TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut

IDK40LT

Kaarel Kivistik 123665IAPB

KIIRSUHTLUSRAKENDUS MIKROTEENUS-ARHITEKTUURI BAASIL

Bakalaureusetöö

Juhendaja: Deniss Kumlander

Tehnikateaduste
doktor
Vanemteadur

Tallinn 2016

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Kaarel Kivistik

20.05.16

Annotatsioon

Antud töö käsitleb täisfunktsionaalse kiirsuhtlusrakenduse loomist ühe aina enam populaarsust koguva tarkvaraarhitektuuri – mikroteenuste – baasil.

Mikroteenused on uus lähenemisviis tarkvara paremaks loomiseks, mille võtmesõnadeks on paindlikkus, modulaarsus ja sõltumatus. Oma olemuselt tähendavad mikroteenused tarkvara ehitamist klotside abil ning klotsideks on antud juhul eraldiseisvad programmid, mis suhtlevad teineteisega rangelt avalikke liideseid pidi ja seega võivad kasutada vabalt valitud sisemist ehitust.

Töö annab ülevaate mikroteenuste teoreetilisest poolest, plussidest ja miinustest ning võrdleb neid kahe teise arhitektuuriga, milleks on monoliit-arhitektuur ja teiseks, mikroteenuste eelkäija – teenus-orienteeritud arhitektuur.

Töö lahkab mikroteenus-arhitektuuri disainimisel kerkivaid probleeme ning pakub neile lahendusi. Teiste seas on käsitletavateks probleemideks skaleeritavus, teenustevaheline suhtlemine, andmete säilitamine ehk need küsimused, mis tekivad esmakordsel kohtumisel mikroteenus-arhitektuuriga. Tuuakse välja ka konkreetsed tehnoloogiad, millega on võimalik nimetatud probleeme lahendada.

Töö tulemuseks on tarkvara, mille implementatsiooni detaile kirjeldatakse mikroteenus-haaval ehk mida see teeb, kuidas see töötab ja milline on avalik liides. Töö käigus loodud tarkvara on täies mahus autori poolt disainitud ja programmeeritud.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 31 leheküljel, 4 peatükki, 17 joonist, 6 tabelit.

Abstract

Microservice-based instant messaging application

Aim of this work is to create a fully functional instant messaging software application based on an increasingly popular architecture – microservices.

Microservices are a new approach to design software in a better way – some of the keywords include flexibility, modularity and independency. Using microservices in its essence means building software using blocks, which in this case are separate computer programs exchanging data strictly through public interfaces. This allows every microservice to use whichever programming language, additional software or implementation they prefer.

This thesis introduces microservices and its theoretical side – how it all started, what it is compared to the traditional architecture (monolith), pros and cons of microservices, how they are different from its predecessor – service-oriented architecture.

This work tries to analyse and solve the problems which software developers and architects may encounter on their first attempts at microservices. Problems that are being discussed include scalability, exchanging data between microservices and storing data. The author also proposes some possible technologies – software, protocols – which could be used to solve these problems. Resulting software is fully designed and programmed by the author.

The result of this work is a software application which is described in detail – what it does, how it works and what it exposes as its interface.

The thesis is in Estonian and contains 31 pages of text, 4 chapters, 17 figures, 6 tables.

Lühendite ja mõistete sõnastik

HTTP	*Hypertext transfer protocol*
URI	*Uniform resource identifier*
API	*Application programming interface*, programmiliides
AMQP	*Advanced message queue protocol*
REST	*Representational state transfer*
SOA	*Service oriented architecture*, teenus-orienteeritud arhitektuur
JSON	*JavaScript object notation*
SSL	*Secure sockets layer*
TTL	*Time to live*

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Tänapäeval on tarkvaraarenduses kanda kinnitamas uut tüüpi arhitektuur, milleks on mikroteenused. Mikroteenuste põhiprintsiip on ühe tervikliku rakenduse (monoliidi) jaotamine väiksemateks eraldiseisvateks osadeks ehk mikroteenusteks, mis suhtlevad teineteisega avalike programmiliideste kaudu, teadmata teineteise sisemist ehitust. Praeguseks pole olemas ühest, selgelt välja kujunenud praktikat sellise arhitektuuri realiseerimiseks [1].

Käesolevas töös on lahendatavaks probleemiks loodava tarkvara arhitektuuri disain. Selle probleemi lahendamiseks tuleb vastata järgmistele küsimustele:

- Kuidas ja millise põhimõtte alusel jaotada programm mikroteenusteks?
- Kuidas panna mikroteenused omavahel suhtlema?
- Milliseid protokolle kasutada?
- Kuidas tagada teenustevahelise suhtluse kiirus?
- Kuidas tagada skaleeritavus?
- Millist tarkvara kasutada?
- Milliseid programmeerimiskeeli kasutada?
- Milliseid tehnoloogiaid kasutada?

Töö vastab nende küsimustele sisu 3. osas.

Töö autor peab mikroteenus-arhitektuuril põhineva rakenduse loomist huvitavaks väljakutseks peamiselt selle tõttu, et mikroteenus-arhitektuur on tänapäevase tarkvaraarenduse kontekstis äärmiselt aktuaalne, kuid veel nii-öelda lõplikult lahendamata valdkond. Rakenduse enda sisuks osutus kiirsuhtlus sellel põhjusel, et see sisaldab minimaalsel hulgal äriloogikat ning annab võimaluse keskenduda pigem

arhitektuuri disainile, kuid samas loob huvitavad tehnilised väljakutsed, näiteks reaalajas vahendatavad sõnumid.

1.1 Ülesandepüstitus

Töö eesmärgiks on luua mikroteenus-arhitektuuril baseeruv kiirsuhtlust võimaldav rakendus, mis suudab vastavalt vajadusele vahendada suurt hulka sõnumeid. See tähendab, et loodud rakendust peab olema võimalik skaleerida, et

1) suurendada sõnumite läbilaskvust ühel ajahetkel

2) tagada suurenenud hulga sõnumite salvestumine andmebaasi.

Sõnumite vahetamise all peetakse silmas kahe kasutaja vaheliste sõnumite saatmist, mitme kasutaja vestlused antud rakenduse funktsionaalsuses ei sisaldu. Rakendus koosneb järgmistest komponentidest: brauseripoolne klient-rakendus ja serveris paiknev teenus, mis koosneb mikroteenustest. Antud töös asetub rõhk viimasele.

1.2 Metoodika

Töö autor kavatseb töö käigus luua täisfunktsionaalse tarkvara. Eesmärgi saavutamiseks on autor läbi lugenud Sam Newmani raamatu „*Building Microservices: Designing Fine-grained Systems*“ ja Shahir Daya ja teiste „*Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*“. Nimetatud teostes välja pakutud mustrid on aluseks töö käigus valmiva tarkvara arhitektuuri disainile. Lisaks nõuab eesmärkide täitmine paratamatult teatud hulgal eksperimenteerimist, sest kõiki tehnoloogilisi piiranguid ei ole võimalik ette näha.

1.3 Ülevaade tööst

Töö esimeses osas annab autor ülevaate mikroteenuste teoreetilisest poolest. Töö teine osa keskendub arhitektuuri disainile, kolmandas osas kirjaldatakse realiseeritud rakenduse implementatsiooni teenuste kaupa.

2 Ülevaade mikroteenustest

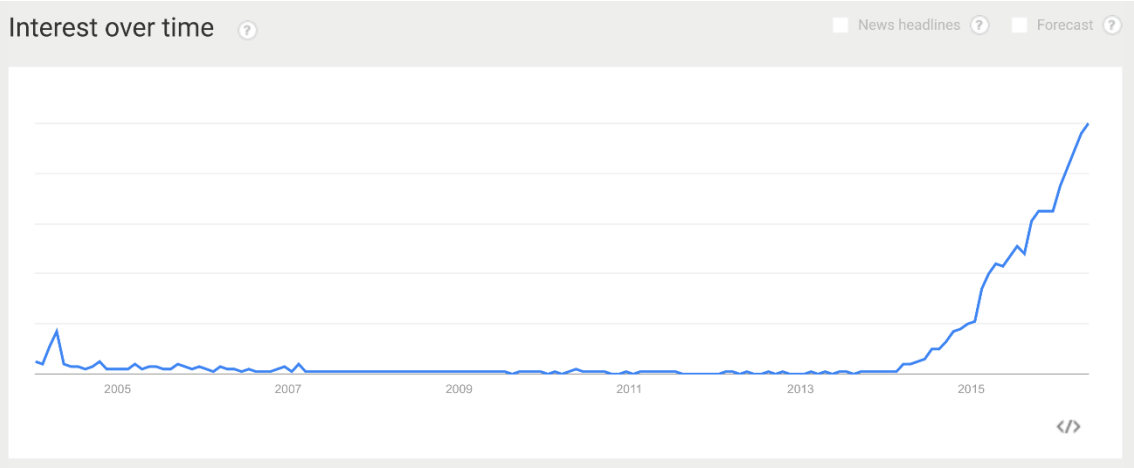
Mikroteenused on hetkel äärmiselt populaarne teema – seda loetakse lahenduseks tänases tarkvaramaailmas, kus probleemiks on skaleeruvus ning suured mahud. Mikroteenuseid võib nimetada lausa revolutsiooniliseks, näiteks korporatsioonimõõtu ettevõtted nagu Netflix on seda arhitektuuri edukalt rakendanud ning avaldanud suurel hulgal selleteemalist kirjandust. Kõik see annab mikroteenustele hoogu ainult juurde ning autori arvates ollakse selle töö kirjutamise hetkel mikroteenuste laineharja tipul.

Hoolimata populaarsuset ei ole mikroteenused lahenduseks kõikidele tarkvaraprobleemidele, vaid enne nende rakendamist tasuks hoolikalt kaaluda põhjuseid, miks mikroteenuste poole üleüldse vaatama hakati. Valdav osa tarkvarasüsteemidest ei hakka tõenäoliselt mitte kunagi teenindama andmemahte, mis ilmtingimata vajaksid piiramatut skaleerimist ning sellisel juhul on mikroteenused autori arvates pigem vale valik.

2.1 Ajalugu

Mikroteenus-arhitektuuri esimesed alged ulatuvad aastasse 2005, kui Dr. Peter Rodgers mainib oma esitluses terminit „mikro-veebiteenused“ (*micro-web-services*) ning kirjeldab tarkvarakomponente kui mikro-veebiteenuseid, millel on igaühel isiklik, sisemiselt ligipääsetav URI; mikro-veebiteenuste omavaheline suhtlus käib *unixi*-laadsete „torude“ kaudu (*unix pipe*) ja mis põhiline, kõik on teenusekeskne [2]. Sellisel arhitektuuril on üsna tugevad ühised jooned tänapäevase mikroteenus-arhitektuuriga.

Termin „mikroteenused“ kerkis esile alles 2010-ndate alguses, kui sellega eksperimenteerisid samaaegselt mitmed tarkvaraarendajad [1]. Sel hetkel hakkas nimetatud arhitektuur võtma tänapäevasemat kuju. Tugevam avalik huvi mikroteenuste vastu hakkas tekkima 2014. aasta alguses, mida näitab ka Google otsingu statistika, mis on välja toodud joonises 1.



Joonis 1 Mikroteenuseid puudutav otsingustatistika Google otsingumootoris 2016. aasta mai seisuga

2.2 Monoliit

Mikroteenuste kui mõiste paremaks selgitamiseks on hea kõrvutada seda traditsioonilise arhitektuuri ehk „monoliidiga“. Martin Fowler kirjeldab monoliiti kui ühte terviklikku, üksikult käivitatavat programmi, mis sisaldab endas kogu äriloogikat, kasutajaliidese presenteerimist, andmebaasi kirjete loomist, lugemist, kustutamist ja kõike muud vajalikku. Selline monoliitne arhitektuur on mõeldud töötama ühe füüsilise protsessina [1].

Monoliidi algstaadiumis on selline arhitektuur kõige loomulikum valik, kuid mida aeg edasi, seda keerukamaks see muutub. Arendaja perspektiivist muutub monoliitne arhitektuur ühel hetkel oma koodibaasi suuruse tõttu raskesti hallatavaks, uutel arendajatel kulub tohutu aeg koodi „õppimiseks“ ja lisaks suureneb monoliidi käivitumisaeg [3, p. 7].

Juurutamise seisukohast on monoliit väga õrn, sest hoolimata muudatuse suurusest tuleb (taas)käivitada kogu rakendus, mis võib olla aeganõudev protsess.

2.3 Definitsioon

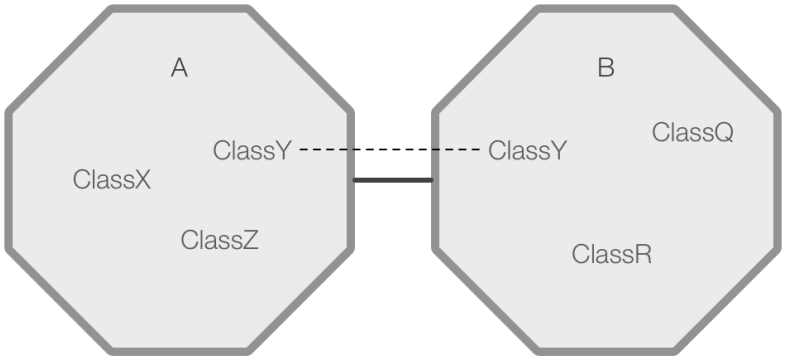
Mikroteenuste näol on tegu tarkvaraarhitektuuriga, kus tarkvara koosneb teenustest, mis keskenduvad ühe ülesande täitmisele. Enamasti on see ülesanne seotud mingi konkreetse ärilise probleemiga, milleks võib olla näiteks kasutajate haldamine või arvete saatmine. Teenused võivad olla kirjutatud erinevates programmeerimiskeeltes, sest

teenustevaheline suhtlus käib keelest sõltumatute programmiliideste (API) kaudu [3, p. 4]. Martin Fowler rõhutab, et mikroteenuste tsentraliseeritud haldus peab olema minimaalne [1].

Mikroteenused on iseseisvad, mis tähendab, et nad töötavad eraldi protsessidena ja võimalusel eraldi füüsilistes serverites. Teenustevahelise suhtluse transpordiks on üldiselt võrguprotokollid nagu HTTP.

Tihtipeale räägitakse mikroteenuste puhul suurusest. Öeldakse, et need peaksid olema „võimalikult väikesed“. Kui väikesed – selle kohta ühene arusaam puudub, mis on ka loogiline, sest mikroteenuse suurus sõltub tugevalt talle antud ülesandest. Ben Morris pakub oma artiklis välja, et sõna „mikro“ ei tähenda „väikest“ suuruse kontekstis, vaid rõhub agiilsusele ja minimaalsele infrastruktuurile [4]. Samas öeldakse, et mikroteenus peaks olema nii suur, et seda arendava meeskonna toidaks ära kahe pitsaga [3, p. 5]. Kolmas definitsioon ütleb, et mikroteenus peaks olema programmeeritav ja juurutatav kahe nädalaga [5].

Suurusest olulisem faktor on mikroteenuste nõrk omavaheline sõltuvus (*loose coupling*). Shahir Daya ja teised on oma raamatus öelnud, et mikroteenuse juurutamisel ei tohiks puutuda teisi, olemasolevaid teenuseid. Lisaks räägitakse mikroteenuste puhul „piiritletud kontekstist“ (*bounded context*) – nimelt ei peaks üks mikroteenus teadma mitte midagi sellest, kuidas teine mikroteenus implementeeritud on ehk kuidas ta andmeid käitleb, millist andmebaasi kasutab [3, p. 5]. Üldistatult seletatuna võiks see tähendada seda, et iga konteksti (antud juhul teenuse) jaoks on mingi klass esitatud erineval kujul.



Joonis 2 Piiritletud kontekst

2.4 Eelised

Mikroteenuste plussid ja miinused võrreldes teiste arhitektuuridega on teatud määral subjektiivsed, kuid autori arvates on mikroteenus-arhitektuuri kõige suuremaks eeliseks fakt, et iga teenus on sisuliselt eraldi programm. See loob mitu võimalust:

- Iga teenust arendab ja juurutab üks meeskond, kes tunneb selle teenuse eripärasid ning on vigade parandamise osas paindlikum
- Tarkvara saab täiendada teenuste kaupa näiteks uuenenud teenuseid võib käitada paralleelselt vanadega ning samm-sammult suunata sissetulevaid päringuid ümber vanast teenusest uude (*rolling update*)
- Skaleerida võib täpselt neid rakenduse osi (teenuseid), mis vajavad rohkem ressurssi, erinevalt monoliidist, kus skaleerida tuleb tervet rakendust
- Iga teenus võib olla kirjutatud ise keeles, monoliit on tüüpiliselt kirjutatud ühes programmeerimiskeeles

2.5 Puudused

Mikroteenuste näol ei ole tegu hõbekuuliga – neil on ka arvestatavad puudused, mida peaks kaaluma tõsiselt enne mikroteenus-arhitektuuri kasuks otsustamist. Ühe suure puudusena tuuakse välja kaugprotseduuride väljakutseid (*remote procedure*) ja nendest tekkivaid ajalisi viiteid (*latency*) [3, p. 12] [6]. Monoliitrakenduse puhul toimub kogu infovahetus protsessisiseselt, kus ajaline viide puudub.

Lisaks viitele tuleb arvestada ka võimalusega, et kaugprotseduuri väljakutsumine ebaõnnestub sootuks ning see on üks lisanduv faktor, millega tuleb arvestada iga kord, kui mikroteenused soovivad omavahel infot vahetada. Väljakutsumise ebaõnnestumisel muutub rohkem kui kahe teenuse vahelise suhtluse silumine (*debugging*) keerukamaks ja sõltuvalt rakenduse ülesehitusest ei pruugi olla selge, kus viga ilmnes. Selline olukord võib tekkida näiteks REST-stiili kasutava HTTP-liidese puhul, mis pärib andmeid mõne kolmanda teenuse käest. Vea tekkimiseks on kaks varianti: esines teenusesisene viga või kolmanda teenusega ei õnnestunud suhelda ning ainuüksi standardsete HTTP veakoodidega selliseid olukord eristada ei saa.

Mikroteenuste puhul on äärmiselt oluline tugeva infrastruktuuri ja pädeva tiimi olemasolu, kuna „liikuvaid osi“ on tunduvalt rohkem kui monoliidis ja kõiki neid tuleb konfigureerida, monitoorida, hallata. Lisaks haldamisele suureneb ka vajadus ressursside järele, kuna igal teenusel on teatav administratiivne ressursikulu ehk *overhead*, mis võib kehvasti disainitud arhitektuuri puhul osutuda liialt suureks. Sellise kulu ja mikroteenustest saadava kasu suhe võib osutuda ebamõistlikuks, mis on omakorda märk halvast arhitektuuri disainist või valest arhitektuurivalikust.

2.6 Teenus-orienteeritud arhitektuur

Oma olemuselt võiks mikroteenus-arhitektuuri nimetada teenus-orienteeritud arhitektuuriks ehk SOA-ks [7]. Piir nende kahe vahel on üsna hägune ja arvamusi selle kohta, kas mikroteenuste näol on tegu SOA-ga, on mitmeid, näiteks Netflix on oma mikroteenus-arhitektuuri kohta öelnud, et tegemist on hoolikalt valminud teenus-orienteeritud arhitektuuriga [8]. Shahir Daya ja teised on öelnud, et SOA rõhk on taaskasutatavusel (*reusability*) ja modulaarsusel (*modularity*), aga mikroteenuste eesmärk on teha monoliit väiksemateks, kergemini käsitletavateks tükkideks [3, p. 14]. Arvatakse ka, et mikroteenused on õnnestunum variant SOA-st [1].

Oma olemuselt on mikroteenuste ja SOA vahel siiski teatavad erinevused. Mark Richards nimetab oma raamatus neist järgmisi:

- SOA puhul jagatakse teenuseid/komponente maksimaalselt, mikroteenuste puhul üritatakse seda teha minimaalselt [9, p. 22]
- SOA puhul toimub pigem teenuste orkestreerimine, mikroteenuste puhul koreograafia ehk keskse juhtimise puudumine [9, p. 24]
- SOA toetub vahevarale (*middleware*), mikroteenused programmiliidesele (API) [9, p. 30]
- SOA puhul on normaalne kasutada mitmeid erinevaid suhtlusprotokolle, mikroteenus-arhitektuuril on tavaliselt üks keskne protokoll, näiteks REST [9, p. 33].

3 Arhitektuuri disain

Nagu iga teise tarkvaraarhitektuuri, on ka mikroteenus-arhitektuuri kavandamine keeruline protsess. Antud töös valmiva rakenduse funktsionaalsus on minimaalne, et keskenduda maksimaalselt arhitektuuri disainimisele.

Antud protsessi puhul ei ole autor aluseks võtnud ühtegi kindlat juhist või mustrit ning oma vähese funktsionaalsuse tõttu on võimalik kõiki rakenduses esinevaid komponente muuta vajadusel implementatsiooni käigus.

3.1 Mikroteenusteks jaotamine

Esimene probleem arhitektuuri disainimisel on rakenduse jaotamine mikroteenusteks. Chris Richardson loetleb kaks põhilist lähenemisviisi:

1) jaotamine tegevuste või kasutuslugude

2) ressursside põhjal [10].

Antud töös on autor aluseks võtnud kombinatsiooni mõlemast, selgitades kõigepealt välja probleemid, mida on vaja lahendada.

Loodavas kiirsuhtlusrakenduses on viis probleemi: kasutajate säilitamine, kasutajate identifitseerimine, sõnumite säilitamine, sõnumite vahendamine reaalajas, kasutajaliidese presenteerimine. Antud rakenduses ei ole võimalik neid probleeme üks-ühele üle kanda mikroteenusteks, kuna paratamatult tekib vajadus teatavate abiteenuste järele. Järgnevalt on loetletud käesolevas rakenduses sisalduvad mikroteenused.

- **Kasutajate säilitamise teenus.** Teenuse eesmärgiks on võtta vastu registreerimispäringuid ning säilitada vastav kasutajakirje edaspidiseks kasutamiseks.
- **Kasutajate identifitseerimise teenus.** Teenuse eesmärgiks on tuvastada kasutaja ning korrektsete tunnuste korral anda tagasi vastav luba (*token*).
- **Kasutajaliidese presenteerimise teenus.** Teenuse eesmärgiks on serveerida kasutajaliidest.

- **Sõnumite säilitamise teenus.** Teenuse eesmärgiks on võtta vastu sõnumipäringuid, säilitada vastavad sõnumikirjed ning tagastada need mingil loogilisel ja organiseeritud kujul pärijale.

- **Sõnumite reaalajas vahendamise teenus.** Teenuse eesmärgiks on võimaldada kasutajaliidesel reaalajas saata ja vastu võtta sõnumeid.

- **Sõnumite järjekorrast salvestamise teenus.** Teenuse eesmärgiks on vastu võtta järjekorras ootel olevad sõnumid ning edastada need sõnumite säilitamise teenusele.

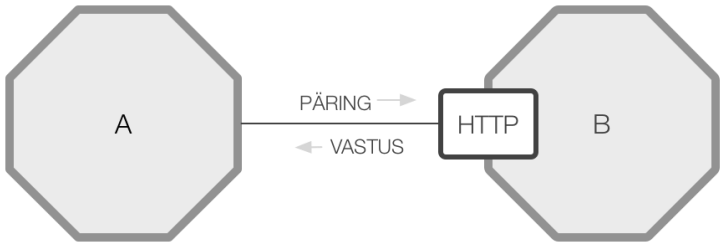
- **Avalik programmiliidese lüüs.** *API gateway*. Teenuse eesmärgiks on vahendada ja autentida väljaspoolt sisevõrku saabunud päringuid.

3.2 Teenustevaheline suhtlemine

Rakenduses on kasutusel kaks suhtlemisviisi – HTTP ja järjekorraprotokoll AMQP [11].

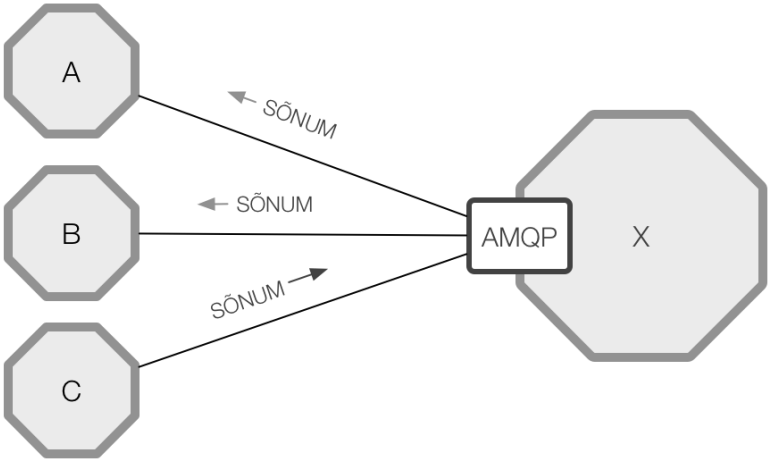
HTTP eesmärgiks on tagada küsimus-vastus (*request-response*) stiilis suhtlus. Antud rakenduses kasutavad HTTP protokolli **autentimisteenus**, **sõnumite ja kasutajate säilitamise teenused** ja **avaliku programmiliidese lüüs**. HTTP päringutes ja vastuses kasutatav andmeformaat on JSON [12], mis on veebiteenustes kujunenud *de facto* standardiks.

HTTP asemel võinuks kasutada ka mõnda muud päring-vastus protokolli, näiteks *Protocol Buffers* [13] või *Apache Thrift* [14]. Kahe nimetatud protokolli eeliseks on jõudlus, kuna esiteks vahetakse kodeerimata binaarset infot ja teiseks, nende protokollide puhul on iga päringuga vahetatav mittesisuline info (*overhead*) hulk tunduvalt väiksem võrreldes HTTP-ga, kus iga päringu ja vastusega saadetakse teatav hulk päiseid (*headers*). Küll aga on nende protokollide oluliseks puuduseks staatilisus – selleks, et klient ja server saaksid infot vahetada, on vaja eelnevalt vaja mõlemasse paigaldada definitsioonifailid, mis kirjeldavad, milline on vahetatavates pakettides saadetav info. See aga tähendab, et serveripoolse definitsioonifaili muutmisel on vaja teha sama ka kliendipoolses rakenduses.



Joonis 3 Suhtlus HTTP abil

Järjekorraprotokolli eesmärgiks on saata kolmandatele osapooltele sõnumeid – huvitatud osapooled (siin kontekstis teenused) registreerivad end teatud järjekorras tarbijana (AMQP kõnepruugis *consumer*). Tarbijat teavitatakse saabunud sõnumitest. Oluline erinevus HTTP protokolliga võrreldes on fakt, et sõnumitest teavitav teenus (AMQP kõnepruugis *producer*) ei ole teadlik sellest, kas ja keegi neid sõnumeid üleüldse käsitleb ehk tarbib ning tarbijate puudumisel on järjekorras olevad sõnumid lihtsalt ootel. Antud rakenduses kasutavad nimetatud protokolli **sõnumite reaalajas vahendamise teenus** ja **sõnumite järjekorrast salvestamise teenus.**



Joonis 4 Suhtlus *message queue* abil

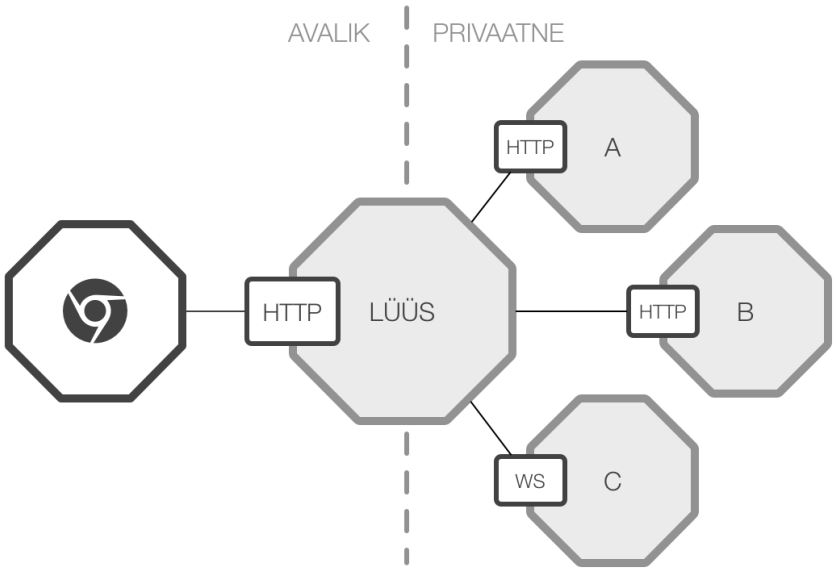
HTTP ja AMQP üheks oluliseks erinevuseks on sünkroniseeritus – HTTP nõuab mõlema osapoole samaaegset kohalolu, AMQP puhul toimub sõnumi edastamine ja lugemine asünkroonselt ehk sõnumi tarbija võib end järjekorda registreerida endale sobival ajal. Keelelistel põhjustel nimetatakse antud dokumendis nii järjekorraprotokolli sõnumit kui ka kiirsuhtlussõnumit sama sõnaga.

3.3 Kasutajaliidesega suhtlemine

Kasutajaliides on realiseeritud brauserirakendusena. See tähendab, et kõik päringud tehakse lõppkasutaja brauserist ehk välisvõrgust. Neid päringuid vahendab **avalik programmiliidese lüüs.** Lüüs vahendab järgmisi teenuseid: **autentimisteenus**, **sõnumite ja kasutajate säilitamise teenused** ja **sõnumite reaalajas vahendamise teenus.**

Lüüsi võimalikeks ülesanneteks on hoida kontrolli all sise- ja välisvõrgu vahel liikuvaid päringuid ehk autentida, autoriseerida, jagada koormust. Antud rakenduse avalik lüüs otseselt autoriseerimisega ei tegele ning sisuliselt koondab see teatud sisevõrgus olevad teenused üheks avalikuks teenuseks, et kasutajaliides ei peaks suhtlema mitme erineva serveriga [15] [16].

Üks võimalikke avaliku lüüsi rakendusi on veel *SSL termination* ehk turvalise ja ebaturvalise ühenduse vahendamine – brauser ja lüüs suhtlevad turvalist ühendust kasutades, lüüs ja sisevõrgu teenused aga kasutavad ebaturvalist ühendust. Käesolevas töö s nimetatud funktsionaalsust realiseeritud pole.



Joonis 5 Avalik programmiliides

3.4 Andmebaasid ja andmete säilitamine

Iga mikroteenus säilitab andmeid iseseisvalt. Teenused ei tohi lubada ligipääsu teineteise andmekogumitele, näiteks tabelitele [17]. Sam Newman ütleb, et andmebaasi otsese ligipääsu andmine erinevatele teenustele on küll kõige lihtsam viis andmeid jagada, kuid

sellisel juhul ei kontrolli miski, kuidas neid andmeid kasutatakse [18, p. 16]. Selle asemel peaksid teenused suhtlema ainult läbi avalike liideste ning seeläbi on tagatud teatav järjepidevus. Näiteks kui teenus A otsustab ühel hetkel vahetada mitterelatsioonilise andmebaasi relatsioonilise vastu, siis teenus B, kes sõltub teenusest A, töötab muutumatul kujul edasi ilma probleemideta.

Käesolevas rakenduses on kaks teenust, mis andmebaasi kasutavad – need on kasutajate ja sõnumite säilitamise teenused. Teenuste jaoks pole otseselt oluline, kas need jagavad ühte füüsilist andmebaasi või mitte. Arenduskeskkonnas on lihtsam kasutada ühte andmebaasi. Sellest hoolimata loevad ja kirjutavad teenused ainult enda andmekogumeid ning andmete muutmiseks on mõlemal HTTP liides.

3.5 Skaleeritavus

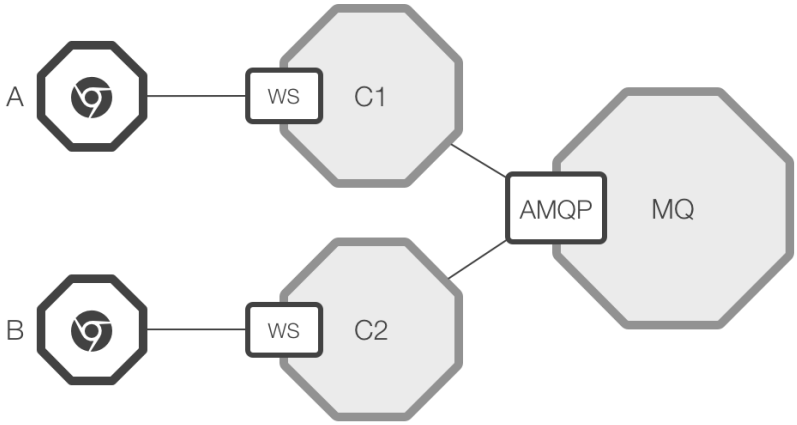
Rakenduse arhitektuuri disaini üks olulisimaid punkte on skaleeritavus. See tähendab, et mikroteenuse instantside arvu suurendamisel kasvab ka rakenduse võimekus päringuid vastu võtta.

3.5.1 Instantsidevaheline suhtlus

Lihtsamatel juhtudel ühendub teenus andmebaasi ning piisab, kui suurendada teenuse instantside arvu, kuna instantsid ei pea otseselt teineteisega suhtlema, sest selle probleemi on lahendanud andmebaasisüsteem. Teenuse instants A loob andmebaasi kirje X ja teenuse instants B võib endale sobival hetkel seda kirjet lugeda. See, kuidas on lahendatud andmebaasisüsteemi skaleerimine, on teenuste jaoks varjatud. Nimetatud mudelit kasutavad **sõnumite ja kasutajate säilitamise teenused.**

Teatud teenuste ülesehitus nõuab aga instantside omavahelist suhtlemist. Antud juhul on üheks selliseks teenuseks **sõnumite reaalajas vahendamise teenus.** Probleem tekib sellest, et lõppkasutaja ühendused, läbi mille sõnumite teavitusi edastatakse, suunatakse erinevatesse instantsidesse. Kui kasutaja A, kes on ühendatud instantsi C1, saadab sõnumi kasutajale B, kes on ühendatud instantsi C2, siis kuidas see sõnum kohale toimetada? Antud juhul on kirjeldatud probleem lahendatud AMQP serveriga. Kõnealune teenus loob ühendunud lõppkasutaja jaoks kasutajanimelise järjekorra ning registreerib ennast selle järjekorra tarbijaks. Nüüd kui kasutaja A saadab sõnumi kasutajale B, siis toimetatakse

sõnum kohale läbi AMQP serveri, teavitades instantsi C2. Sarnaselt esimesele mudelile on teenuse jaoks AMQP serveri skaleerimine varjatud.



Joonis 6 Instantsidevaheline suhtlus *message queue* abil

3.5.2 Tööjaotus

Rakenduses on kasutusel veel üks variant ehk teenused, mille eesmärgiks on tarbida AMQP järjekordadesse saabuvaid sõnumeid. Siin tuleb mängu AMQP järjekorra üks oluline aspekt – nimelt võib end järjekorra tarbijaks registreerida rohkem kui üks teenuse instants. Sellisel juhul jaotab AMQP server järjekorda saabunud sõnumid mingi eelnevalt paika pandud reegli alusel erinevatele tarbijatele. Näiteks end järjekorda tarbijaks registreerinud kaks tarbijat, kelleks on A ja B. Järjekorda saabub kolm sõnumit. Kõige lihtsamal juhul toimib jaotus järgmiselt[19].

1. Saabub sõnum X – teavitatakse tarbijat A
2. Saabub sõnum Y – teavitatakse tarbijat B
3. Saabub sõnum Z – teavitatakse tarbijat A

Selline jaotamine loob väga hea võimaluse skaleerimiseks. Oletame, et AMQP serverile pole saadetavate sõnumite hulk probleemiks, küll aga ei suuda järjekorda registreerunud tarbija neid piisava kiirusega vastu võtta. Probleemi lahendab tarbijate (teenuse instantside) arvu suurendamine, sest instantsid võivad sõnumeid töödelda samaaegselt. Nimetatud mudelit kasutab **sõnumite järjekorrast salvestamise teenus.**

3.6 Tehnoloogia

Mikroteenus-arhitektuur ei sea süsteemiüleseid keelelisi ega tarkvaralisi piiranguid. Iga teenus peaks kasutama just neid tehnoloogiaid, mis selleks puhuks kõige sobivamad on ning see hõlmab programmeerimiskeelt, raamistikke, teeke, andmebaasisüsteeme. Mõnele teenusele võib sobida relatsiooniline, teisele mitterelatsiooniline andmebaas. Mõne lihtsama teenus jaoks võib kasutada Node.js keskkonda, samal ajal kui teine, kriitilisem reaalaja teenus on kirjutatud C++ programmeerimiskeeles [1].

3.6.1 Protokollid

Protokollidest on kasutusel HTTP (kahe teenuse vaheliseks päring-vastus suhtluseks), AMQP (teavitaja-tarbija stiilis sõnumite edastamiseks), WebSocket (brauseri ja serveri vaheliseks reaalaja päring-vastus suhtluseks).

3.6.2 Programmeerimiskeeled

Antud rakenduse kõik teenused on kirjutatud JavaScriptis ning on mõeldud töötama Node.js keskkonnas. JavaScript sobib antud konteksti hästi, kuna see võimaldab kiiret prototüüpimist tänu oma lihtsusele ja populaarsusele (saadaval on väga suurel hulgal teeke [20]). JavaScripti versiooniks on ECMAScript 2015 ehk ES6 [21], mida kompileerib vanemasse, hetkel rohkem toetatud ES5 versiooni Babel [22].

3.6.3 Andmebaasisüsteemid

Andmebaasisüsteemidest on kasutusel MongoDB [23], sest kiirsuhtlusrakenduse puhul on oluline maht ning kiirus. Selleks, et salvestada kasutajalt kasutajale saadetud sõnumeid, pole vaja traditsiooniliselt kasutatavat, relatsioonilist andmebaasi, mida on oma olemuselt keerulisem horisontaalselt skaleerida. Muus osas osutus MongoDB valituks oma populaarsuse pärast.

3.6.4 Raamistikud/teegid

Märkimisväärsetest raamistikest on kasutusel Express kui lihtne HTTP server [24]. AMQP serveriga ühendumiseks on amqplib [25]. MongoDB serveriga suhtlemiseks on kasutusel ametlik Node.js draiver [26]. Brauserirakendus toetub põhiliselt kahele teegile – React ja Redux [27] [28].

3.6.5 Konteinerid

Rakenduse arhitektuuris on silmas peetud konteinerite kasutamise võimalikkust. Konteineritarkvaraks on Docker [29]. Iga teenus sisaldab Dockeri kujutise e. *image* loomiseks mõeldud konfiguratsioonifaili *Dockerfile* [30]. Rakenduse teenuseid võib käitada ka väljaspool konteinereid – nende mittekasutamine ei sea mingeid tehnilisi piiranguid.

3.6.6 Muu tarkvara

Sõnumite vahendamiseks on kasutusel RabbitMQ [31], kuigi selle asemel võiks kasutada ka mõnda teist AMQP protokolli teostavat teenus.

Avaliku programmiliidese lüüsi jaoks on kasutusel nginx HTTP server [32]. HTTP serveri üheks kriteeriumiks oli tagurpidi proksi (*reverse proxy*) olemasolu. Kuigi oma olemuselt sama funktsionaalsust sisaldavad ka Apache (koos mod_proxy mooduliga) [33] ja HAProxy [34], siis määravaks kriteeriumiks nginx puhul osutus populaarsus ning konfiguratsiooni lihtsus.

Sessioonide/lubade hoiustamiseks on kasutusel Redis [35], mis on oma olemuselt mälusisene andmeladu. Redise eesmärgiks ei ole andmete pikaajaline säilitamine, vaid nende kiire ligipääs ja käitlemine, lisaks on sellel tagasihoidlik mälukasutus [36].

4 Implementatsioon

Töö käigus lõi autor täisfunktsionaalse tarkvara, mis on mõeldud võrdlemisi lihtsa vaevaga paigaldatama Docker-konteinereid toetavasse süsteemi, milleks võib-olla tavaline sülearvuti või hajus pilveteenus. Kogu tarkvara on kirjutatud JavaScriptis ning töötab Node.js keskkonnas.

Loodud rakendus täidab mõlemad püstitatud eesmärgid ehk see võimaldab saata kasutajatel teineteisele sõnumeid ning baseerub mikroteenus-arhitektuuril. Autori arvates võib viimast põhjendada järgmiselt:

- iga teenus on füüsiliselt eraldiseisev programm, mille sisemine implementatsioon on välisest liidesest täielikult eraldatud,
- teenused on võrdlemisi väikseid – kui vaadata teenuset koodiridade arvu, jääb see vahemikku ~80-280 rida (kui jätta välja kasutajaliides),
- igal teenusel on üks selgesti piiritletav ülesanne,
- teenused on omavahel seotud ainult avalike liidestega.

4.1 Sõnumite säilitamise teenus

Oma olemuselt on nimetatud teenus HTTP server, mis võtab vastu päringuid ning loob või tagastab vastavaid kirjeid MongoDB andmebaasist.

4.1.1 NoSQL andmemudel

Andmemudel on inspireeritud MongoDB blogipostitusest sõnumite saatmise andmemudeli teemal [37]. Antud teenus võtab eeskujuks artiklis väljatoodud meetoditest viimase (*fan out on write with buckets*). Selle meetodi printsiibiks on grupeerida sõnumid mingi fikseeritud koguse järgi ning teatud koguse juures luua uus grupp ehk ämber. Selline lähenemine sobib rakendusse hästi, sest sõnumeid on vaja tagastada vestluspõhiselt ja rohkemal hulgal kui üks, mis tähendab, et andmebaas ei otsi üksikuid sõnumeid, vaid ämbreid, milles on mingi suurem hulk sõnumeid korraga. Kui lõppkasutaja avab brauserirakenduses vestluse, siis piisab andmebaasil ühest lugemisoperatsioonist. Andmemudelis sisalduvad järgmised kogumid:

Tabel 1 Sõnumite säilitamise teenuse andmemudel

Kogumi nimetus	Väärtuse nimetus	Väärtuse kirjeldus
User	*_id*	Unikaalne identifikaator
	name	Kasutajanimi
	conversations	Loetelu viidetest vestlustele, kus kasutaja osaleb
Conversation	*_id*	Unikaalne identifikaator
	participants	Loetelu vestluses osalejate kasutajanimedest
	messageCount	Vestluses vahetatud sõnumite koguarv
	lastMessage	Viimati vahetatud sõnum
Bucket	*_id*	Unikaalne identifikaator
	conversationId	Viide vestlusele, mille juurde antud ämber kuulub
	sequence	Ämbri järjekorranumbrit iseloomustav number, mis arvutatakse vestluses vahetatud sõnumi koguarvust
	messages	Loetelu vestluses vahetatud sõnumitest

Ämbrite kogumi näol on tegu MongoDB kildkogumiga (*shard collection*), mille kildvõtmeks (*shard key*) on kahest väljast koosnev liitvõti (vestluse identifikaator ja ämbri järjekorranumber). Nimetatud võti on vajalik, sest teenus loeb ja uuendab ämbrite kogumi dokumente just nende väljade põhjal.

4.1.2 HTTP liides

Tabel 2 Sõnumite säilitamise teenuse HTTP liides

HTTP verb	URL	Kirjeldus
POST	/conversations	Loob andmebaasi kirje kahe kasutaja vahelisest vestlusest. Sisendiks on üks parameeter – *participants*, milleks on massiiv vestluses osalejate kasutajate nimedest.
GET	/conversations	Tagastab andmebaasist kõik vestlused, milles etteantud kasutaja osaleb. Sisendiks üks parameeter: *name* – kasutaja, kelle vestlusi soovitakse näha.
POST	/messages	Loob andmebaasi kirje ühelt kasutajalt teisele saadetud sõnumist. Parameetrid: • *conversationId* – viide vestlusele • *from* – saatja kasutajanimi • *to* – saaja kasutajanimi • *text* – sõnumi sisu
GET	/buckets	Tagastab grupeeritud kujul etteantud vestluses vahetatud sõnumid. Parameetrid: • *conversationId* – viide vestlusele • *sequence* – ämbri järjekorranumber, vaikimisi tagastatakse viimane ämber • *limit* (vaikimisi 1) – maksimaalne hulk ämbreid, mida tagastada

4.1.3 Koodinäited

```
User = db.collection("users");
Conversation = db.collection("conversations");
Bucket = db.collection("buckets");

User.createIndex({
    name: 1
}, {
    unique: true
});

Bucket.createIndex({
    conversationId: 1,
    sequence: -1
});
```

Joonis 7 Sõnumite säilitamise teenuse MongoDB kollektsioonid

```
const message = {
    _id: new ObjectID(), timestamp: new Date(), from, to, text
};

Conversation.findOneAndUpdate({
    _id: ObjectID(conversationId)
}, {
    $inc: {
        messageCount: 1
    },
    $set: {
        lastMessage: message
    }
}, {
    new: true
}).then(result => {
    const { value: document } = result;
    const { messageCount = 0 } = document;

    Bucket.update({
        conversationId,
        sequence: Math.floor(messageCount / BUCKET_SIZE)
    }, {
        $push: {
            messages: message
        }
    }, {
        new: true,
        upsert: true
    });
});
```

Joonis 8 Sõnumi säilitamine MongoDB andmebaasi

```
Conversation.insertOne(conversation).then(result => {
    const { insertedId } = result;

    User.find({
        name: {
            $in: participants
        }
    }).toArray().then(results => {
        const existing = results.map(result => result.name);
        const nonExisting = underscore.difference(participants, existing);

        const promises = [];

        User.insertMany(nonExisting.map(name => {
            return {
                name,
                conversations: [insertedId]
            };
        }));

        User.updateMany({
            name: {
                $in: existing
            }
        }, {
            $push: {
                conversations: insertedId
            }
        });
    });
});
```

Joonis 9 Vestluse loomine sõnumite säilitamise teenuses

4.2 Sõnumite reaalajas vahendamise teenus

Nimetatud teenus võtab brauserilt läbi WebSocket serveri vastu sõnumi, töötleb ning suunab edasi RabbitMQ serverisse, kus see õigesse järjekorda pannakse. Seejärel kordub vastupidine sündmuste jada – teenus saab teavituse saabunud sõnumist ning edastab selle WebSocket serveri külge ühendunud brauserile, kus see lõppkasutajale presenteeritakse.

4.2.1 WebSocket liides

Teenuse käivitumisel luuakse WebSocket server [38]. Kasutaja brauseri ühendumisel kontrollitakse URL-i parameetreid, mille hulgas peab sisalduma kasutajanimi ja luba (*token*). Seejärel kontrollib käesolev teenus identifitseerimisteenuselt, kas antud luba eksisteerib ning võrdleb loaga seotud ja liidesega ühendumisel esitatud kasutajanime.

Ebaõige kombinatsiooni korral loodud ühendus katkestatakse – see saavutatakse *verifyClient* meetodi abil [39].

Korrektsete isikutunnuste korral veendub teenus RabbitMQ serveris kasutajanimelise järjekorra olemasolus [40] ja asub vastu võtma WebSocket serverist ja kasutajanimelisest järjekorrast saabuvaid teavitusi .

Sisemiselt vahetatakse kahte tüüpi teavitusi – esimene teavitab sõnumi saatmisest, teine vestluse loomisest. See on tingitud sõnumite säilitamise teenuse andmemudeli iseärasusest, kuna selleks, et salvestada sõnumit, on vaja teada vestluse identifikaatorit. Juhul kui brauserirakendusele saabub teavitus uuest sõnumist (eelnevalt on kasutaja A loonud vestluse, aga B pole vahepeal oma vestluste loetelu värskendanud) ja kasutaja B soovib kasutajale A vastata, tekib probleem, kus pole teada, millise vestluse juurde antud sõnum kuulub. Selleks on olemas teist tüüpi teavitus, mis annab märku loodud vestlustest ja nende identifikaatoritest.

```
{
    "type": "MESSAGE_SENT",
    "message": {
        "to": "nimi",
        "text": "sõnumi sisu"
    }
}

{
    "type": "CONVERSATION_CREATED",
    "conversation": {
        "participants": ["A", "B"],
        "conversationId": "xyz"
    }
}
```

Joonis 10 Sisemised sõnumite tüübid sõnumite reaalajas vahendamise teenuses

4.2.2 Koodinäited

```
socket.on("message", data => {
    const decoded = JSON.parse(data);

    const { type } = decoded;

    let to, forwardedMessage;

    switch(type) {
        case "MESSAGE_SENT":
            const { message: {
                    conversationId, to: messageTo, text } } = decoded;
            to = messageTo;

            forwardedMessage = {
                type: "MESSAGE_SENT",
                message: {
                    conversationId, from: name, text
                }
            };

            break;

        case "CONVERSATION_CREATED":
            const { conversation: { id, participants: [a, b] } } = decoded;

            to = name === a ? b : a;

            forwardedMessage = {
                type: "CONVERSATION_CREATED",
                conversation: {
                    id, participants: [a, b]
                }
            };

            break;
    }

    channel.publish(exchange, to,
      new Buffer(JSON.stringify(forwardedMessage)));
});
```

Joonis 11 WebSocket ühendusest saabuva sõnumi edastamine järjekorda

```
channel.consume(name, message => {
    const decoded = JSON.parse(message.content.toString());

    const { type } = decoded;

    let forwardedMessage;

    switch(type) {
        case "MESSAGE_SENT":
            const { message: {
                conversationId, from, text } } = decoded;

            forwardedMessage = {
                type: "MESSAGE_SENT",
                message: {
                    conversationId, from, text
                }
            };

            break;

        case "CONVERSATION_CREATED":
            const { conversation: { id, participants } } = decoded;

            forwardedMessage = {
                type: "CONVERSATION_CREATED",
                conversation: {
                    id, participants
                }
            };

            break;
    }

    connections[name].send(JSON.stringify(forwardedMessage));

    channel.ack(message);
});
```

Joonis 12 Järjekorrast saabuva sõnumi edastamine WebSocket ühendusele

4.3 Sõnumite järjekorrast salvestamise teenus

Teenuse eesmärgiks on siduda sõnumite järjekord ja sõnumite säilitamise teenus. Antud teenus registreerib end RabbitMQ serveris *messages-persistence* nimelise järjekorra tarbijaks. Sellesse järjekorda saabuvad kõikide kasutajate saadetud sõnumid. Sõnumi saabumisel edastatakse see sõnumite säilitamise teenuse HTTP liidesele.

Vigaseid sõnumeid, mida antud teenus dekodeerida ei suuda või millel pole kõiki vajalikke parameetreid, ei edastata. Juhul kui sõnumi salvestamine ebaõnnestub, pannakse antud sõnum tagasi järjekorda. Protsess kordub, kuni sõnum on salvestatud.

See teenus ei oma välist liidest.

4.3.1 Koodinäited

```
channel.consume(queueName, rabbitMessage => {
    const { content: rawContent, fields: { routingKey: to } } =
rabbitMessage;
    const decoded = JSON.parse(rawContent.toString());
    const { type } = decoded;

    switch(type) {
        case "MESSAGE_SENT":
            const { message: { conversationId, from, text } } = decoded;

            request({
                url: rest + "/messages",
                method: "POST",
                json: true,
                body: {
                    conversationId, from, to, text
                }
            }, (error, response, body) => {
                channel.ack(rabbitMessage);
            });
    }
});
```

Joonis 13 Järjekorrast sõnumi salvestamine

4.4 Kasutajate säilitamise teenus

Antud teenuse loob ning tagastab andmebaasist kasutajakirjeid vastavalt päringutele.

4.4.1 NoSQL andmemudel

Tabel 3 Kasutajate säilitamise teenuse andmemudel

Kogumi nimetus	Väärtuse nimetus	Väärtuse kirjeldus
User	*_id*	Unikaalne identifikaator
	name	Kasutajanimi
	password	Parool

4.4.2 HTTP liides

Tabel 4 Kasutajate säilitamise teenuse HTTP liides

HTTP verb	URL	Kirjeldus
POST	/	Loob andmebaasi kasutajakirje. Parameetrid: *name*, *password*.
GET	/	Tagastab andmebaasist kasutajakirje. Parameetrid: *name*.

4.5 Kasutajate identifitseerimise teenus

Teenuse eesmärgiks on jagada lube (*token*) ning kontrollida nende kehtivust. Sisemiselt hoiustab teenus lubade infot Redis serveris. Andmestruktuur on oma olemuselt võti-väärtus paaride hulk, kus võtmeks on luba (identifitseerimise hetkel suvaliselt genereeritud sõne või number) ja väärtuseks kasutajanimi.

Iga kasutajaga võib olla seotud suvaline hulk lube. Selleks, et hoida kontrolli all lubade arvu, seatakse antud implementatsioonis igale loale EXPIRE käsuga [41] teatud eluiga – TTL [42]. See eluiga nullitakse iga autoriseerimispäringuga – uue sisselogimise puhul

kustub eelnev luba mingi aja pärast. Loa lõplik eluiga on kasulik ka turvalisuse aspektist, kuna kolmanda isiku ligipääs loale võib osutuda kasutuks, kui luba on oma kehtivuse kaotanud.

Teenus viib läbi kahte operatsiooni – loa andmine ja loa kontrollimine. Esimese puhul on sisendiks kasutajanimi ja parool, mille õigsust kontrollitakse kasutajate säilitamise teenusest. Korrektsete isikutunnuste korral genereerib identifitseerimisteenus uue loa ning seob selle kasutajanimega, ebakorrektsete puhul tagastatakse vastav HTTP veakood ja luba ei anta. Teise operatsiooni puhul on sisendiks luba – teenus kontrollib, kas antud võti-väärtus paar eksisteerib ning kui see eksisteerib, tagastatakse positiivne HTTP kood ning loaga seotud kasutajanimi ja kui võti-väärtus paari eksisteeri, vastav HTTP veakood.

Antud teenust kasutavab autentimiseks avalik lüüs – pärast edukat autentimist edastab lüüs sisemisele teenusele autenditud kasutaja nime, mille alusel toimub autoriseerimine.

4.5.1 HTTP liides

Tabel 5 Kasutajate autoriseerimise teenuse HTTP liides

HTTP verb	URL	Kirjeldus
POST	/	Korrektsetse isikutunnuste korral genereerib ning tagastab kasutajaga seotud loa. Parameetrid: *name*, *password*.
GET	/	Kontrollib kasutajaga seotud loa kehtivust. Parameetrid: *token*.

4.5.2 Koodinäited

```
request({
    url: usersUrl,
    qs: {name, password}
}, (error, response, body) => {
    const { password: actualPassword } = JSON.parse(body);

    if(password === actualPassword) {
        const token = Math.random().toString(36).slice(2);

        redisClient.set(token, name, (error, reply) => {
            res.send({token});
        });
    } else {
        res.status(401).end();
    }
});
```

Joonis 14 Kasutajale loa andmine

```
redisClient.get(token, (error, reply) => {
    if(reply) {
        res.set("X-Authenticated-User", reply)

        if(headerToken)
            res.end();
        else
            res.send({name: reply});
    } else {
        res.status(401).end();
    };
});
```

Joonis 15 Loa kehtivuse kontrollimine

4.6 Avalik programmiliidese lüüs

Antud teenus on nginx server, mille konfiguratsioon koondab sisemised teenused kokku üheks väliselt ligipääsetavaks teenuseks.

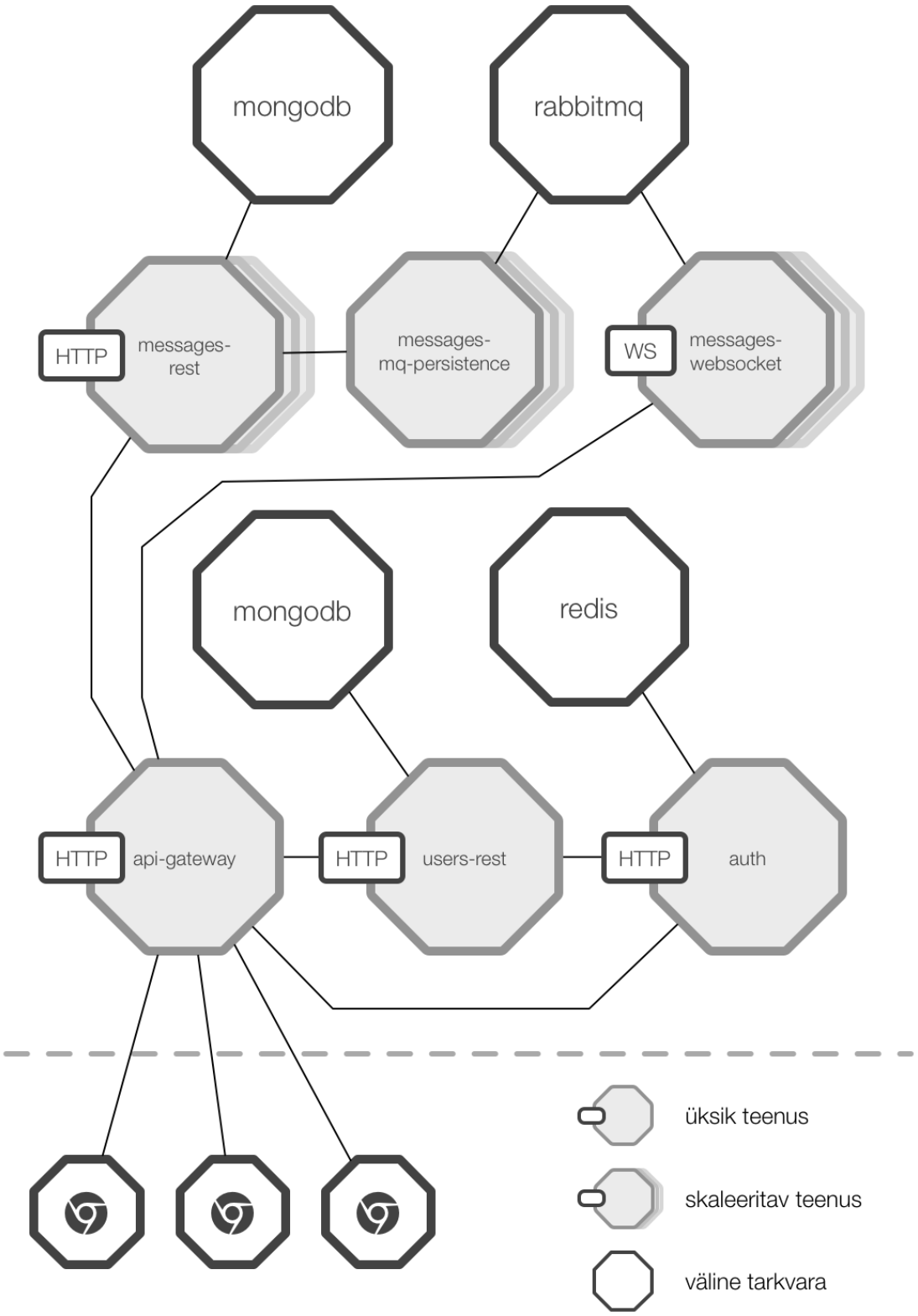
Alloleval joonisel on nimetatud konfiguratsioon. Sellel defineeritakse ära kaks serverite gruppi (*upstream*-blokk [43]) – *auth* (viitab identifitseerimisteenusele) ja *messages-websocket* (viitab sõnumite reaalajas vahendamise teenusele). Nendes blokkides on loetletud antud teenuste instantsid (joonisel kolm instantsi teenuse kohta), mille vahel koormust jaotatakse.

Upstream-blokkidele järgneb *server*-blokk [44], milles defineeritakse väline liides. Antud juhul on välise liidese aadressideks teenuste nimed.

```
events {}

http {
       upstream users-rest {
              server users-rest;
       }

       upstream messages-websocket {
              server messages-websocket-1;
              server messages-websocket-2;
              server messages-websocket-3;
       }

    server {
              listen 80;

              auth_request /auth/;
              auth_request_set $auth_user $upstream_http_x_authenticated_user;

              proxy_set_header X-Authenticated-User $auth_user;

              location /users-rest/ {…}

              location /messages-rest/ {…}

              location /messages-websocket/ {
                     auth_request off;

                     proxy_pass http://messages-websocket/;
                     proxy_set_header Upgrade $http_upgrade;
                     proxy_set_header Connection "upgrade";
                     proxy_read_timeout 7d;
              }

              location /auth/ {…}
    }
}
```

Joonis 16 nginx-serveri konfiguratsioon

4.7 Arhitektuuri diagramm



Joonis 17 Arhitektuuri diagramm

4.8 Edasine töö

Töö tulemusena valminud rakendus on küll täisfunktsionaalne, kuid oma funktsionaalsuse poolest minimaalne, kuna sisaldab vaid hädavajalikku, et teostada kahe kasutaja vahelist sõnumivahetust.

Üheks võimalikuks täienduseks on mitut kasutajat sisaldavad vestlused. Selle saavutamiseks tuleks teha väiksemaid muudatusi teatud teenustes:

- sõnumite säilitamise teenuses tuleks muuta andmemudelit – kaotada sõnumitelt adressaat (loeb ainult saatja – vestluses osalejad on teada),
- sõnumite reaalajas vahendamise teenus peaks olema teadlik vestluses osalevatest kasutajatest – hetkel pannakse iga sõnumiga kaasa adressaat,
- sisemad, pigem kosmeetilised täiendused kasutajaliideses.

Teise täiendusena on võimalik luua kasutaja kohalolu indikaator – selle toimimismehhanismiks oleks kasutaja brauseri ja WebSocket serveri vahelise ühenduse olemasolu kontrollimine. See nõuaks tõenäoliselt ühte lisateenust, et säilitada kasutaja kohalolu staatus ja loomulikult muudatusi kasutajaliideses.

Mis puudutab turvalisust, siis selles osas on võimalik teha mitmeid täiendusi. Esiteks tuleks paigaldada avalikule lüüsile turvaline HTTPS ühendus, milleks on vaja sertifikaate, mida on tänapäeval võimalik saada juba ilma rahata [45]. Teiseks ei krüpteeri kasutajate säilitamise teenus paroole, reaalses maailmas oleks see lubamatu. Kolmandaks oleks võimalik täiendada autentimis- ja kasutajate säilitamise teenust rollide funktsionaalsusega – hetkel luuakse ainult luba-kasutajanimi paar, kuid kasutajanime saaks asendada detailsema infoga, näiteks kasutajanimi ja tema rollid. Viimasena tuleks lubade genereerimise jaoks kasutada krüptograafilisi funktsioone täitvaid teeke.

Veel ühe olulise täiendusena tuleks rakendada vahemälu nendes kohtades, kus toimub tihe teenustevaheline andmevahetus. Töö käigus valminud rakenduses toimub selline pärimine ainult autentimisteenuses, kuid tunduvalt olulisem oleks see mitme kasutaja vestluste puhul, kus sõnumite reaalajas vahendamise teenus peaks iga saabunud sõnumi puhul pärima sõnumite säilitamise teenuselt vestluse infot ehk vestluses osalejaid.

Kokkuvõte

Töö eesmärgiks oli luua täisfunktsionaalne tarkvara - kiirsuhtlust võimaldav rakendus.

Kuna mikroteenuste näol on tarkvaraarenduse kontekstis tegu võrdlemisi uue lähenemisega, pidi autor kombineerima selleteemalistest artiklitest ja raamatutest saadud teadmisi ja teatud määral eksperimenteerima .

Esmalt avas töö autor mikroteenuste teoreetilise poole ehk mikroteenuste olemuse, tekkepõhjused, eelised ning puudused. Selgub, et mikroteenused pole kontseptsiooni poolest uus asi, vaidpigem loetakse seda teenus-orienteeritud arhitektuuri reinkarnatsiooniks, kuid sellegipoolest on nende kahe vahel olulised erinevused, nagu näiteks orkestreeritus teenus-orienteeritud arhitektuuris versus koreograafia mikroteenustes.

Töö sisulises osas analüüsiti mikroteenuste puhul kerkivaid probleeme, millest olulisemateks osutusid mikroteenuste omavaheline kommunikatsioon ja rakenduse skaleeruvus. Viimase juures tõi töö autor välja ka konkreetsed tehnoloogiad, mida autori arvates on sobilik kasutada väljatoodud probleemide lahendamiseks. Nendeks osutusid tänapäeval küllaltki levinud lahendused, mida teatud juhtudel võiks nimetada lausa de facto standardiks. Analüüsile toetudes asus autor töö eesmärgina seatud rakendust realiseerima.

Tulemuseks on mikroteenus-arhitektuuril baseeruv tarkvara, kus kasutajad saavad vahetada sõnumeid, mille kohaletoimetamisaeg on inimsuhtluse mõistes hetkeline.

Loodud rakendus on võimeline töötama hajutatult, mis tähendab, et iga teenuse instants võib paikneda füüsiliselt eraldiseisvas serveris. Lisaks sõltub rakenduse jõudlus otseselt instantside arvust – nende kriteeriumite alusel võib autori arvates lugeda püstitatud probleemi lahendatuks. Realisatsiooni detaile ehk teenuste liideseid, sisemist ehitust ja loogikat kirjeldatakse töö viimases osas.

Summary

Purpose of this thesis was to build a fully functional piece of software, which enables users to exchange instant messages.

To complete the task at hand the author had to rely on available articles and books and combine that knowledge with some amount of experimentation, since microservices are a relatively new thing.

First chapter gave an overview of the theoretical side of microservices – the essence of it, needs and forces, pros and cons. It appears that microservices are not something new at all, it is quite often compared to service-oriented architecture and it is thought that microservices are just fine-grained service-oriented architecture. But still, there are important differences between them like orchestration in service-oriented architecture and orchestration in microservices.

The author then proceeded to analyse the problems that surface when first encountering microservices. Two of the most important ones are communication between microservices and scalability. Possible technologies which could be used to solve these problems were also listed – turns out that most of these technologies are already a de facto standard regarding microservices. Armed with this knowledge the author started implementing the software application.

Result is a microservice-based application, where users can send instant messages to one and another. This application is able to work in a distributed environment. It means that every instance of a service can be placed on a physically separate server. Furthermore, performance of the application directly depends on the number of instances. Regarding these two criteria, the author considers goals of this thesis accomplished. Implementation details are explained in the last part of the work.

Kasutatud kirjandus

[1] M. Fowler, „Microservices,“ [Võrgumaterjal]. Available: http://martinfowler.com/articles/microservices.html. [Kasutatud 13 04 2016].

[2] P. Rodgers. [Võrgumaterjal]. Available: http://www.cloudcomputingexpo.com/node/80883. [Kasutatud 05 05 2016].

[3] S. Daya, N. Van Duy, K. Eati, C. Ferreira, D. Glozic, V. Gucer, M. Gupta, S. Joshi, V. Lampkin, M. Martins, S. Narain ja R. Vennam, Microservices from Theory to Practice, Redbooks, 2015.

[4] B. Morris, „How big is a microservice?,“ [Võrgumaterjal]. Available: http://www.ben-morris.com/how-big-is-a-microservice/. [Kasutatud 05 05 2016].

[5] J. Eaves, „Micro services, what even are they?,“ [Võrgumaterjal]. Available: http://techblog.realestate.com.au/micro-services-what-even-are-they/. [Kasutatud 05 05 2016].

[6] M. Fowler, „Microservice Trade-Offs,“ [Võrgumaterjal]. Available: http://martinfowler.com/articles/microservice-trade-offs.html. [Kasutatud 06 05 2016].

[7] The Open Group, „Service Oriented Architecture,“ [Võrgumaterjal]. Available: http://www.opengroup.org/soa/source-book/soa/soa.htm. [Kasutatud 06 05 2016].

[8] A. Wang ja S. Tonse, „Announcing Ribbon: Tying the Netflix Mid-Tier Services Together,“ [Võrgumaterjal]. Available: http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html. [Kasutatud 06 05 2016].

[9] M. Richards, „Microservices vs. Service-Oriented Architecture,“ O’Reilly Media, 2016.

[10] C. Richardson, „Microservices: Decomposing Applications for Deployability and Scalability,“ [Võrgumaterjal]. Available: http://www.infoq.com/articles/microservices-intro. [Kasutatud 13 05 2016].

[11] OASIS, „AMQP,“ [Võrgumaterjal]. Available: https://www.amqp.org. [Kasutatud 12 04 2016].

[12] Ecma International, [Võrgumaterjal]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf. [Kasutatud 22 04 2016].

[13] Google, „Protocol Buffers,“ [Võrgumaterjal]. Available: https://developers.google.com/protocol-buffers/. [Kasutatud 21 04 2016].

[14] Apache, „Apache Thrift,“ [Võrgumaterjal]. Available: https://thrift.apache.org. [Kasutatud 21 04 2016].

[15] C. Richardson, „Microservices.io,“ [Võrgumaterjal]. Available: http://microservices.io/patterns/apigateway.html. [Kasutatud 12 04 2016].

[16] NGINX, „Introduction to microservices,“ [Võrgumaterjal]. Available: https://www.nginx.com/blog/introduction-to-microservices/. [Kasutatud 13 04 2016].

[17] C. Richardson, „Microservices - database per service,“ [Võrgumaterjal]. Available: http://microservices.io/patterns/data/database-per-service.html. [Kasutatud 12 04 2016].

[18] S. Newman, Building Microservices, O'Reilly Media, Inc., 2015.

[19] Pivotal Software, Inc., „Consumer priorites," [Võrgumaterjal]. Available: https://www.rabbitmq.com/consumer-priority.html. [Kasutatud 13 04 2016].

[20] npm, Inc., „npm," [Võrgumaterjal]. Available: https://www.npmjs.com. [Kasutatud 13 04 2016].

[21] Ecma International, „ECMAScript 2015 Language Specification," [Võrgumaterjal]. Available: http://www.ecma-international.org/ecma-262/6.0/. [Kasutatud 22 04 2016].

[22] „Babel · The compiler for writing next generation JavaScript," [Võrgumaterjal]. Available: https://babeljs.io. [Kasutatud 14 04 2016].

[23] MongoDB, „MongoDB," [Võrgumaterjal]. Available: https://www.mongodb.org. [Kasutatud 21 04 2016].

[24] „Express - Node.js web application framework," [Võrgumaterjal]. Available: http://expressjs.com. [Kasutatud 14 04 2016].

[25] squaremo, „AMQP 0-9-1 library and client for Node.JS," [Võrgumaterjal]. Available: https://github.com/squaremo/amqp.node. [Kasutatud 14 04 2016].

[26] mongodb, „Mongo DB Native NodeJS Driver," [Võrgumaterjal]. Available: https://github.com/mongodb/node-mongodb-native. [Kasutatud 14 04 2016].

[27] Facebook, „A JavaScript library for building user interfaces | React," [Võrgumaterjal]. Available: https://facebook.github.io/react/. [Kasutatud 14 04 2016].

[28] „Read Me | Redux," [Võrgumaterjal]. Available: http://redux.js.org. [Kasutatud 14 04 2016].

[29] Docker, „Docker - Build, Ship, and Run Any App, Anywhere," [Võrgumaterjal]. Available: https://www.docker.com. [Kasutatud 14 04 2016].

[30] Docker, „Dockerfile reference," [Võrgumaterjal]. Available: https://docs.docker.com/engine/reference/builder/. [Kasutatud 14 04 2016].

[31] Pivotal Software, Inc., „RabbitMQ - Messaging that just works," [Võrgumaterjal]. Available: https://www.rabbitmq.com. [Kasutatud 14 04 2016].

[32] nginx, „nginx," [Võrgumaterjal]. Available: http://nginx.org. [Kasutatud 21 04 2016].

[33] Apache, „mod_proxy," [Võrgumaterjal]. Available: https://httpd.apache.org/docs/current/mod/mod_proxy.html. [Kasutatud 21 04 2016].

[34] HAProxy, „The PROXY protocol," [Võrgumaterjal]. Available: http://www.haproxy.org/download/1.5/doc/proxy-protocol.txt. [Kasutatud 21 04 2016].

[35] redislabs, „Redis," [Võrgumaterjal]. Available: http://redis.io. [Kasutatud 22 04 2016].

[36] redislabs, „FAQ - Redis," [Võrgumaterjal]. Available: http://redis.io/topics/faq. [Kasutatud 22 04 2016].

[37] MongoDB, Inc., „Schema Design for Social Inboxes in MongoDB," [Võrgumaterjal]. Available: http://blog.mongodb.org/post/65612078649/schema-design-for-social-inboxes-in-mongodb. [Kasutatud 14 04 2016].

[38] websockets, „ws: a node.js websocket library," [Võrgumaterjal]. Available: https://github.com/websockets/ws. [Kasutatud 22 04 2016].

[39] websockets, „ws API reference,“ [Võrgumaterjal]. Available: https://github.com/websockets/ws/blob/master/doc/ws.md. [Kasutatud 19 04 2016].

[40] squaremo, „amqlib - channel API reference,“ [Võrgumaterjal]. Available: http://www.squaremobius.net/amqp.node/channel_api.html#channel_assertQueue. [Kasutatud 19 04 2016].

[41] redislabs, „EXPIRE - Redis,“ [Võrgumaterjal]. Available: http://redis.io/commands/expire. [Kasutatud 13 05 2016].

[42] redislabs, „TTL - Redis,“ [Võrgumaterjal]. Available: http://redis.io/commands/ttl. [Kasutatud 13 05 2016].

[43] nginx, „ngx_http_upstream_module,“ [Võrgumaterjal]. Available: http://nginx.org/en/docs/http/ngx_http_upstream_module.html. [Kasutatud 22 04 2016].

[44] nginx, „ngx_http_core_module,“ [Võrgumaterjal]. Available: http://nginx.org/en/docs/http/ngx_http_core_module.html#server. [Kasutatud 22 04 2016].

[45] Internet Security Research Group, „Let's Encrypt - Free SSL/TLS Certificates,“ [Võrgumaterjal]. Available: https://letsencrypt.org. [Kasutatud 20 05 2016].