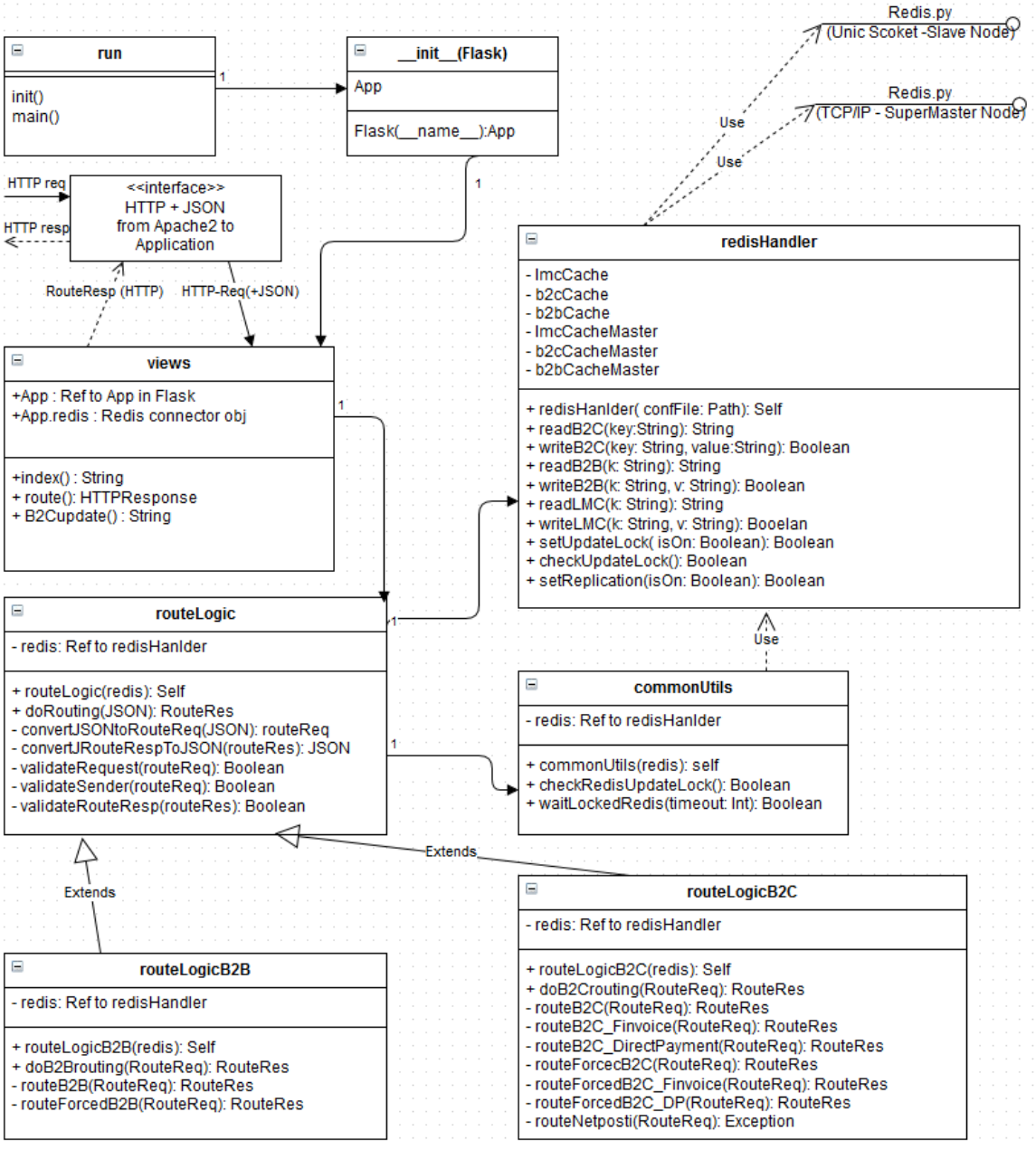
6.1 Syksy 2014: Projektin aikataulu muuttuu

Kesälomakauden jälkeen NIP projektissa tehtiin arkkitehtuurin katselmointi ja päätettiin muutoksista ja parannuksista NIP:n arkkitehtuuriin joitakin palvelun alikomponentteja tai alipalveluita muokattaisiin. Muutostöistä johtuen NIP:n tuotteistus siirtyi joulukuuhun (12/2014), muutoksesta johtuen myös reititysvälimuistin kehitykselle saatiin lisää aikaa.

Palvelun laatuvaatimusten kannalta koodi oli jo riittävän hyvällä tasolla, mutta muun muassa Redis -yhteyskirjasto ja reitityslogiikka olivat osiltaan toisistaan riippuvaisia. Esimerkiksi Redis -yhteyskirjastossa oli muutamissa paikoissa selvästi reitityslogiikkaan kuuluvaa koodia. Päätin kirjoittaa osan palvelun koodista uudelleen sekä siistiä olemassa olevaa koodia jatkokehittämisen ja ylläpitämisen helpottamiseksi.

Koodin uudelleenjärjestelyn ja parantelun yhteydessä irrotin Redis -yhteyskirjastosta kaikki reitityslogiikkaan liittyvät koodin pätkät ja siirsin ne itse reitityslogiikkaa sisältävään moduuliin, tämän jälkeen pilkoin vielä reitityslogiikan moduulin loogisesti B2B ja B2C –reititysosuuksiin joiden alle pilkoin vielä molempiin tapauksiin pakotetun reitityksen ja normaalin reitityksen osuudet selvästi omiksi osa-alueiksi ja metodeiksi.

Sovelluskoodin uudelleenkirjoittamisen jälkeen reititysvälimuistipalvelun arkkitehtuuri oli muuttunut alla olevan kuvan (8) mukaiseksi.



Kuva 8. Palvelun arkkitehtuuri sovelluskoodin uudelleenkirjoituksen jälkeen.

Sovelluksen osittaisella uudelleenkirjoituksella ei ollut havaittavia vaikutuksia palvelun tehokkuuteen tai nopeuteen, mutta myöhempää jatkokehitystä ajatellen koodin ylläpidettävyys oli erittäin paljon parempi, koska suuret monoliittiset yhden ominaisuuden metodit pilkottiin pieniin toisistaan riippumattomiin metodeihin joita ketjuttamalla itse ominaisuudet saatiin aikaan.

6.2 Netposti-reititys

Netposti-reititystä ei valitettavasti voitu toteuttaa siten että reitit voitaisiin kysellä Netpostista niin sanotusti lennossa (”run-time”), koska Netposti:n rajapinta on suunniteltu sellaiseksi, että yhden kyselysanoman sisällä lähetetään useita lähettäjä-vastaanottajatietoja. Netposti -palvelun kehittäjät ja ylläpitäjät suoraan kielsivät lähettämästä yksittäisiä kyselyitä nopeasti. Tästä johtuen reititysvälimuistipalvelu suunniteltiin siten, että kaikista kuluttajalaskuista jotka reititysvälimuistipalvelun kautta välitettiin, tallennettiin lähettäjän ja vastaanottajan tiedot Redis:in omaan tietokantaindeksiin.

Tämän tietokantaindeksin tarkoituksena on toimia väliaikaisena varastona Netposti-kyselyiden luontiin. Tietueille asetettiin TTL (Time To Live) arvo 90 vuorokautta. Redis:in TTL toimii siten, että TTL asetetaan aina uudestaan, jos sama tietue tallennetaan uudelleen. Tällöin siis jokaisen mahdollisen Netposti vastaanottajan tietoja kyseltäisiin vähintään noin kolmen kuukauden ajan uudelleen reittien päivittämiseksi.

Seuraavana yönä eräajotyyppisesti kaikki nämä väliaikaisesti tallennetut lähettäjä-vastaanottajaparit luettiin Redis:in tietovarastosta ja muunnettiin Netpostin määritelmän mukaisiksi SOAP -sanomiksi, siten että jokaiseen sanomaan tulisi maksimissaan 10000 Netposti-reititykselyä ja ensimmäisen sanoman rakentamisen jälkeen sanoma lähetettiin Netposti Matcher –palveluun. Palvelun antamasta vastauksesta parsittiin jokainen vastaanottaja, ja niiden tiedot joilla oli valittuna välitystapana ”Netposti” tai ”Netposti kopio” tallennettiin Redis:in hyvin samalla tavalla kuin B2C –reitit.

Tallennettujen reittien TTL -arvoksi asetettiin kolme vuorokautta joka sovittiin Netposti-kehittäjien antamien tietojen ja ohjeiden perusteella. Netpostissa asiakkaan vastaanottoasetukset päivittyvät ”Netposti Matcher” -palveluun vuorokauden välein tämän lisäksi Netpostin toiminta on määritelty siten että Netposti vastaanoton sulkemisen jälkeen Netpostiin saa välittää ko. vastaanottoon liittyviä laskuja tai kirjeitä vielä 2 vuorokautta vastaanoton sulkemisen jälkeen.

Tällä tavoin meidän reittimme elinaika vastaisi Netposti palvelun määritelmiä, mikäli asiakas sulkisi Netposti -vastaanoton, niin seuraavana yönä Netposti Matcher palvelu ei enää palauttaisi meille Netposti -reittiä kyseiselle lähettäjä-vastaanottajaparille, jolloin reitin TTL ei enää päivittyisi ja Redis automaattisesti TTL:n loputtua poistaisi reitin tietokannasta jonka jälkeen materiaalia ei enää Netpostiin välitettäisi.

7 PALVELU TUOTANTOKÄYTÖSSÄ

Seitsemännessä kappaleessa kuvaillaan palvelun tuotannon käyttöönottoa sekä tuotantokäyttöä, ja kuvaillaan ylläpitotoimista johtuneita palvelukatkoja sekä sovellusvirheitä. Lisäksi kuvataan palvelun tuotantoympäristö sekä palvelun suorituskyky tuotannon aloitukseen allokoiduilla resursseilla.

7.1 Käyttöönotto

Reititysvälimuistipalvelun tuotantokäyttö alkoi vuoden 2015 alussa vaikka reititysvälimuistipalvelua käyttävä alusta oli tuolloin vielä pilottituotannossa. Järjestelmään oli luotu muutaman pilottiasiakkaan laskujuoksut pilotointia varten. Pilottiasiakkaiden laskutusta ajettiin ns. monistetusti rinnakkain vanhassa sekä uudessa järjestelmässä ja molempien järjestelmien laskujuoksujen lopputulosten mahdollisia eroavaisuuksia valvottiin koneellisesti. Koneellisen valvonnan raportoimat poikkeukset tutkittiin ja poikkeusten alkuperä selvitettiin.

Uuden alustan pilottituotannon aikana asiakastoteutuksista vastaava tiimi myös perusti uusia asiakkaita järjestelmään jolloin materiaalimäärät sekä erilaisten käyttötapausten varianssi kasvoi. Pilottituotantoa jatkui pitkälle syksyyn 2015, jonka aikana reititysvälimuistipalvelu aiheutti muutamia ongelmatapauksia ja virhetilanteita joista sovelluskehitykselle raportoitiin. Reititysvälimuistipalveluun liittyviä tukipyyntöjä tuona aikana tuli useita ja suurin osa pyynnöistä oli lähinnä käyttäjätukea, liittyen erilaisiin erikoiskäyttötapauksiin joita asiakastoteutustiimi kyseli luodessaan vanhan asiakastoteutuksen kopiota uuteen järjestelmään. Pääasiassa käyttötapauksista kyseltiin senkin vuoksi, että järjestelmien välillä useissa ominaisuuksissa oli suuria eroja ja asiat tuli tehdä hieman toisin kuin vanhassa järjestelmässä. Niin sanotun käyttäjätuen lisäksi reititysvälimuistipalvelusta raportoitiin reititysvirhetapauksia (verrattuna vanhan järjestelmän antamiin reititystuloksiin) sekä muutama palvelukatko ongelma.

Vakavia palvelukatkoja tai sovellusvirheitä tuotantojakson aikana raportoitiin seuraavasti:

- Palvelukatkoja 4 kpl
- Sovellusvirheitä tai sovelluksesta johtuvia reititysvirheitä 8 kpl

Uuden ja vanhan järjestelmän erovaisuuksia reitityksen osilta löytyi todellisuudessa huomattavasti enemmän, mutta niitä ei raportoitu reititysvälimuistipalvelun tukeen, koska virheet tai eroavaisuudet johtuivat virheellisistä käsittelyistä jo laskumateriaalin kohdalla. Esimerkiksi laskumateriaalista poimittiin väärää dataa jonka perusteella reititys tehtiin ja jonka vuoksi se epäonnistui.

7.2 Palvelukatkot

Reititysvälimuistipalvelun palvelukatkot johtuivat pääsääntöisesti virheellisestä palvelin- ja käyttöjärjestelmäylläpidosta. Kahden palvelukatkotapauksen syyt saimme selvitettyä, mutta toiset kaksi jäivät osittain mysteeriksi, emmekä myöskään saaneet koskaan selvää vastausta konesalipalvelujen tarjoajalta näihin tapauksiin.

Molempiin tapauksiin selvästi kuitenkin liittyi jokin virtuaalipalvelinten ylläpitotoimi ja sen epäonnistuminen. Molemmissa tapauksissa reititysvälimuistipalvelun lokitiedoista nähtiin kuinka käyttöjärjestelmän kello yllättäen oli hypännyt taaksepäin yli puolella vuorokaudella. Kellon hyppy näytti lamauttavan palvelun toiminnan, ongelmaksi osoittautui Redis tietokanta, mutta en varsinaisesti saanut tarkemmin selville miksi Redis lopetti palvelemisen kellon siirtyessä taaksepäin. Voin vain olettaa kellon taaksepäin siirtyessä Redis:in sisäisten ajastuslaskurien sekoavan. Redis laskee sisäisesti avain-arvo parien TTL (Time To Live) arvoa automaattisesti, lisäksi Redis:ille määritellään sisäinen toimintataajuus jonka puitteissa Redis yrittää ajaa omaa pääsilmukkaa (main loop). Vakiona asetus on 10Hz (hertsi) jota en muuttanut koskaan.

Yhdessä tapauksessa käyttöjärjestelmän päivitysten asentamisen yhteydessä ylläpito ei ollut noudattanut palvelun pysäytys- ja käynnistysohjeita, jolloin reititysvälimuistipalvelun Web-palvelut oli käynnistetty ohjeiden mukaan, mutta Redis – tietokannat oli jätetty sammuksiin. Web-pavelu ilmoitteli onnistuneesti uudelle järjestelmälle ongelmaksi tietokantojen alhaalla olon, mutta virheilmoitusta ei varsinaisesti käsitelty vaan pelkästään kirjoitettiin lokiin.

Toinen selvitetty palvelukatko johtui ohjeistuksen vastaisesta Redis-konfiguraatiosta sekä käyttöjärjestelmän ylläpitotoimista. Redis:in ”Unix Socket”-tiedosto oli virheellisesti

asetettu konfiguraatiotiedostossa käyttöjärjestelmän väliaikaistiedostojen kansioon (/tmp), konesaliylläpito asensi palvelimelle joitakin päivityksiä jonka jälkeen he olivat putsanneet tmp –kansiosta sokkona kaikki tiedostot; kun socket –tiedosto katosi ei Web-palvelin saanut enää yhteyttä paikalliseen Redis –tietokantaan jolloin palvelu ei voinut enää vastata reitityskyselyihin ja palautti vain virheilmoitusta. Korjaavana toimenpiteenä socket-tiedosto siirrettiin hakemistoon ”/var/run”.

7.3 Sovellusvirheet

Kaksi ensimmäistä sovellusvirhettä liittyivät Netposti reittien latauksen toteuttavaan NetpostiMatcherClient –sovellukseen, osa Netposti reittien latauksista epäonnistui sovelluksen luodessa SOAP -standardin vastaisen kyselyn. Ongelma johtui merkistökoodauksista sekä muutamien kiellettyjen merkkien sanitoinnin puutteesta. Merkistökoodaus saatiin kohdilleen määrittämällä käyttämälleni SOAP –kirjastolle haluttu merkistö, mutta jostain syystä merkkijonojen sanitointi edelleen epäonnistui muutamien erikoismerkkien kohdalla. Toteutin XML -kontrollimerkkien (’&’, ’<’, ’>’, ”” ja ’’’) sanitoinnin omassa sovelluskoodissani ja ongelma saatiin korjattua. Tämä ongelma havaittiin jo sisäisessä testauksessa pilottiasiakkaalla jonka oikea tuotanto edelleen kulki vanhan järjestelmän läpi, joten ongelmia ei aiheutunut ulkopuolisille tahoille.

Palvelun kahdennukseen liittyvä sovellusvirhe löydettiin samaisen pilottiasiakkaan laskujuoksujen testauksessa. Sovelluskoodissa oli jäänyt huomioimatta tilanne jossa Redis huoltonoodi (SuperMaster) sammutettiin. Tällöin palvelu ei voisi päivittää välimuistiin kuluttajalle lähetetyn laskun viitenumeroa eikä myöskään kirjoittaa mahdollisia postiosoitetietoja välimuistiin jonka perusteella päivitettäisiin Netposti reittejä yöllisissä eräajoissa. Palvelu kirjoitti virheet lokiin onnistuneesti, mutta Redis huoltonoodiin tiedot kirjoittavasta metodista oli unohtunut yhteysongelmien varalta try-catch –rakenne, luonnollisesti sovellus kaatui virheeseen ja WWW-palvelin palautti HTTP 500 virhettä eikä palauttanut reititysvastausta.

Loput viisi reititysvirhettä liittyivät palvelun omaan tietokantakirjastoon jolla B2B- ja B2C-reitit ladattiin RLU -palvelun tietokannasta reititysvälimuistin tietokantaan. Joissain ongelmatapauksissa RLU -palvelun tietokantaan oli jäänyt tietylle reititysavaimelle (Y-

tunnus/OVT) useita reititysrivejä joista kaikki olivat aktiivisessa tilassa, tällaisessa tapauksessa B2B reittejä lataava komponentti saattoi tallentaa väärän reitin välimuistiin, ongelma korjattiin muuttamalla tietokantamoduulia siten että usean reittitiedon tapauksessa se lataisi tiedoista uusimman.

Samalla havaitsin mahdollisen ongelman lähettäjäkohtaisten pakotettujen (”private route override”) reittien lataamisessa ja korjasin sen samalla tietokantakirjastoon. Emme voineet kuitenkaan testata tätä reititystä pilottiasiakkaan materiaalilla, koska pilottiasiakas ei tällaista ominaisuutta käyttänyt, joten loimme testiasiakkaan ja testimateriaalin itse käsin testausta varten.

B2C reititysten latauksen puolelta löytyi kaksi reittien latausongelmaa ja yksi reitityslogiikan ongelma itse reititysvälimuistipalvelusta. B2C -reittien (kuluttajareittien) lataus epäonnistui norjalaisille lähettäjille joiden laskuja välitettiin Norjan pankkiverkkoon.

Ongelma johtui B2C –reittien latauskomponentista jossa tietokannasta löytyvää dataa koitettiin validoida suomalaisten sääntöjen mukaan. Norjan kuluttajalasku rekisteröinneistä ei löydy kuluttaja-asiakkaan pankkitilin IBAN –osoitetta, IBAN:in puuttuessa B2C reittien latauskomponentti hylkäsi reittitiedon puutteellisena. Toinen B2C -reittien ongelma liittyi kuluttajan reitin päivittämiseen laskun lähettäjän toimesta RLU -palvelun webkäyttöliittymästä. B2C -reittejä päivittävään sovellukseen luotiin uusi päivitysmenetelmä joka etsi tietokannan historia- ja muutostauluista reittimuutoksia jotka oli tehty RLU WebUI -palvelun kautta eikä RI-import komponentin kautta.

7.4 Palvelun tuotantoympäristön lyhyt kuvaus

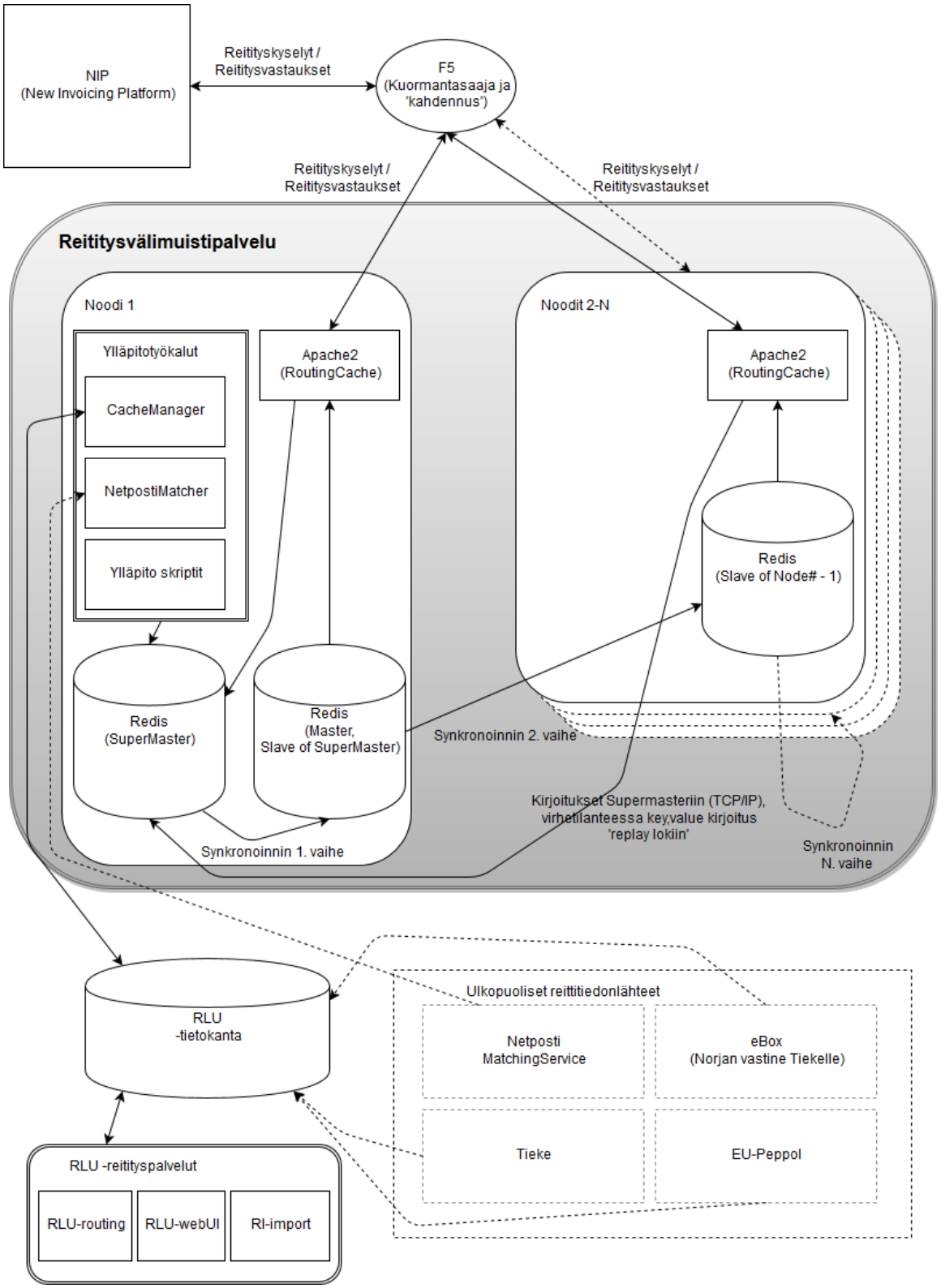
Palvelun pilottituotanto ja varsinainen tuotanto aloitettiin huomattavasti matalammilla suorituskykyvaatimuksilla kuin projektin alussa määritellyt 1400 kyselyä sekunnissa, koska alussa näin suurelle kapasiteetille ei ollut käyttöä. Tuotanto aloitettiin kahdella virtuaalipalvelimella joihin allokoitiin molempiin 2 suoritinydintä ja 32 Gt muistia. Virtuaalipalvelimia pyörittävä palvelin käytti Intel Xeon E7-4870 suorittimia, jotka toimivat 2,4Ghz taajuudella, rautatasolla suorittimet tukivat Hyper-Threading ominaisuutta, mutta ominaisuuden hyödyistä virtualisoidulla alustalla ei pysty tekemään

johtopäätöksiä virtualisoidun käyttöjärjestelmän tarjoamilla työkaluilla.

Tuotantopalvelimien yhdistetyksi suorituskyvyksi kuormantasaajan kautta mitattiin 600-650 kyselyä sekunnissa, mittaustulokset tuntuivat heiluvan jatkuvasti näiden rajojen välissä. Mittaustulosten kohtuullisen pienestä heilunnasta voitaneen todeta virtualisointiympäristön resurssien vastaavan kohtuullisen hyvin kuormitukseen ja tarpeisiin, koska mittaustulokset pysyivät samanlaisina eri vuorokauden aikoihin ja jokaisella mittauksella varianssi pysyi samana.

Tarvittaessa palvelua voitaisiin nyt skaalata kahdella tavalla, mikäli suorituskykyä tarvitsisi nostaa nopeasti, voitaisiin virtuaalipalvelimiin lisätä nopeasti suoritinytimiä. Pidemmällä aikavälillä palvelun suorituskykyä voidaan nostaa lisäämällä palveluun lisää orjanoodeja jolloin suorituskyvyn pitäisi kasvaa lineaariesti noodien määrän lisäyksen mukaan, koska noodit eivät ole riippuvaisia toisistaan lukuoperaatioiden osalta.

Alla olevasta kuvasta (kuva 9) nähdään rakennetun reititysvälimuistipalvelun järjestelmäkuva ns. loogisena kokonaisuutena. Järjestelmiä kuvaavien laatikoiden koot eivät kuvaa järjestelmien kokoa vaan esittää niiden kytkeytymistä toisiinsa. Sovelluksena esimerkiksi NIP -alusta on koodilliselta kooltaan yli 100-kertainen verrattuna reititysvälimuistiin. Kuvassa on katkoviivoin mallinnettu palvelunoodit 3-N jotka voidaan tarvittaessa rakentaa tulevissa kapasiteetin lisäyksissä.



Kuva 9. Reititysvälimuistipalvelun yleiskuva.

8 YHTEENVETO

Reititysvälimuistin sovelluskehitysprojekti osoittautui erittäin haastavaksi lukuisista ongelmista ja muutoksista johtuen. Jo projektin alussa havaittiin että puhtaan välimuistitoteutuksen tekeminen ei sellaisenaan onnistuisi, minkä vuoksi ryhdyttiin rakentamaan liiketoimintalogiikkaa sisältävää nopeaa reitityspalvelua, joka kuitenkin sinällään toimi eräänlaisena välimuistitasona reititystiedolle RLU:n tietokannassa.

Sovellusteknisten ongelmien lisäksi projektiin vaikuttivat ulkoiset tekijät, kuten projektin keskeytys kahdesti uusien asiakasominaisuuksien pystyttämiseksi ja tuotannon suorituskykyongelmien ratkonnan ajaksi. Tämän lisäksi NIP -projektin sisältö- ja aikataulumuutokset vaikuttivat reititysvälimuistiprojektiin, joten kaikin puolin toimintaympäristö oli haastava.

Haastavan projektin ainoaksi lopputuotteeksi ei jäänyt vain reititysvälimuistipalvelu, itse palvelun lisäksi yritykseen uudelleen perustettu sovelluskehitysyksikkö sekä muut ympäröivät yksiköt kuten tuotehallinta kehittyivät NIP -projektin edetessä. Selvää kehitystä tapahtui niin yksiköiden välisessä yhteistyössä kuin jokaisen yksikön ns. sisäisissä toimissa.

Lyhyesti voidaan tehdä seuraava yhteenveto projektin aikana opituista asioista.

- Valtavan kokoisia sovelluskehitysprojekteja tulisi suunnitella paremmin vaikka käytettäisiin niin kutsuttuja ketteriä menetelmiä, NIP -projektin useat aikataulumuutokset vaikuttivat ikävästi reititysvälimuistiprojektiin.
- Yrityksen kommunikaatio on avaintekijä sovelluskehitysprojektin onnistumisen kannalta, jatkuvat aikataulumuutokset ja kehitysprojektin keskeyttävät muut työt ovat hyvin haitallisia.
- Suljettuihin järjestelmiin liittyvät kirjastot voivat aiheuttaa yllättäviä ongelmia toisistaan eroavilla julkaisuilla samasta käyttöjärjestelmästä, vaikka käytössä olisi avoimen lähdekoodin ohjelmointikieli, joka normaalisti toimii näillä alustoilla niin sanotusti suoraan.
- Liian yksinkertainen tietovarasto voi aiheuttaa lisätöitä sovelluskoodin puolella, Redis:iä ei tule käyttää tämän tapaisessa palvelussa jossa sovelluskoodissa

joudutaan tekemän paljon liiketoimintalogiikkaa.

- Reititysvälimuistin arkkitehtuuri olisi ollut parempi toteuttaa erillisillä palvelimilla tietovarastolle ja itse palvelulle, koska palvelun liiketoimintalogiikka vaati huomattavasti enemmän resursseja kuin itse tietovarasto.
    - Palvelun arkkitehtuuri voidaan tarvittaessa kohtuullisella työmäärällä muuttaa tulevaisuudessa edellä mainitun mukaiseksi
- Projekteissa tulisi varata riittävästi aikaa POC -toteutuksen tekemiseen ja mahdollisesti tuotantopalvelun tekemiseen alusta asti kokonaan uudelleen, mikäli POC -toteutus ei sellaisenaan sovellu käyttötapaukseen.

LÄHTEET

[1] **Redis**
http://redis.io
[12.08.2013]

[2] **F5 Big IP – load balancer and High Availability**
https://f5.com/products/big-ip
[12.08.2013]

[3] **python™ - Global Interpreter Lock**
https://wiki.python.org/moin/GlobalInterpreterLock
[12.08.2013]

[4] **Django vs Flask vs Pyramid: Choosing a Python Web Framework**
https://www.airpair.com/python/posts/django-flask-pyramid
[13.08.2013]

[5] **Django**
https://www.djangoproject.com/
[13.08.2013]

[6] **CherryPy**
http://www.cherrypy.org/
[13.08.2013]

[7] **Flask**
http://flask.pocoo.org/
[13.08.2013]

[8] **The Flask Mega-Tutorial**
http://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world
[14.08.2013]

[9] **RAM SSDs versus flash SSDs? - (aspects of asymmetries in SSD design)**
http://www.storagesearch.com/ssd-ram-v-flash.html
[14.08.2013]

[10] **IBM Knowledge Center: DB2 CREATE TRIGGER STATEMENT**
http://www-01.ibm.com/support/knowledgecenter/SSEPGG_9.1.0/com.ibm.db2.udb.admin.doc/doc/r0000931.htm
[14.08.2013]

[11] **Replication - Redis**
http://redis.io/topics/replication
[20.08.2013]

[12] **Intel(R) Hyper-Threading Technology**
https://software.intel.com/en-us/intel-hyper-threading-technology-your-questions-answered
[27.08.2013]

[13**] Performance Insights to Intel® Hyper-Threading Technology**
Submitted by Antonio Valles (Intel) on November 20, 2009
https://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology
[27.08.2013]

[14] **Maximized performance: Comparing the effects of Hyper-Threading, software updates**
http://www.extremetech.com/computing/133121-maximized-performance-comparing-the-effects-of-hyper-threading-software-updates
[27.08.2013]

[15] **Chapter 4. Compute Nodes - Overcommitting**
http://docs.openstack.org/openstack-ops/content/compute_nodes.html
[28.08.2013]

[16] **Issue #53 - ibm-db on Ubuntu**
http://code.google.com/p/ibm-db/issues/detail?id=53
[19.9.2013]

[17] **Ilmoittamispalvelun soveltamisohje**
http://www.finanssiala.fi/finvoice/dokumentit/Finvoice_2_1_soveltamisohje.pdf
(Huom. FKL:n jatkuvien sivustomuutosten takia linkki ei välttämättä toimi, päivitetty useita kertoja)
[24.09.2013, 04.03.2015, 28.2.2016, 03.04.2017, 29.04.2017]

[18] **The Effects of Virtualization on Main Memory Systems**
http://event.cwi.nl/damon2010/virtualization_effects.pdf
Martin Grund, Jan Schaffner, Jens Krueger, Jan Brunnert, Alexander Zeier
Hasso-Plattner-Institute at the University of Potsdam
August-Bebel-Str. 88
Potsdam, Germany
{martin.grund, jan.schaffner, jens.krueger, jan.brunnert,
alexander.zeier}@hpi.uni-potsdam.de
[07.10.2013]

[19] **KEYS – Redis documentation**
http://redis.io/commands/KEYS
[08.10.2013]

[20] **SLAVEOF – Redis documentation**
http://redis.io/commands/SLAVEOF
[10.10.2013]

[21] **CherryPy – Lightweight Web Server and Web FrameWork**
http://www.cherrypy.org/
[12.03.2014]

[22] **Learning Experience of NUMA and Intel's Next Generation Xeon Processor**
https://software.intel.com/en-us/blogs/2009/03/11/learning-experience-of-numa-and-intels-next-generation-xeon-processor-i
[12.03.2014]

[23] **Optimizing Applications for NUMA**
https://software.intel.com/en-us/articles/optimizing-applications-for-numa
[12.03.2014]

LIITE 1. Redis-bechmark –sovelluksen tulokset

Redis -bechmark tulokset ja CPU graafi käyttäen TCP/IP rajapintaa lokaalisti. Graafista nähtävissä suorituskyvyn rajoittuvan TCP/IP rajapinnan käsittelyihin, palvelinsovellus käyttää 74,6% saatavilla olevista resursseista kun samaan aikaan redis-bechmark sovellus kuormittaa koko suoritinytimen. (Sovellukset toimivat vain yhtä ydintä kuormittaen).

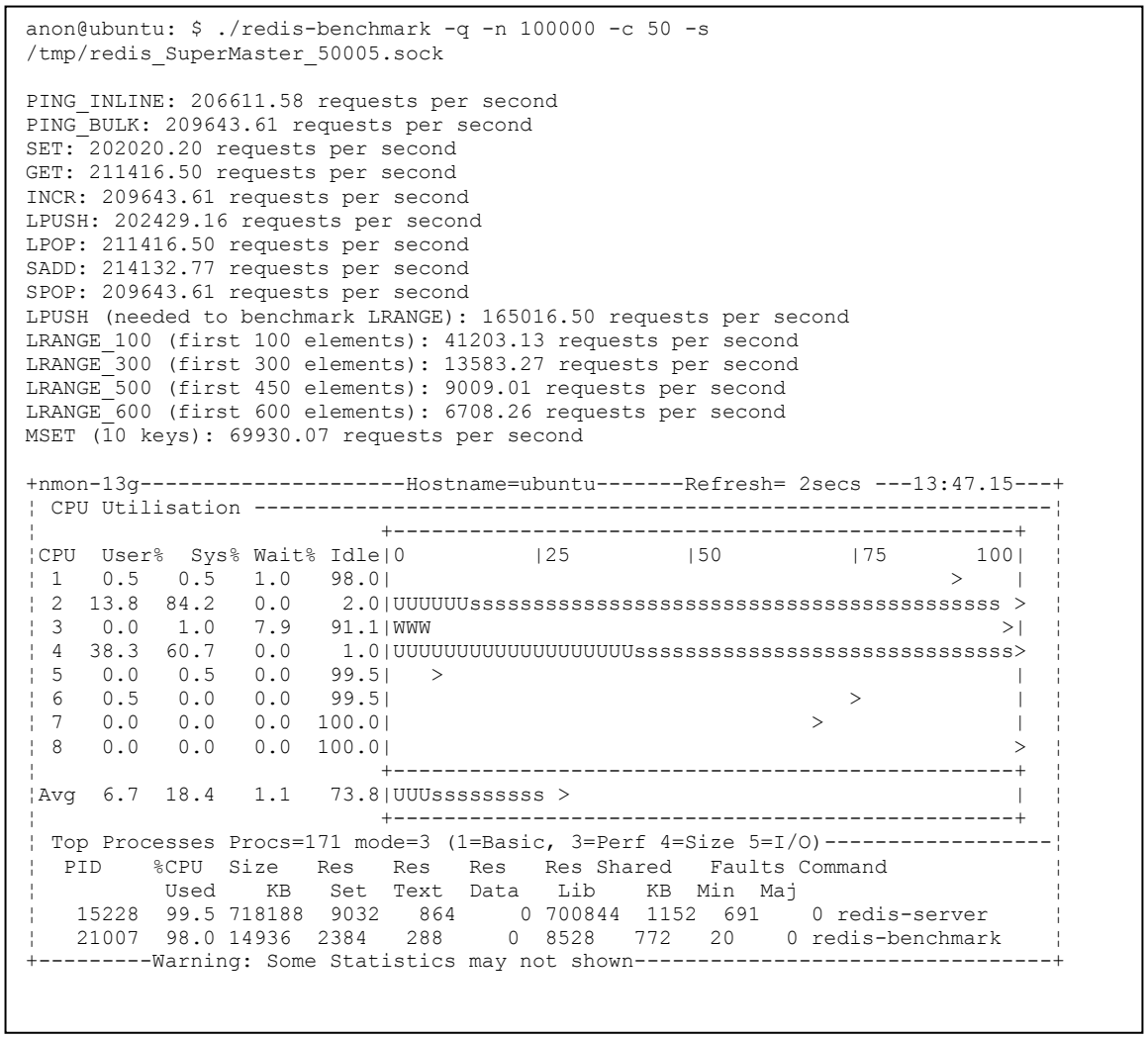
```
anon@ubuntu: $ ./redis-benchmark -q -h 127.0.0.1 -p 50005 -n 100000 -c 50
PING_INLINE: 85836.91 requests per second
PING_BULK: 85034.02 requests per second
SET: 89686.10 requests per second
GET: 128700.12 requests per second
INCR: 126262.62 requests per second
LPUSH: 126262.62 requests per second
LPOP: 129366.11 requests per second
SADD: 130548.30 requests per second
SPOP: 128040.97 requests per second
LPUSH (needed to benchmark LRANGE): 127551.02 requests per second
LRANGE_100 (first 100 elements): 48638.13 requests per second
LRANGE_300 (first 300 elements): 17373.18 requests per second
LRANGE_500 (first 450 elements): 12322.86 requests per second
LRANGE_600 (first 600 elements): 9562.06 requests per second
MSET (10 keys): 51282.05 requests per second
```

```
+nmon-13g---------------------Hostname=ubuntu-------Refresh= 2secs ---13:43.06---+
¦ CPU Utilisation -------------------------------------------------------------¦
¦                          +--------------------------------------------------+  ¦
¦CPU  User%  Sys% Wait% Idle|0          |25         |50          |75       100|  ¦
¦ 1   0.0   0.0   0.5   99.5|  >                                               |  ¦
¦ 2  56.3  17.4   0.5   25.8|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUssssssss         >  ¦
¦ 3   0.5   0.0   0.0   99.5|      >                                           |  ¦
¦ 4   0.0   0.5   0.0   99.5| >                                               |  ¦
¦ 5   0.0   0.0   0.0  100.0| >                                               |  ¦
¦ 6   0.0   0.5   0.0   99.5| >                                               |  ¦
¦ 7   0.5   0.5   0.0   99.0| >                                               |  ¦
¦ 8  83.7  16.3   0.0    0.0|UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUssssssss>  ¦
¦                          +--------------------------------------------------+  ¦
¦Avg 16.9   4.3   0.1   78.8|UUUUUUUUss >                                       |  ¦
¦                          +--------------------------------------------------+  ¦
¦ Top Processes Procs=171 mode=3 (1=Basic, 3=Perf 4=Size 5=I/O)------------------¦
¦  PID    %CPU  Size   Res   Res   Res   Res Shared   Faults Command              ¦
¦         Used    KB   Set  Text  Data   Lib    KB  Min  Maj                   ¦
¦   19894 100.0 14936  2908   288     0  8528   804   27    0 redis-benchmark    ¦
¦   15228  74.6 718188  9188   864     0 700844  1152  372    0 redis-server     ¦
+---------Warning: Some Statistics may not shown---------------------------------+
```

(jatkuu)

LIITE 1. (jatkoa)

Redis -bechmark tulokset ja CPU -graafi käyttäen Unix Socket –rajapintaa. Tuloksista nähtävissä molempien sovellusten kuormittavan omat CPU ytimensä lähes täysin (>=98%).



LIITE 2. Reititysvälimuistipalvelun reitityskyselyn ja -vastauksen tietosisältö.

Reititysvälimuistipalvelun JSON –reitityskyselyn tietosisällön esimerkki. Esimerkistä jätetty pois HTTP kutsun ”header” -elementti.

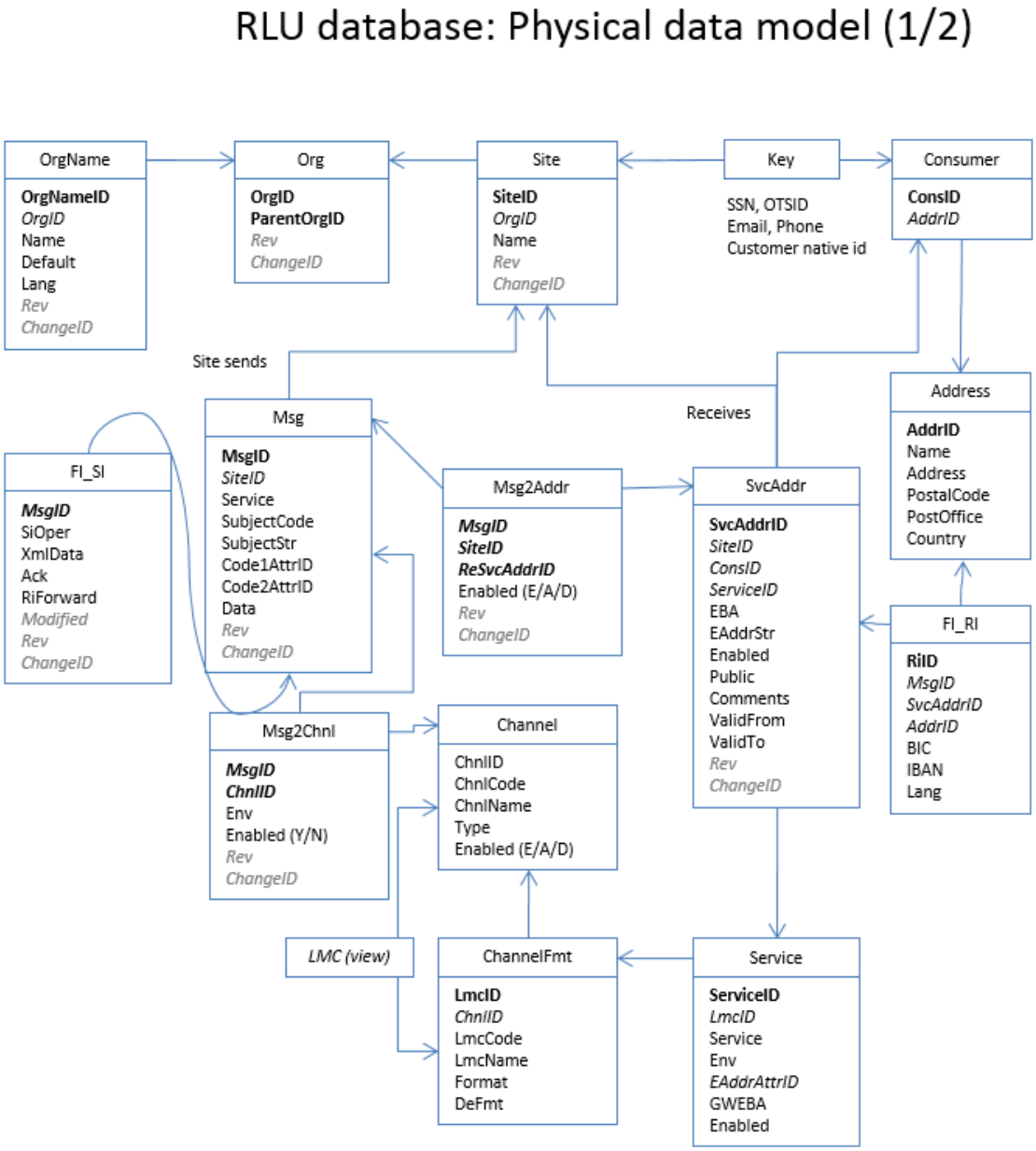
```
{
        "reqID": "1234",
        "sender": {
                "ID": "003701234567",
                "idtype": "EDIRA",
                "service": "invoice",
                "subject": "TESTILASKU",
                "B2C": "true"
        },
        "receiver": {
                "msgID": "BEL000DEMODEMODEMODEMODEMODEMODEMO1",
                "IDs": [{
                        "ID": "12345678",
                        "idtype": "#CID1"
                }],
                "name": "Teppo Testaaja",
                "addr1": "Testikatu 1 A 2",
                "pcode": "00100",
                "city": "Helsinki",
                "country": "FI"
        }
}
```

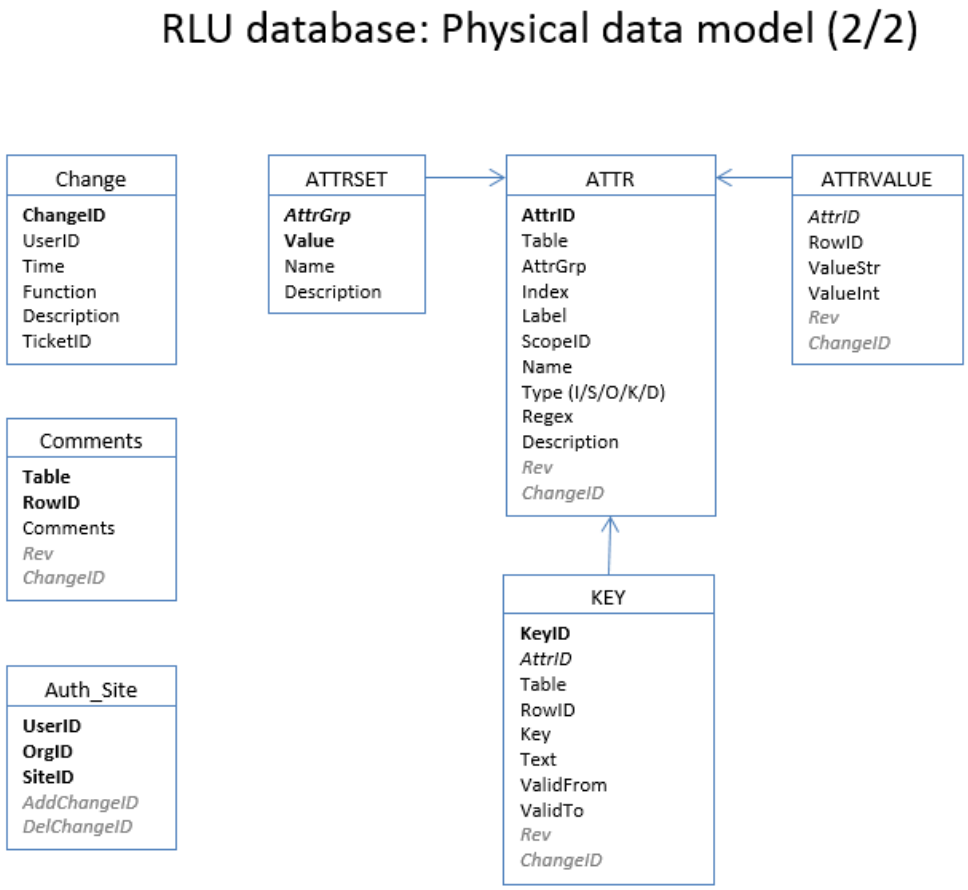
Reititisyvälimuistipalvelun JSON –reititysvastauksen tietosisällön esimerkki.

```
{
        "status": "OK",
        "reqID": "1234",
        "msgID": "BEL000DEMODEMODEMODEMODEMODEMODEMO1",
        "B2C": "true",
        "hdrlog": ["Electronic route found from redis cache"],
        "msglog": ["Test MSGLOG entry"],
        "routes": [{
                "eba": "BELRFI0000DEMO",
                "operator": "HANDFIHH",
                "format": "f2c",
                "eaddr": {
                        "addr": "FI-IBAN-ANONYM-001",
                        "operator": "HANDFIHH",
                        "format": "f2c",
                        "envaddr": "FI-IBAN-ANONYM-001@HANDFIHH/f2c"
                }
        }],
        "executionTime": "0.011048 seconds"
}
```

LIITE 3. RLU –palvelun tietokantakuvaus.

RLU -palvelun tietokantakuvaus. Kursivoidut harmaat arvot päivittyvät tietokantaan määriteltyjen ”triggereiden” kautta, käyttäjällä ei ole oikeutta muuttaa niitä.



(jatkuu)

LIITE 3. (jatkoa)