TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Jakob Roots 179094IADB

Tarkvarasüsteemi võrguliikluse reguleerimise lahendus teleturundusettevõtte näitel

Bakalaureusetöö

Juhendaja: Priit Raspel

MSc

Tallinn 2021

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: Jakob Roots

17.05.2021

Tallinn 2021

Annotatsioon

Käesoleva töö eesmärgiks on kavandada ning luua süsteemi prototüüp, mis on võimeline reguleerima klientide võrguliiklust tarkvarasüsteemis.

Töö esimene osa koosneb ettevõtte süsteemsete ning äriliste nõuete kirjeldamisest, lahendatava probleemi tausta tutvustamisest ning süsteemi kavandamise ja ehitamisega seotud tehnoloogiate ülevaatest.

Teine osa, ehk analüüsi osa, kirjeldab tehnoloogiliste ja arhitektuuriliste valikute analüüsi, mis on aluseks lahenduse kavandamisel.

Kavandis kirjeldatakse süsteemi kompositsiooni kasutades analüüsi osas tehtud arhitektuurilisi ja tehnoloogilisi valikuid. Komponendid ja nende omavahelised suhted on kirjeldatud ükshaaval ning tulemuseks on süsteemi lõplik kavand.

Rakenduse osas realiseeritakse kavandi põhjal süsteem ning kirjeldatakse realiseerimiseks vajalikud tegevused. Kõige viimasena viiakse läbi koormustest, mille tulemuste analüüsiga otsustatakse prototüübi võimekus.

Viimases, ehk järelduste osas, tuuakse välja süsteemi väljaehitamise kulud ning potentsiaalsete ülal hoiu kulude analüüs. Lisaks tehakse ettepanekuid süsteemi parendamiseks ja edasiarenduseks.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 44 leheküljel, 7 peatükki, 16 joonist, 11 tabelit.

Abstract

Application Network Traffic Regulation Solution by Example of a Telemarketing Company

The goal of this thesis is to design and implement a prototype for a system which would be able to regulate the network traffic of customers in a software system.

The first part of the work consists of describing the systemic and business requirements of the solution, describing the background of the probleem and the general overview of the technologies related to the design and implementation of the system.

The second part, which is the analysis part, describes the study of choices for the technologies and architecture on which the design of the system will depend on.

In the design chapter of the work, the system composition is described using the choices made in the previous chapter. The components and their relationships in the system are described one by one to form the final design of the system.

The system is realized in the implementation chapter using the design from the previous chapter. All of the activities necessary for the implementation are described in the chapter. In the last part of the chapter, a load test is conducted and the results of it are analysed to determine the viabilty of this prototype.

In the conclusion chapter, the cost of building the prototype and the potential costs of running it in a production environment are analysed. The chapter also includes a part about the potential betterments for the system and the propositions for further development.

The thesis is in Estonian and contains 44 pages of text, 7 chapters, 16 figures, 11 tables.

Lühendite ja mõistete sõnastik

API	Programmiliides, ehk arvutiprogrammides alamprogrammi määratluste, protokollide ja tööriistade komplekt rakendustarkvara ehitamiseks
REST	Tarkvaraarhitektuuri laad, mis kasutab HTTP päringuid
SLA	Teenusetaseme leping
Sõnum	Kliendi poolt edastatud päringu sisu
Thread	Lõim, ehk programmi omadus jaotuda mitmeks protsessiks
Ülesvoolu teenus	Mikroteenus, mis on käsitletavale mikroteenusele andmeid edastavas rollis
Allavoolu teenus	Mikroteenus, millele käsitletav teenus andmeid edastab
OSI	Avatud süsteemide sidumise arhitektuur
AWS	AWS on pilveteenuste platvorm, mis pakub suure variatsiooniga globaalseid pilveteenustel põhinevaid tooteid
EC2	Amazon Elastic Compute Cloud on veebiteenus, mis pakub turvalisi, muutuva mastaabiga pilveservereid
ECS	Amazon Elasric Container Service on veebiteenus, mis võimaldab konteinerite orkestreerimist
SQS	Amazon Simple Queue Service, on täielikult hallatud sõnumite järjekorra ja edastus süsteem
ALB	Application Load Balancer on 7. OSI kihi peal töötav koormusjaotur
NLB	Application Load Balancer on 4. OSI kihi peal töötav koormusjaotur
DynamoDB	Amazon DynamoDB on võti-väärtus ja dokumentide andmebaas
DAX	Amazon DynamoDB Accelerator on täielikult hallatud, kiirelt kättesaadav vahemälu süsteem Amazon DynamoDB andmebaasi jaoks

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Tänapäevases infotehnoloogia maailmas on kiirus ning kättesaadavus ühed kõige tähtsamad mõõdikud internetiteenuste kvaliteedi mõõtmises. Kuid need kaks terminit on kohati üksteisele vasturääkivad. Väga kiire süsteem võib iseseisvalt hästi töötada, kuid alati on risk, et mõni süsteemiga seotud osa ei suuda sama hästi toimida. Väga kättesaadav süsteem peab ennast aga kaitsma liigse ülekoormuse eest ning seetõttu ei ole alati kõige kiirem.

Käesolevas töö eesmärk on ehitada prototüüpsüsteem, mis loob hea kompromissi nii kiiruse kui ka kättesaadavuse osas. Kavandatav prototüüp peab olema kõrge kättesaadavusega, et tagada klientide rahulolu. Lisaks peab prototüüp olema ka võimeline tagama klientidega kokkulepitud teenusepakkumise kiirust ning kaitsma ülejäänud ettevõtte süsteeme ülekoormuse eest.

1.1 Ülesande püstitus

Lõputöö ülesanne on koostatud anonüümse teleturunduse ettevõte näitel. Ettevõte tegeleb suures mahus SMS ning MMS sõnumite edastamisega ja kasutab selleks kohalike telekommunikatsiooni võrgustike. Ettevõtte klientideks on firmad, kes kasutavad ettevõtte pakutavaid teenuseid nii reklaamide edastamiseks kui ka klientide autentimiseks. Tõrked ettevõtte teenustes tähendavad enamikul juhtudel rahalist kahju klientide äritegevusele.

Ettevõttel on kohustus reguleerida oma võrguliiklust, et mitte üle koormata sõnumite edastust pakkuvaid kohalike telekommunikatsiooni ettevõtteid. Iga telekommunikatsiooni ettevõttega on lepingus määratud liikluse tempo ülemmäär. Lepingut rikkudes võib teenusepakkuja sulgeda uute sõnumite vastuvõtmise.

Ettevõttel on lisaks veel sisemised võimekuse piirangud. Pahatahtlikel klientidel on teoreetiliselt võimalik enda kasutusse võtta kogu teenuse võimekus, takistades teistele klientidele ligipääsu teenusele.

- Firma suuruse tõttu on kliente ja liiklust palju. Üheks kindlaks nõudeks on, et iga süsteem firmas on skaleeritav ning toimib efektiivselt ka suure koormuse all.
- Klientidele kehtivad mitmed erinevad liikluspiirangud korraga. Süsteem peab austama alati kõige suuremat piirangut ajahetkel ning tagama piirangu sellele vastavalt.
- Piirangutele määramine sõnumitele peab toimuma väga kiiresti. Piirang peab ideaalis peale määramist koheselt kehtima hakkama, kuid on ka vastuvõetav mõnekümne sekundiline hilinemine.
- Süsteem peab toetama sõnumite kordust ning elustamist, kui nende töötlemisel tekib tõrkeid.
- Süsteem peab suutma tagada ligi 100% täpsusega klientidele seatud limiite. Suurema liikluse puhul on oht süsteemile ning vähema liikluse puhul on oht kliendile.
- Süsteem peab olema täielik ning iseseisev klaster. Süsteemi peab olema võimalik juurutada olemasolevate süsteemide vahele. Selleks peab süsteem võimaldama suurt valikut sättetes ning olema võimalikult abstraktne.

Täpselt nendele nõuetele vastavat süsteemi maailmas müügil ei ole. Seetõttu tuleb kavandada süsteem mitmetest erinevatest tööriistadest ning tehnoloogiatest.

Töö üheks tulemuseks on analüüsitud teoreetiline süsteem, mis vastab ettevõtte nõuetele. Analüüsis võrreldakse erinevaid võimalusi ettevõtte probleemi lahendamiseks. Süsteem tuleb ka praktiliselt realiseerida. Praktiline osa pakub tagasisidet analüüsis püstitatud hüpoteesidele. Süsteemi võimekust ning töökindlust on vaja ka testida ning tulemusi analüüsida. Selleks tuleb läbi viia koormustest.

Praktilise osa tulemusel on ka päris äriline kasu. Kui süsteem osutub efektiivseks ning töövõimeliseks, on suur tõenäosus, et see võetakse ka praktikas kasutusse. Tulevikus tagab süsteem ettevõtte klientidele kindlama ning stabiilsema teenuse.

Kui süsteem ei rahulda ette seatud nõudeid, hoitakse ära suur ajaline kadu kui seda ettevõttes suurema investeeringuga realiseeritud oleks.

2 Taust

Käesolevas peatükis antakse ülevaade töö taustast ja probleemidest. Peatükis kajastatakse süsteemi kvaliteedi tagamise kokkuleppeid, võrguliikluse reguleerimise tehnoloogiaid ja võimalusi, algoritmilisi lahendusi reguleerimiseks ettevõtte poolseid nõudeid ja probleemi tausta.

2.1 Ärilised vajadused ja nõudmised

Peatükis tuuakse välja tööle seatud piirangud ning nõudmised, mis tulenevad töös näiteks võetud ettevõtte vajadustest.

2.1.1 Äriline Taust

Ettevõte pakub klientidele võimalust edastada reklaamikampaaniaid, mille läbiviimiseks kasutatakse SMS sõnumeid. Ettevõte soovib pakkuda klientidele sõnumite liikluse tempo määra kui teenust, kuid ei taha keelata klientidele sõnumite edastust kui nende hetke edastusmäär on lubatust suurem. Klientide sõnumite tagasilükkamine igakord kui nad piirmäära ületavad tähendaks klientidele väga palju peavalu ning ebamugavat kasutajakogemust, kuna peaksid väga palju tegelema veakoodide töötlemise ning uuesti sõnumite edastamisega. Seetõttu vajab ettevõte süsteemi, mis oleks võimeline kliendi sõnumeid vastu võtma igal hetkel, kuid edastaks neid vastavalt kokkulepitud piirangutele.

2.1.2 Ärilised nõuded

Klientidele kehtivad mitmed erinevad liikluspiirangud korraga. Süsteem peab austama alati kõige suuremat piirangut ajahetkel ning tagama piirangu sellele vastavalt.

Süsteem peab suutma tagada ligi 100% täpsusega klientidele seatud limiite. Suurema liikluse puhul on oht süsteemile ning vähema liikluse puhul on oht kliendile.

Piirangutele määramine sõnumitele peab toimuma väga kiiresti. Piirang peab ideaalis peale määramist koheselt kehtima hakkama, kuid on ka vastuvõetav mõnekümne sekundiline hilinemine.

Süsteem peab toetama sõnumite kordust ning elustamist, kui nende töötlemisel tekib tõrkeid.

Kliendi liikluspiirangud on väärtused, mis esindavad kliendi maksimaalset teenusekasutuse määra kindlal ajaühikul. Käesolevas töös on piirangute all mõeldud sõnumite edastamise hulka ühes sekundis. Andmebaasis on piirangud kirjeldatud kui võti väärtus paarid, kus võtmeks on sõnumite kategoriseerimise parameeter ja väärtuseks on lubatud kasutuse ülemmäär.

2.1.3 Süsteemsed nõuded

Süsteem peab olema täielik ning iseseisev klaster. Süsteemi peab olema võimalik juurutada olemasolevate süsteemide vahele. Selleks peab süsteem võimaldama suurt valikut sättetes ning olema võimalikult abstraktne.

Firma suuruse tõttu on kliente ja liiklust palju. Üheks kindlaks nõudeks on, et iga süsteem firmas on skaleeritav ning toimib efektiivselt ka suure koormuse all.

Süsteemi osad peavad olema üksteisest lahti seotud. Ühe komponendi probleem ei tohiks mõjutada teise komponendi tööd.

Süsteem peab asuma Amazon Web Services võrgutaristus.

2.2 Teenusekvaliteet

Teenusekvaliteet ehk Quality of Service (lühendatult QoS) on võimekus kontrollida liikluse haldamise mehhanisme võrgus nii et võrk vastaks kindlatele võrgu eeskirjadele alluvate teenuste ja kasutajate poolsetele vajadustele. QoS võrkudel peavad olema mehhanismid, mis võimaldaksid kontrollida ressursside eraldamist teenuste ja kasutajate vahel [1].

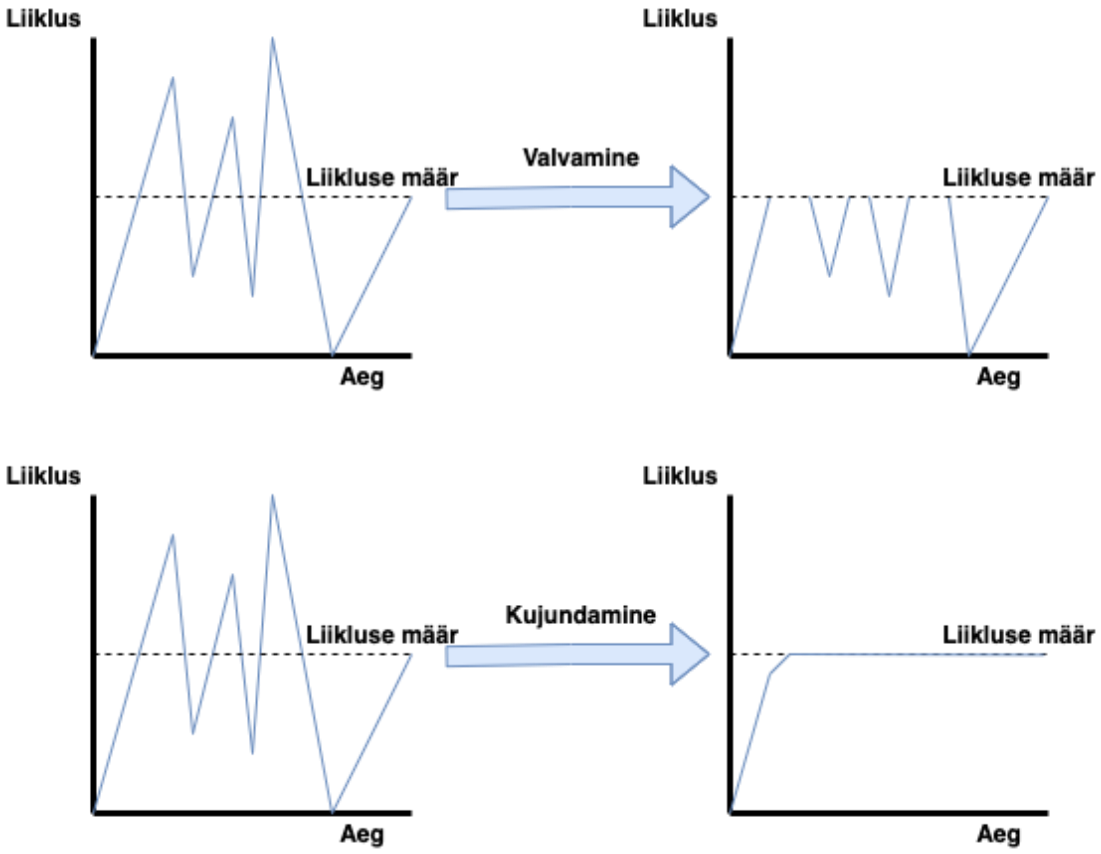
QoS kui termin tuli kasutusele tänu moodsatele rakendustele, mis panid uued nõuded võrgu jõudlusele, näiteks nagu reaalajas multimeedia teenused. Sellised rakendused lõid vajaduse defineerida võrgus edastatavate andmete vastuvõetavad hilinemise ajad [1].

QoS mõõdetakse tavaliselt neljas kategoorias: ribalaius (ülekandevõime e. andmehulk ajaühikus), viide ehk hilinemine, värin ehk hilinemiste variatsioon, kadu ehk pakettide protsent, mis läheb teel kaduma [1].

2.3 Liikluse reguleerimine

Liikluse reguleerimine on tehnika, millega reguleeritakse keskmist andmete sisenemise tempot ning järske tempo variatsioone. Eesmärk on lubada kasutajal edastada varieeruva koguse ja kiirusega andmeid. Kui andmeedastus voog on üles seatud siis kasutaja ning teenus lepivad kokku kindla liiklusmustri sellele voole. Sellist lepingut kutsutakse SLA lepinguks [2].

Liikluse reguleerimiseks on kaks varianti: liikluse kujundamine (*shaping*) ning liikluse valvamine (*policing*). Valvamise korral loobutakse regulatsioone ületavatest sõnumitest. Kujundamise puhul ühtlustatakse ülevoolavate sõnumite voog üle pikema aja vastavalt piirangule [2]. Liikluse kujundamine ja valvamine on toodud välja visuaalselt Joonisel 1.



Joonis 1. Võrgu reguleerimise tehnikad.

2.4 Reguleerimise algoritmid

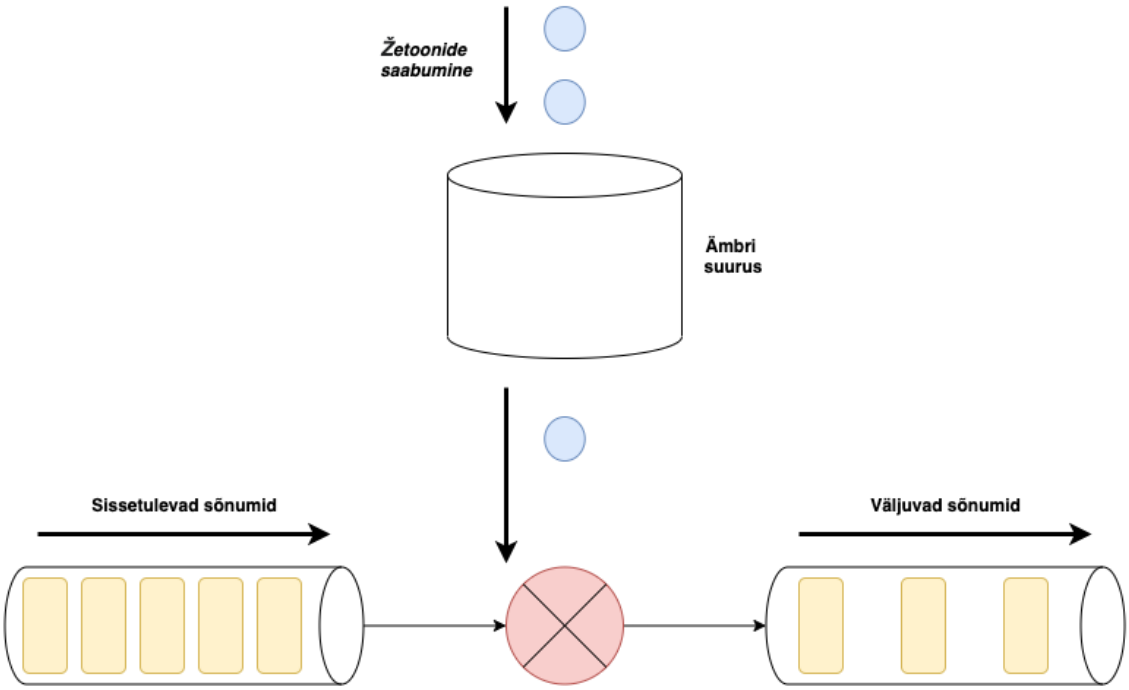
Liikluse reguleerimiseks on kasutusel kaks peamist algoritmi. Need algoritmid limiteerivad pikaajalist liiklusvoo määra, aga lubavad lühiajalisi variatsioone maksimaalse sätestatud määrani [3].

2.4.1 Lekkiv ämber

Lekkiv ämber algoritm töötab analoogselt veeämbrile, millel on alumisel küljel auk. Ükskõik millise kiirusega vesi ämbrisse siseneb, väljavoolamine on alati konstantse määraga. Kui ämber saab täis, siis üldiselt kõik üleliigne vesi kaob üle ääre. Seda algoritmi kasutatakse liikluse valvamiseks ja kujundamiseks. Kui päring saabub ämbrisse täis olekus, tuleb otsustada kas lisada päring järjekorda või loobuda edastamisest [3].

2.4.2 Žetoon ämbris

Žetoon ämbris algoritm töötab analoogselt ämbrile, kuhu voolab konstantselt sisse žetoone, kuid mitte rohkem kui ämber hoida suudab. Enne kui päringut on võimalik edastada, tuleb võtta ämbrist žetoon. Kuna ämbris on võimalik hoida maksimaalselt ainult limiteeritud hulk žetoone, võib tekkida olukord kus ämber on tühi. Ämbri tühja oleku korral tuleb oodata kuni uued žetoonid ämbrisse jõuavad [3]. Žetoon Ämbris algoritm on toodud välja visuaalselt Joonisel 2.

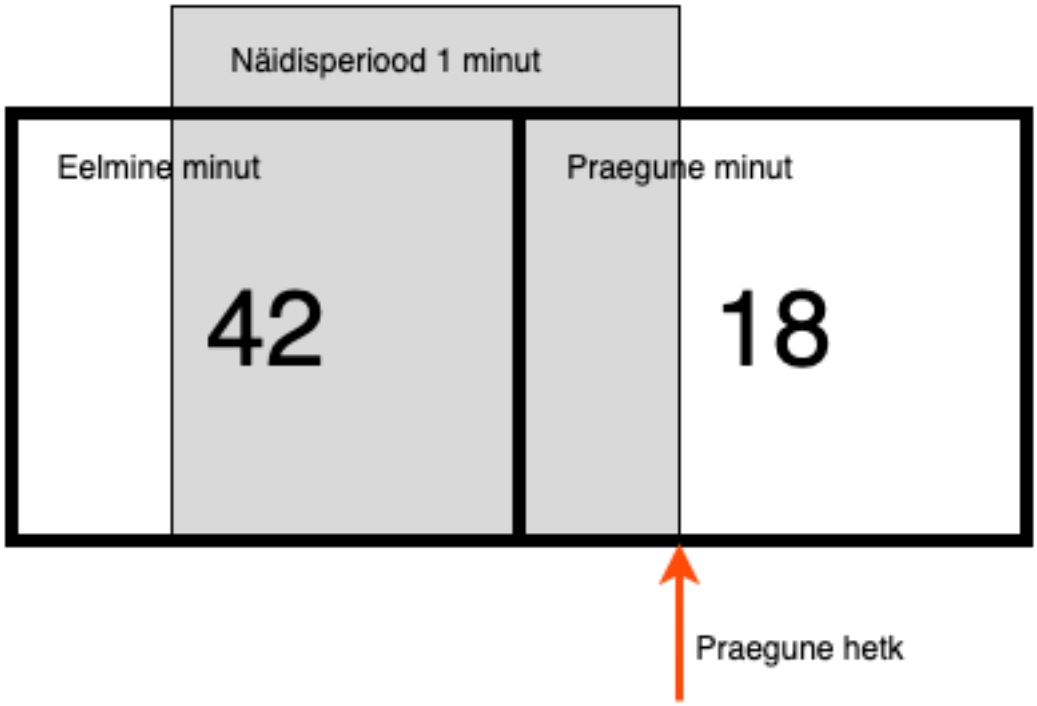


Joonis 2. Žetoon Ämbris algoritmi joonis.

2.4.3 Libisev aken

Libiseva akna algoritmis jagatakse aeg kindlateks mõõtmisvahemikeks ja määratakse igale vahemikule maksimaalne arv päringuid. Päringu saabudes arvutatakse kehtivale ja eelmisele ajavahemikule nende protsentuaalne tähtsus vastavalt sellele kui suur osa viimase päringu saabumise ajast loetud mõõtmisvahemikust asub eelmises vahemikus ja kui palju praeguses vahemikus [4].

Näiteks võib tuua API lõppsõlme, mille piirang on 50 päringut sekundis. Joonisel 3 on see kujutatud visuaalselt [4].



Joonis 3. Libisev aken algoritm visualiseeritud kujul [4].

Praeguses minutis, mis algas 15 sekundit tagasi, saabus 18 päringut. Eelmises minutis saabus kokku 42 päringut. Kasutades neid andmeid on võimalik kalkuleerida liiklusmäär [4].

$$Liiklusmäär = 42 * \left(\frac{60-15}{60}\right) + 18 = 42 * \frac{3}{4} + 18 = 49.5 \frac{päringut}{minutis} \quad (1)$$

Valem 1 esitab algoritmiga hetkepiirangu arvutust. Selle arvutuse tulemus kuvab päringu saabumise hetkest arvutatud viimase minuti liikluse määra, mis antud juhul on 49.5 päringut minutis. Rohkem päringuid sellesse minutisse ei mahuks [4].

2.5 Bucket4j

Bucket4j teek on Java programmeerimis keelel põhinev Žetoon Ämbris algoritmi teostus. Teek on ehitatud üles toetama mastaapseid jaotatud lõimedega rakendusi. Bucket4j ei kasuta komakohtadega arve, et vältida ümardamisel tekkivad ebatäpsuseid. Lisaks võimaldab teek veel kirjeldada ühele piirangu olemile mitmeid erinevaid piiranguid, näiteks nagu 1000 sündmust tunnis, aga mitte rohkem kui 100 sündmust minutis [5].

Bucket4j toetab JCache API, mis võimaldab ühendamist iga sellele standardile vastava mälusisese andmevõrgu rakendusega kasutades kaht rida koodi. Kodulehel on välja toodud toetatud andmevõrgud nagu Hazelcast, Apache Ignite, Infinispan, Oracle Coherence [5].

Teek võimaldab pärida liikluse hetkel võimaliku määra, žetoonide arvu ja aega järgmise žetooni ämbrisse jõudmiseni [5].

2.6 Mikroteenuste arhitektuur

Nõuete järgi peab süsteem kasutama mikroteenuste arhitektuuri. Antud peatükis toob autor välja mõned mikroteenuste arhitektuuri osad, mis haakuvad töös käideldava teemaga.

2.6.1 Komponendid teenustena

Komponent on ühik tarkvara, mis on iseseisvalt väljavahetatav ja uuendatav. Mikroteenuste arhitektuurid küll kasutavad ka teeke, kuid nende põhiline komponentideks jaotamise metoodika on teenusteks jagamine. Teenused on protsessivälised komponendid, mis suhtlevad omavahel näiteks veebipäringute kaudu [6].

2.6.2 Mikroteenuste suhtlus

Mikroteenuste omavahelise suhtluse korraldamist eelistatakse mikroteenuste kogukonnas vaadata kui tarku lõpp punkte ja rumalaid torusid. Mikroteenuste arhitektuuri järgi ehitatud rakendused peavad oma olemuselt olema väga lahti ühendatud ja ühtsed. Teenused peavad vastutama oma domeeni loogika eest ja olema nagu filtrid Unix maailmas - võtma vastu päringuid, rakendama loogikat ja tootma vastuse [6].

Kaks kõige populaarsemat protokolli suhtlemise jaoks on HTTP päring-vastus API-ga ja kerge kaaluga sõnumside. HTTP protokolli kasutatakse sarnaste põhimõtetega millele world wide web tugineb. Ressursse on tihti võimalik väga lihtsalt vahemälus salvestada. Teine lähenemine ehk sõnumside on kasutusel üle kerge sõnumibussi. Taristu, mida selle jaoks kasutatakse on tihti ainult sõnumite marsruuter. Sõnumside pakub teenuste suhtlusele vastupidavat asünkroonset kihti [6].

2.6.3 API väravrakendus

API Väravrakendus (API Gateway) on API haldamise tööriist, mis asub kliendi ja ettevõtte serverite vahel. Väravrakendus võtab vastu klientide API päringuid, kutsub välja vajalikud rakendused nende töötlemiseks, ning tagastab kliendile vastuse [7].

Väravrakendust on vaja mikroteenuste arhitektuuri kasutades, kuna tihti on vaja mitut teenust, et töödelda kliendi päringuid. See võimaldab peita klientide eest ettevõtte süsteemi keerukust ning muutusi, avaldades kliendile ainult ühe suhtluspunkti [7].

Väravrakendus tagab lihtsama järel valve süsteemile, sest klientidega seotud seireandmed tulevad kõik ühest süsteemist [7].

2.7 Amazon Web Services

Nõuetest tulenevalt peab töös kavandatav süsteem asuma Amazon Web Services (AWS) võrgutaristus. Selles peatükis toob autor välja mõned valitud AWS platvormi teenuste ülevaated ning nende kasutamise hinnad.

AWS on pilveteenuste platvorm, mis pakub suure variatsiooniga globaalseid pilveteenustel põhinevaid tooteid, nagu näiteks tooted kalkuleerimiseks, andmete hoiustamiseks, andmete analüüsimiseks, arendustööriistu, IoT ehk asjade internet ja palju muud. Teenused on saadavad vastavalt vajadusele, kiirelt ülesseatavad ja hind sõltub vastavalt kasutusele. Selline mudel lubab ettevõtetel reageerida kiirelt muutuvale ärimaailmale ja vältida suuri süsteemide ülesseadmisega seonduvaid ettemakse [8].

2.7.1 Amazon EC2

Amazon EC2 ehk Amazon Elastic Compute Cloud on veebiteenus, mis pakub turvalist, muutuva mastaabiga pilveservereid. EC2 eesmärk on pakkuda klientidele võimekust hankida pilveservereid ja arvutusvõimsust minimaalse takistusega kasutades EC2 veebiliidest. Kasutajal on täielik kontroll pilves asuvate arvutiressursside üle [8].

EC2 kulu sõltub teenuse kasutusest. Hind sõltub serveri tüübist ja kasutustundide arvust. Autor toob Tabelis 1 välja kahte tüüpi EC2 serverit, mida antud töös arenduseks ja testimiseks kasutatakse [9].

Tabel 1. EC2 hinnatabel.

Tüüp	Hind tunnis ($)	Kasutuseesmärk
t2.micro	0,0125	Süsteemi ehitamine ja eksperimenteerimine
t2.xlarge	0,2016	Koormustestimine

2.7.2 Amazon Fargate

Amazon Fargate on võrgumootor Amazon ECS jaoks, mis võimaldab hallata konteinereid ilma vajaduseta hallata individuaalseid servereid. Fargatei kasutades ei ole vaja virtuaalmasinate klastreid vaja manuaalselt üles seada ega mastaape muuta. Fargatei kasutamiseks on vaja teenused konteineriseerida, kirjeldada mälu ja portsessori võimekuse nõuded ning võrgureeglid [8]. Tabelis 2 on välja toodud mõned võimalikud Fargate teenuse seaded ja nende kulu [10].

Tabel 2. Amazon Fargate teenuse hinnatabel.

Virutaalsete CPU-de arv	Mälu suurus (GB)	Hind tunnis ($)
0,25	1	0,01901
0,5	2	0,02913
1	4	0,05826

2.7.3 Amazon SQS

Amazon SQS ehk Amazon Simple Queue Service, on täielikult hallatud sõnumite järjekorra ja edastus süsteem, mis võimaldab lahti siduda mikroteenuste suhtlust. SQS eemaldab palju sõnumiside vahevara haldamisega seostuvat keerukust ja lisakulu. SQS võimaldab sõnumite edastamist, salvestamist ja pärimist igal koormusel ilma sõnumite kaota ja ilma teiste teenuste saadaval olemise vajaduseta [11].

SQS pakub kahte sorti sõnumite järjekordi. Standard järjekord ehk Standard queue pakub maksimaalset läbilaskevõimet, parima pingutusega järjestust ja vähemalt-korra edastust. SQS FIFO järjekorrad on mõeldud sõnumite kindla järjekorraga edastamiseks ja täpselt ühekordseks edastamiseks [11].

Amazon SQS hinna mudel sõltub teenuse kasutusest. Hinnas mängib rolli ainult tehtud päringute arv. Tabelis 3 on välja toodud AWS SQS teenusekasutuse kulud. Andmed võetud AWS regioonist Euroopa (Iirimaa) [22].

Tabel 3. AWS SQS hinnatabel.

	Standard järjekord (Standard queues) (miljoni päringu hind, $)	**Järjestatud järjekord (FIFO queues) (miljoni päringu hind, $)**
Esimesed 1 mln päringut kuus	Tasuta	Tasuta
1 mln kuni 200 mld päringut kuus	0,40	0,50
100 mld kuni 200 mld päringut kuus	0,30	0,40
Üle 200 mld päringu kuus	0,24	0,35

2.7.4 Amazon DynamoDB

Amazon DynamoDB on võti-väärtus ja dokumentide andmebaas mis võimaldab alla 10 millisekundilist päringute kiiruse võimekust igal skaalal. DynamoDB on täielikult hallatud, mitme regiooniline süsteem, kuhu on sisse ehitatud turvalisus, tagavara haldus ja andmebaasi taastamise funktsionaalsus. Andmebaas võimaldab rohkem kui 10 triljonit päringut päevas ja 20 miljonit päringut sekundis [12].

Amazon DynamoDB pakub hästi dokumenteeritud API juhendit, mille näidete abil on võimalik süsteemi integreerimine kiirelt ära õppida. API on kirjutatud C++, Go, Java, JavaScript, PHP, Python ja Ruby programmeerimiskeeletejaoks ning ka .NET raamistiku jaoks [13].

Amazon DynamoDB võimaldab vajaduspõhist ja eelvarustatud hinna mudelit. Antud juhul uurib autor vajaduspõhist mudelit, kuna eelvarustatud mudeli arvestamiseks on vaja teada eeldatav liikluse suurus. Erinevatel andmebaasi operatsioonidel on erinevad hinnad ning lisaks maksab veel andmebaasi suurus. Tabelis 4 kuvatud andmed on võetud AWS regioonist Euroopa (Iirimaa) [14].

Tabel 4. AWS DynamoDB hinnatabel.

Teenus	**Vajaduspõhine mudel**
Kirjutamine andmebaasi	1,4135 $ / 1mln päringut
Lugemine andmebaasist	0,283 $ / 1 mln päringut
Andmebaasi suurus	25 GB on tasuta, 0,283 $ / 1 GB / 1 kuu

2.7.5 Amazon DAX

Amazon DynamoDB Accelerator ehk DAX on täielikult hallatud, kiirelt kättesaadav vahemälu süsteem Amazon DynamoDB andmebaasi jaoks. DAX lubab kuni 10 kordset jõudluse suurendamist isegi miljonite päringutega sekundis. DAX süsteem töötab sama API lahenduse peal nagu DynamoDB, mis tähendab et DAX süsteemi lisamine ja eemaldamine on kerge. Tootja poolelt on lubatud mikrosekundi kiirused vastused [15].

Amazon DAX hinna leiab DynamoDB lehelt ning jaguneb samamoodi vajaduspõhiseks ning eelvarustatud mudeliks. DAX puhul tuleneb hind serveri kasti võimekuse valikust. Andmed võetud AWS regioonist Euroopa (Iirimaa). Tabelis 5 on välja toodud AWS DAX teenuse kulud. Autor eemaldas tabelist kõige suurema võimekusega valikud, kuna need ei ole antud töös olulised [14].

Tabel 5. AWS DAX hinnatabel.

Tüüp	vCPU	Mälu (GiB)	Hind ($ tunnis)
t3.small	2	2	0,042
t3.medium	2	4	0,086
t2.small	1	2	0,042
t2.medium	2	4	0,086
r5.large	2	16	0,284
r5.xlarge	4	32	0,566
r5.2xlarge	8	64	1,134
r5.4xlarge	16	128	2,267
r4.large	2	15,25	0,300
r4.xlarge	4	30,5	0,598
r4.2xlarge	8	61	1,197
r4.4xlarge	16	122	2,393

2.7.6 Amazon Network Load Balancer

Network Load Balancer (lühendatult NLB) ehk võrgukoormusjaotur on süsteem, mis töötab kui kasutaja side keskpunkt. Koormusjaotur jaotab sissetuleva võrguliikluse üle mitme sihtmärgi, nagu näiteks Amazon EC2 serverite, konteinerite ja IP aadresside. Registreeritud sihtmärkide peal viib koormusjaotud läbi tervisekontrolli seiret, ning edastab võrguliiklust ainult tervetele ehk töökorras sihtmärkidele [16].

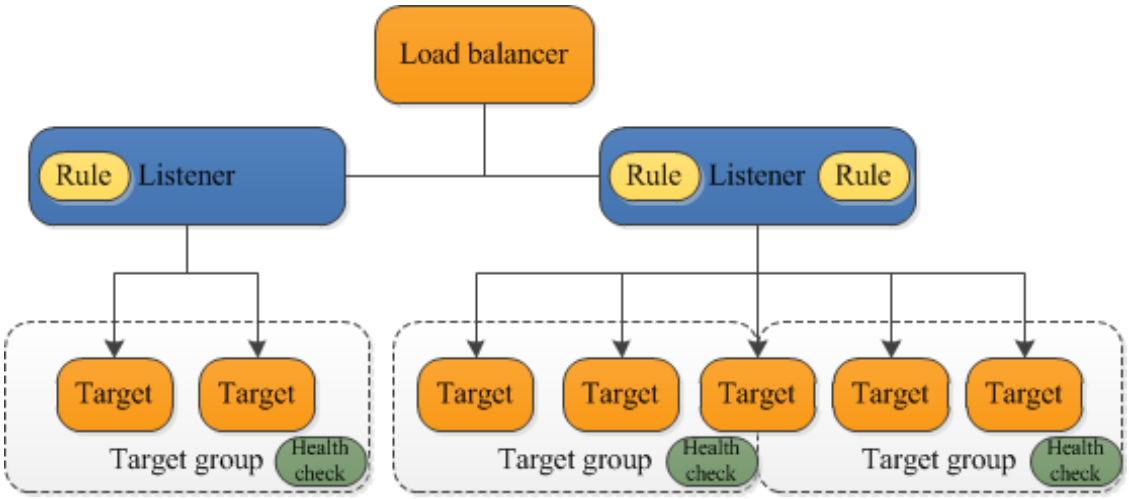
Võrgukoormusjaotur koosneb kahest komponendist. Esimeseks komponendiks on Listener ehk Kuulaja, mis kuulab klientide poolt tulevaid võrguühendusi, vastavalt omaniku seatud portide ja protokollide konfiguratsioonile ning edastab need ühendused Sihtgrupile. Teiseks komponendiks on Target Group ehk Sihtgrupp. Sihtgrupp suunab kliendi poolt tulnud võrguühendused edasi ühele või mitmele registreeritud sihtmärgile. Sihtgrupp on osa koormusjaoturist, mis viib läbi eelnevalt mainitud tervisekontrolli seiret [16].

Network Load Balancer toimib OSI mudeli neljandas kihis. Ta on võimeline edastama miljoneid päringuid sekundis. Kui koormusjaotur saab päringu, valib ta sihtgrupistelet

välja ühe sihtmärgi ning proovib avada TCP ühenduse vastavalt eelnevalt konfigureeritud portile [16].

TCP liikluse edastamiseks valib koormusjaotur sihtmärgi kasutades algoritmi, mis võtab arvesse protokolli, allika IP aadressi, allika porti, siht IP aadressi ning siht porti [16].

Joonisel 4 on välja toodud koormusjaoturi osad ja nende omavaheline suhtlus [17]. Load Balancer ehk koormusjaotur suhtleb Listener'idega ehk kuulajatega. Kuulajatega on seotud Rule'id ehk reeglid, mis vastutavad päringute edasijuhtimise eest. Kuulajad edastavad päringuid Target'ile ehk sihtmärgile. Sihtmärgid on antud töös grupeeritud konteinerid või EC2 serverid.



Joonis 4. Koormusjaoturi komponentide suhtlus.

Võrgukoormusjaoturi kasutamise hind arvutatakse kasutustundide, uute ühenduste arvu ja aktiivsuse aja, baitide koguse ja reeglite kasutamise järgi. Antud töös toob autor välja ainult kasutustundide hinna, kuna üle jäänud tegureid ei ole võimalik täpselt määrata. Üks tund AWS koormusjaoturi kasutust maksab 0,0252 $, see hind kehtib nii rakenduskoormusjaoturi kui võrgukoormusjaoturi kohta [18].

2.6.5 Amazon Application Load Balancer

Application Load Balancer (lühendatult ALB) ehk rakendus koormusjaotur töötab 7. OSI kihi peal, ehk rakenduskihil. Oma ülesehituselt on rakendus koormusjaotur väga sarnane võrgu koormusjaoturiga, kuid tal on mõned olulised erinevused. Võrgu koormusjaotur lihtsalt edastab päringuid. Rakendus koormusjaotur analüüsib päringu sisu ja päiseid, et otsustada, kuhu päringut suunata [17].

Rakendus koormusjaotur võimaldab erinevalt võrgu koormusjaoturist päringute ümbersuunamist, fikseeritud vastuseid ning HTTP päise põhist suunamist. Kuid ei toeta staatilist IP aadressi [17].

2.7 Andmebaasi tüübid

Töös kavandatav süsteem vajab kliendi piirangute salvestamiseks andmebaasi. Seetõttu toob autor käesolevas peatükis välja kaks peamist andmebaasi tüüpi, et neid analüüsi osas omavahel võrrelda.

2.7.1 Relatsiooniline andmebaas

Relatsiooniline andmebaas on andmebaasi tüüp, mis salvestab ja pakub ligipääsu andmetele, mis on üksteisega seotud. Relatsioonilised andmebaasid põhinevad relatsioonilisel mudelil, ehk andmed on esitatud tabeli kujul. Iga tabeli rida on kirje, millel on unikaalne ID ehk võti. Tabeli veerud hoiavad kirjete atribuutide väärtust ning igal kirjel on tavaliselt olemas väärtused iga veeru jaoks. Selline süsteem lihtsustab andmete vaheliste seoste loomist [19].

Relatsiooniline mudel on väga hea andmete järjepidevuse hoidmisel, ehk andmebaas tagastab alati kõige värskemalt muudetud andmeid. Mudeliga käivad kaasas lukustamise süsteemid, mis aitavad ära hoida andmete muutmisega kaasnevaid konflikte [19].

2.7.2 Mitterelatsiooniline andmebaas

Mitterelatsioonilised andmebaasid (NoSQL) on kiire jõudlusega, kergelt kasutatavad, skaleeritavad ja vastupidavad andmebaasid. Tabelite asemel koosneb andmebaas struktuurita või poolstruktureeritud andmetest, mis on tihti JSON dokumentide või võti-väärtus paaride kujul. Mitterelatsiooniline mudel tagab kasutajale lõpuks õigeid andmeid, kuid ei garanteeri, et süsteem tagastaks õigeid andmeid kohe peale muudatust [20].

Mitterelatsiooniline mudel hõlmab endas mitmeid erinevaid alam mudeleid, mis ei põhine traditsioonilisel relatsioonilise andmebaasi mudelil [20].

2.8 Mälusisene andmevõrk

Selles peatükis toob autor välja mälusiseste andmevõrkude definitsiooni ning levinumate toodete kirjelduse. Mälusisesed andmevõrgud on töö teema juures aktuaalsed, kuna töös kasutatav algoritm vajab andmete hoidmiseks andmevõrku.

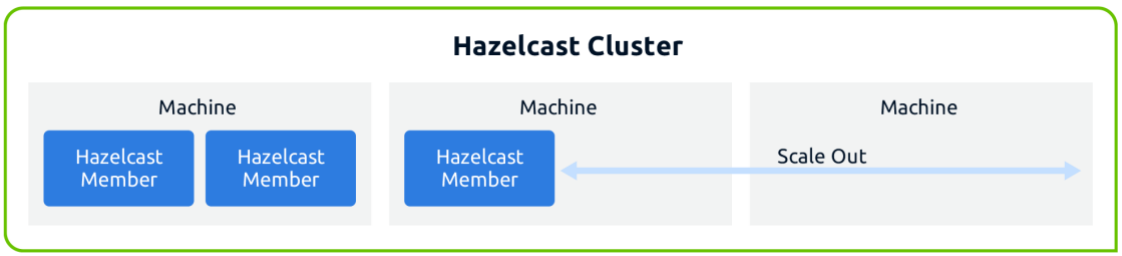
Mälusisene andmevõrk ehk inglise keeles In-Memory Data Grid (lühendatult IMDG) on serverite klaster, mis jagavad üksteisega oma muutmälu ehk RAM-i. Muutmälu jagamine võimaldab andmevõrguga ühendatud rakendustel kasutada samu andmeid. Andmevõrk on mõeldud suurel skaalal süsteemide jaoks, mis vajavad rohkem muutmälu kui on tavaliselt ühele serverile saadaval [21].

IMDG andmevõrk töötab spetsialiseeritud tarkvaraga, mis koordineerib ligipääsu andmetele üle klastri. Iga server andmevõrgu klastris vastutab enda muutmälus asuvate andmete ja vaadete eest ise, kuid kõikidel teistel andmevõrgu serveritel on samuti ligipääs üksteise andmevaadetele. Selline lahendus peidab kasutaja eest andmete ühtlasena hoidmise ja pärimise keerukuse [21].

2.8.1 Hazelcast

Hazelcast on mälusisene andmevõrgu platvorm Java programmeerimis keele jaoks. Hazelcasti arhitektuur võimaldab kõrget mastaapi ja andmete hajutamist klaster keskkonnas. Mastaabi muutusteks on võimalik lisada klastrisse uusi Hazelcast servereid, mis ühenduvad automaatselt teiste serveritega. Andmete tasakaalustamine serverite vahel toimub automaatselt, seda on kujutatud Joonisel 5 [22].

Hazelcast on võimeline töötama igas keskkonnas, mis toetab Java virtuaalmasinaid ning võimaldab kasutada JCache standardile vastavaid integratsioone [22], [23].



Joonis 5. Klastrisse serverite lisamine.

2.8.2 Apache Ignite

Apache Ignite on suurel koormusel arvutamiseks mõeldud hajutatud andmebaas, mille põhilised kasutusalad on hajutatud võti-väärtus süsteemid, ANSI SQL päringud, ACID tehingud ja masinõppe mudelid. Andmeid on võimalik hoida mälus ja kettal ning lisaks on võimalik valida kaks metoodikat andmete püsivuseks. Püsivust saab hoida välise andmebaasi kaudu või Ignite süsteemi enda andmete püsivuse lahenduses [24].

Apache Ignite pakub toetust ka JCache spetsifikatsioonile.

2.8.3 Oracle Coherence Community Edition

Oracle Coherence Community Edition (lühendatult Coherence CE) on Oracle korporatsiooni poolt pakutav vabavara variant Oracle Coherence platvormist [25].

Coherencei kodulehel on toodud välja toote eelised [25].

- Coherence toetab kiiret mastaabi muutust kuni sadade liikmeteni (JVM), millest kõik suudavad andmeid salvestada ning töödelda.
- Andmeid salvestatakse süsteemis alati mitmele liikmele korraga, et tagada tagavara andmete olemasolu avariide korral.
- Iga klastri liige on teadlik, kus teiste liikmete andmed asuvad.
- Võimaldab andmete püsivust kettal. Kommerts vartiant Coherence süsteemist võimaldab lisaks veel andmekeskuse replikatsiooni ja rakenduse tõrkesiiret
- Tuleb kaasa oma Kubernetes operaatoriga, mis võimaldab lisada Coherence põhilisi rakendusi igasse Kubernetes klastrisse kasutades 5 realist YAML faili.

3 Analüüs

Eelnevalt kirjeldatud nõuetele vastamiseks tuleb süsteem kavandada mitmest komponendist ja tehnoloogiast. Selles peatükis analüüsib võimalike lahenduste valikut kasutades eelmises peatükis tutvustatud materjale. Analüüsi tulemustele toetudes koostab autor järgmises peatükis kavandi.

3.1 Sõnumite piiramine

Sõnumite piiramine peab toimuma kahe komponendi koostööl. Rakenduse, mis tegeleb piiramise algoritmi ja rakendamisega, ning andmebaasi, mis salvestab piirangute väärtusi. Käesolevas peatükis analüüsib autor rakenduse jaoks kasutatavat algoritmi ning piirangute jaoks kasutatavat andmebaasi.

3.1.1 Algoritmi valik

Autor toob välja eelmises peatükis tutvustatud kolme liiklusmäära piiramise algoritmi eelised ja puudused [26].

Esimese algoritmina tuuakse välja lekkiv ämber algoritm.

Eelised

- Eemaldab järsud tempo muutused tänu piiratud väljavoolule
- Konstantne määr on seatud kasutades väljavoolu võime reguleerimist

Puudused

- Ülevoolavad sõnumid kaovad kui ämber saab täis
- Ämber võib täituda vanemate sõnumitega ning uued sõnumid voolavad üle

Teise algoritmina tuuakse välja žetoon ämbris algoritm [26].

Eelised

- Lubab piiratud ulatuses järske tempo muutuseid

- Maksimaalne ülemmäär on seatud kindla perioodi peale ning algoritm seda ületada ei luba
- Žetoonide hoidmine ei ole kulukas, kuna salvestatakse ainult ühte numbrit kliendi kohta

Puudused

- Žetoonide koguse haldamine on tehniliselt keeruline, kuna žetoone on vaja pidevalt juurde lisada

Kolmanda algoritmina tuuakse välja libisev aken algoritm [4], [27].

Eelised

- Maksimaalne ülemmäär on seatud kindla perioodi peale ning algoritm seda ületada ei luba
- Mälukasutus on minimaalne, salvestada on vaja ainult kahte numbrit kliendi kohta

Puudused

- Algoritmil on oht olla ebatäpne suure koormuse all

Nõue süsteemile on, et süsteem peab võimaldama sõnumite elustamist ja kordust. Lekkiv ämber algoritm ei vasta sellele nõudele, kuna algoritmi järgi eemaldatakse liigsed sõnumid.

Žetoon Ämbris algoritm ja Libisev Aken algoritm on võrdluses väga sarnaste eeliste ja puudustega. Seetõttu tuleb süsteemi kavandades lähtuda sellest, et kumba neist kahest süsteemi kergem integreerida on.

Žetoon Ämbris algoritmi jaoks on olemas hästi välja arendatud raamistik nimega Bucket4j, mis võimaldab kasutada algoritmi haldamiseks vahemälu. Kuna Libiseva Akna algoritmi jaoks ei eksisteeri nii tuntud ja täisväärtusliku tööriista, siis valiti antud töös kasutatavaks algoritmiks Žetoon Ämbris algoritm.

3.1.2 Piirangute salvestamine

Piirangute salvestamiseks ja muutmiseks on vaja andmebaasi, kus piiranguid hoida. Süsteemsed nõuded näevad ette, et piirangud on salvestatud kui lihtsad võti-väärtus

paarid. Võti-väärtus paarid on väga lihtne andmestruktuur, mille jaoks traditsiooniline relatsiooniline andmebaas võib olla liiga kulukas.

Klientide piirangud on sisemised andmed, mis ei ole ühegi seadusega kaitstud informatsioon, ehk andmete uuenemisega kaasnevad hilinemised on süsteemis lubatud.

Andmete pärimiseks on tarvis väga kiiret andmebaasi süsteemi, kuna iga sissetuleva sõnumi kohta tehakse vähemalt üks päring. Sõnumi elustamise puhul tuleb teha lisapäringuid.

Päringud andmebaasile on oma olemuselt väga lihtsad, alati päritakse ühe kliendi piirangut korraga. Süsteem ei vaja võimalust andmeid korreleerida ja sorteerida.

Nendele nõuetele vastab mitte-relatsiooniline andmebaasi mudel.

3.2 Sõnumite vastuvõtmine

Süsteem peab olema võimeline vastu võtma suure tempo variatsiooniga sõnumite liiklust. Variatsioon ei tohi aga mõjutada sõnumite töötlemise kiirust ja võimekust. Kasutades HTTP päringuid, et sõnumeid algoritmi haldavale rakendusele edastada, on risk rakenduse üle koormamisele, sest rakendus on ühenduses mitme teise allavoolu süsteemiga. Otsene suhtlus klientide ja algoritmi haldava rakenduse vahel tähendaks ka nende süsteemide omavahelist tugevat sidumist. Süsteemi üheks funktsionaalseks nõudeks on aga omavahel lahti ühendatud suhtlus. Lahendus selle probleemile on hoida sõnumeid ajutises puhvris, kus algoritmi haldajal on võimalus neid omas tempos pärida [28]. Klientidel on võimalus edastada nii palju sõnumeid puhvrisse kui nad soovivad, teades et nende sõnumeid edastatakse ainult lepingus sätestatud piirangute järgi.

Sõnumite edastamine puhvrisse käib tavaliselt läbi granulaarse API, mis ei ole klientidele vajalik. Näiteks RabbitMQ API juhend, mis võimaldab suures koguses metoodikaid sõnumite edastamiseks [29]. On veel oht et puhvri süsteemi välja vahetades pole võimalik klientidele enam samasugust API lahendust pakkuda. Selle probleemi lahenduseks on API väravrakendus [30].

Väravrakenduse eelised

- Väravrakendus võimaldaks peita klientide eest rakenduse keerukust, kuna lubaks klientidel suhelda ainult autori poolt defineeritud lõppsõlmedega
- Väravrakendus võimaldaks tagastada klientide päringutele informatiivseid vastuseid läbi autori poolt kirjeldatud HTTP vastustega

Väravrakenduse miinused

- Lisa kulu tänu lisakihile, mis tulevad serveri ülalpidamise ning loomise kuludest

Süsteemile ei ole pandud otseseid kulukuse piiranguid. Autor lähtus analüüsis mõistlikkuse piirides püsimisest ning väravrakendusega kaasnev lisakulu ei tundunud kaaluvat üle väravrakenduse kasutamise eelised.

3.3 Mikroteenuste suhtlus

Mikroteenuste suhtluse peatükis käsitleb autor peamiselt kahte suhtlusarhitektuuri: REST API ja sõnumside edastus. Peatüki eesmärk on leida parim lahendus kõikide süsteemis asuvate mikroteenuste suhtluse vahendamiseks.

3.3.1 REST API lahendus

REST kui tarkvaraarhitektuuri laad on väga hea lahendus klient-server arhitektuurile, kus on kindel vajadus vastata päringutele ning pakkuda avalike API lõppsõlmi. Ettevõtte siseste mikroteenuste suhtluseks ei pruugi see samas kõige parem lahendus olla. Järgmises loetelus toob autor välja REST API lahenduse eelised ja puudused [31].

Eelised

- Päringud ja vastused on omavahel sünkroonis
- Sobib ühendamiseks iga programmeerimis keelega, kuna on väga laialdaselt levinud

Puudused

- Kahe teenuse vaheline tugev sõltuvus. Mõlemad teenused peavad olema sarnases võimekuse mastaabis
- Tänu sünkroonsele suhtluse on serveri lõimedel oht suluseisule
- Veakäitlus on REST API puhul tihti keeruline, kuna ei ole võimalik jätta saatjat igavesti pärinute edastamist uuesti proovima

3.3.2 Sõnumside edastus

REST API paljude puuduste ületamise lahenduseks on sõnumside suhtlus, kus mikroteenuste poolt välja saadetud sõnumite eesmärk on käivitada teises mikroteenuses sündmusi. Sündmuste edastamine on oma olemuselt asünkroonne. Järgmises loetelus toob autor välja sõnumside edastuse eelised ja puudused [31].

Sõnumside edastuse eelised.

- Asünkroonsed sõnumite edastamised ehk Sõnumite edastamisel ei teki vajadust oodata vastust
- Suhtlevad teenused ei tea üksteisest mitte midagi
- Suurem vastupidavus avariide korral, kuna teenused ei saa üksteist otseselt survestada

Sõnumside edastuse puudused.

- Suurem kulu tänu lisakihile. Väiksemate ja lihtsamate teenuste korral piisab REST suhtlusest
- Raskem vigu leida. Sõnumiside süsteemides toimunud vigu on keerukam otsida, kuna sõnumiside süsteemid on oma olemuselt suure peidetud keerukusega

Sõnumside edastusega kaasneb probleem kui allavoolu teenused ei suuda piisava tempoga sõnumeid sõnumibussist lugeda. Seetõttu on vajalik, et sõnumibussist sõnumeid lugevad teenused hülgaks liiga pikalt seisnud sõnumid. Hüljatud sõnumid on vaja kokku korjata ning uuesti süsteemi töötlemisele saata.

Sõnumside edastuse tingimustele vastab Amazon SQS sõnumibuss. SQS võimaldab vaadata sõnumite algset edastusaega. Lisaks võimaldab SQS lisada sõnumeid bussi hilinenud nähtavusega, ehk Piiraja süsteem võib piirangu ületanud kliendi sõnumeid

siiski edastada, kuid neid on võimalik pärida alles hiljem. SQS pakub lisaks veel hüljatud sõnumite haldamise süsteemi, kust neid on võimalik eraldi pärida ning elustada.

3.3.3 Analüüsi tulemusel tehtud lahenduse valik

Järelduseks on, et käesoleva süsteemi ehitamiseks sobib paremini sõnumiside edastus metoodika, kuna sõnumiside eelised haakuvad paremini süsteemsete nõuetega. Näiteks iseseisvas klastris töötamine: sõnumiside võimaldab tööahelas eespool asuvatel teenustel sõnumeid sõnumibussist lugeda, kuid ei ole kohustatud seda koguaeg tegema. Selline lahti seotud suhtlus võimaldab süsteemil taluda paremini tõrkeid.

3.4 Mälusisese andmevõrgu tehnoloogiate analüüs

Mälusiseseid andmevõrke on turul mitmeid, kuid antud peatükis käsitseb autor ainult kahte neist, mis on välja toodud kui Bucket4j raamistiku poolt toetatud. Need kaks platvormi on Hazelcast ja Apache Ignite. Võrdlusest jäi välja Oracle Coherence, kuna Oracle korporatsioon keelab oma toote jõudlus andmete avaldamist ilma nende loata [5].

Autor kasutas võrdluseks kahte erinevat jõudluse võrdlemis tulemusi. Mõlemad võrdlused toimusid omanik ettevõtte poolt läbiviidult. Testides võrreldi mõlema süsteemi baas vahemälu operatsiooni ning tehinguid, mis vastasid JCache (JSR 107) standardile.

Artiklites võrreldi andmebaasi tehinguid ehk *transactions*[1] ja andmebaasi päringute kiirust. Kõikides tulemustes tähendab vasakpoolne graaf andmebaasi operatsioonide arvu sekundis, ning mida suurem see on seda parem tulemus. Parempoolne graaf annab informatsiooni operatsioonide latentsuse kohta, ehk mida väiksem number seda parem.

[1] *Transaction* – Grupp andmebaasi operatsioone, millel on järgnevad omadused: jagamatus, ühtlus, isoleeritus ning vastupidavus [37].

3.4.1 Apache Ignite jõudlus

GridGain ettevõte avaldas oma kodulehel artikli nimega Benchmarking Data Grids: Apache Ignite vs Hazelcast, Part I, milles tõi artikli autor välja kogu jõudluse võrdlemise metoodika ning tulemused [32].

Tabelis 6 on välja toodud keskmine operatsioonide ja transaktsioonide arv sekundis. Tabelis 7 on välja toodud operatsioonide ja transaktsioonide keskmine latentsus, mida mõõdetakse nanosekundites. Tabelis kasutatud andmed on võetud Lisas 2 asuvatest graafikutest [32].

Kasutades elementaaroperatsioone nagu PUT ja GET leiti testi tulemusel, et Ignite ja Hazelcast on mõlemad sarnase jõudlusega, kuid Ignite on keskmiselt 4-7% kiirem.

Testid viidi läbi veel ka elementaartehingute operatsioonidega. Tulemuseks oli, et Ignite on 35-45% kiirem kui Hazelcast.

Tabel 6. Hazelcast ja Apache Ignite operatsioonide kiiruse võrdlus.

	Apache Ignite (operatsioone / sekundis)	**Hazelcast (operatsioone / sekundis)**
PUT- operatsioonide keskmine arv sekundis	113449,50	105591,52
PUT-GET- operatsioonide keskmine arv sekundis	76768,33	74066,85
PUT- transaktsioonide keskmine arv sekundis	44588,98	25958,32
PUT-GET-transaktsioonide keskmine arv sekundis	34430,12	22185,87

Tabel 7. Hazelcast ja Apache Ignite keskmiste latensuste võrdlus.

	Apache Ignite (latentsus, nanosekundites)	**Hazelcast (latentsus, nanosekundites)**
PUT- operatsioonide keskmine latentsus	563511,91	606203,54
PUT-GET- operatsioonide keskmine latentsus	832845,97	863920,07
PUT- transaktsioonide keskmine latentsus	1435,865	2466361,42
PUT-GET- transaktsioonide keskmine latentsus	1859938,39	2885730,49

3.4.2 Hazelcast jõudlus

Hazelcast ettevõte avaldas ka oma kodulehel jõudlus võrdluse artikli, kus võrreldi sarnaseid tööriistu kasutades taaskord Hazelcast ja Apache Ignite. Tulemused oli seekord vastupidised GridGain esitletud tulemustele [33].

Tabelis 8 on sarnaselt eelmise peatükiga toodud välja keskmine operatsioonide ja transaktsioonide arv sekundis. Tabelis 9 on välja toodud operatsioonide ja transaktsioonide keskmine latentsus, mida mõõdetakse nanosekundites. Tabelis kasutatud andmed on võetud Lisas 3 asuvatest graafikutest

PUT ja PUT-GET operatsioonide tulemustest on näha, et Hazelcast on võimeline tegema rohkem operatsioone sekundis ning on lühema latentsusega kui Ignite.

Testid viidi läbi veel ka elementaartehingute operatsioonidega. Mõlema testi tulemused viitavad Hazelcasti madalamale latentsusele ja kiiremale töövõimele. Tabelis kasutatud andmed on võetud Lisas 3 välja toodud graafikutest [33].

Tabel 8. Hazelcast ja Apache Ignite operatsioonide kiiruse võrdlus.

	Apache Ignite (operatsioone / sekundis)	**Hazelcast (operatsioone / sekundis)**
PUT-operatsioonide keskmine arv sekundis	155757,25	179568,33
PUT-GET- operatsioonide keskmine arv sekundis	99411,22	111997,38
PUT-transaktsioonide keskmine arv sekundis	30384,77	91486,22
PUT-GET- transaktsioonide keskmine arv sekundis	39210,62	62346,08

Tabel 9. Hazelcast ja Apache Ignite keskmiste latensuste võrdlus.

	Apache Ignite (latentsus, nanosekundites)	**Hazelcast (latentsus, nanosekundites)**
PUT- operatsioonide keskmine latentsus	1644003,01	1425996,59
PUT-GET- operatsioonide keskmine latentsus	2575613,67	2288413,98
PUT- transaktsioonide keskmine latentsus	8435672,27	2800542,28
PUT-GET- transaktsioonide keskmine latentsus	6542158,98	3987890,45

Lisaks avaldas Hazelcast artikli otsese vastusena GridGain artiklile, mis väidab, et GridGaini poolt avaldatud artikkel on tehtud pahatahtlikult valesti konfigureeritud sätetega. Artikli autor toob välja esialgse artikli veakohad ning jooksutab testi uuesti kasutades parandatud konfiguratsiooni. Testi tulemuseks on Hazelcast süsteemi oluliselt suurem jõudlus [34].

3.4.3 Mälusisese andmevõrgu valik

Eelnevates peatükkides väljatoodud artiklid on mõlemad huvide konfliktis, kuna autorid töötavad mõlemal juhul parema jõudlustulemusega toote ettevõttes. Hazelcasti artiklid siiski adresseerivad otse ka konkurendi tulemusi, ning toovad välja nende testide murekohti. Seetõttu otsustas autor valida tööks Hazelcast süsteemi.

4 Kavand

Kavandis kirjeldatakse süsteemi realiseerimiseks vajalike komponentide eesmärgid, omavahelised suhtlusmetoodikad ja süsteemi loogika.

Klient edastab süsteemile sõne kujul sõnumi sisu ning kliendi identifikaatori, kasutades HTTP päringut.

Sõnumite vastuvõtmise analüüsi tulemusel selgus, et kõige parem viis kuidas sõnumeid klientidelt süsteemi vastu võtta on väravrakendust kasutades. Väravrakendust kasutades on võimalik luua väga piiratud API, mis peidab klientide eest süsteemi keerulise loogika. Lisaks võimaldab väravrakendus kategoriseerida sõnumeid.

Klientide ebastabiilsete käitumismustrite haldamise jaoks leidis autor analüüsi tulemusel, et on tarvis hoida sõnumeid peale vastuvõtmist sõnumisiinis. Siin tuleb seada üles Amazon Simple Queue Service teenust kasutades. Selle sõnumisiini eesti keelseks nimeks saab Sissetulev Sõnumsiin. Sõnumite avalikustamine allavoolu teenustele toimub sarnaselt nende vastuvõtmisele, kasutades sõnumisiini. Selle sõnumibussi eesti keelseks nimeks saab Väljuv Sõnumibuss.

Piirangute hoiustamise jaoks leiti analüüsi tulemusel, et on vaja kiiret NoSQL andmebaasi, mille tingimustele vastab kõige paremini Amazon DynamoDB.

Analüüsi tulemusel selgus, et nii Libisev Aken algoritm kui ka Zetoon Ämbris algoritm on mõlemad süsteemi nõuetele vastavad, kuid Žetoon Ämbris algoritmi jaoks on olemas tööriist nimega Bucket4j, mis haakub autori tehniliste oskustega. Algoritmi jaoks on vaja ehitada mikroteenus, mis on võimeline pärima sõnumeid Amazon SQS sõnumibussist, pärima piiranguid Amazon DynamoDB andmebaasist, kalkuleerima piiranguid vastavalt sõnumi kategooriale kasutades mälupõhist andmevõrku ning edastama sõnumeid Amazon SQS sõnumibussi.

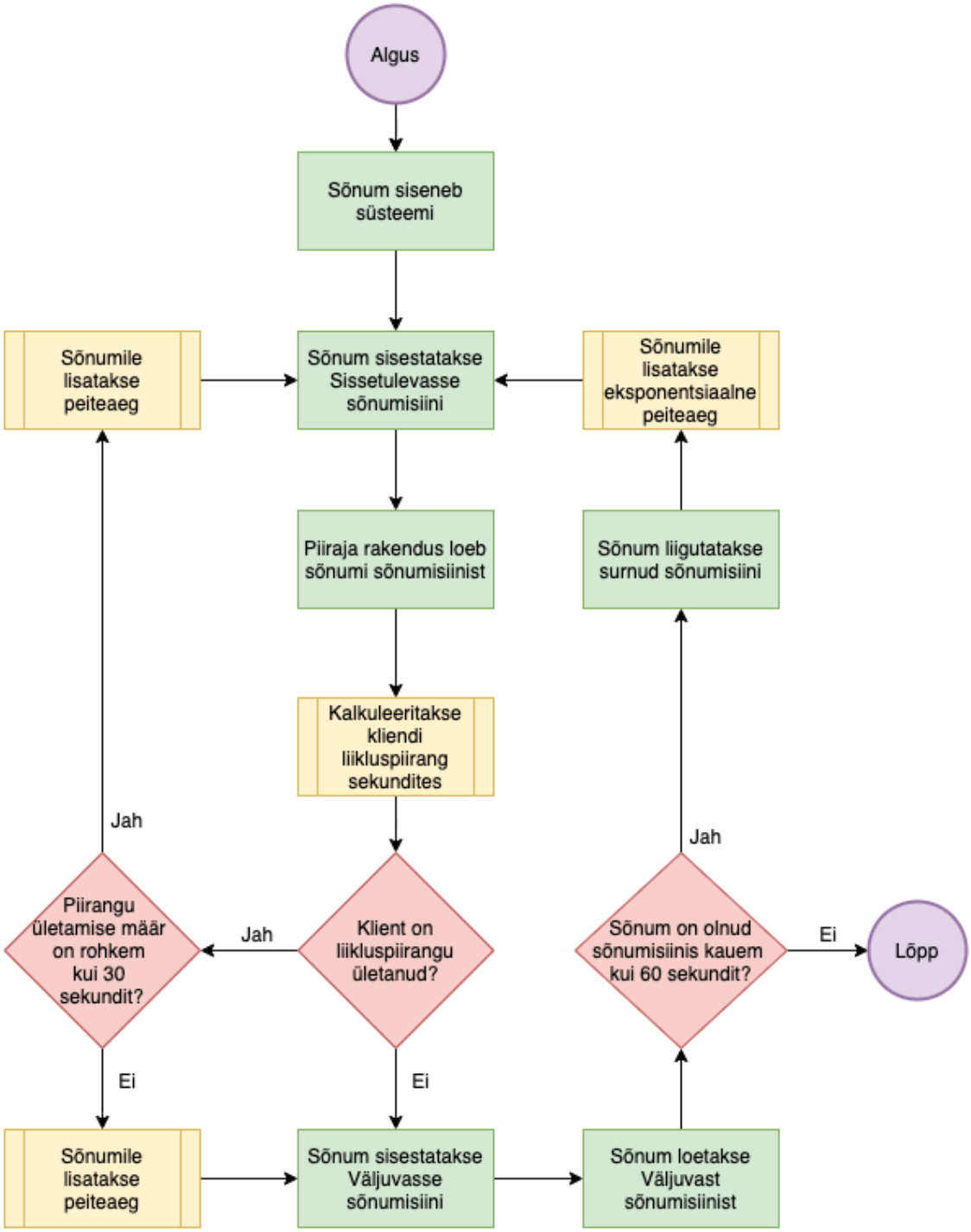
Bucket4j algoritm vajab klastris töötamiseks mälusisest andmevõrku, millega on võimalik ühenduda JCache API kaudu. Sobiv mälusisene andmevõrk valiti läbi mitme

avalikult kättesaadava võrdlustulemuste analüüsi. Valitud andmevõrguks sai Hazelcast tarkvara.

Kasutades ära SQS süsteemi võimet lisada sõnumeid sõnumibussi hilinemisega, ning kombineerides seda Bucket4j võimekusega kalkuleerida liiklusmäära võlga, on võimalik lisada sõnumeid Väljuvasse Sõnumibussi algoritmi tagastatud liiklusmäära võlaga. Kuna äripoolsete nõuete sees on kirjas, et limiitide muutmisel on vastuvõetav mõnekümne sekundiline hilinemine, tuleb lisada väljuvasse sõnumibussi ainult kindlas ajapiiris olevad sõnumid. Autor valis käesoleva töö jaoks selleks ajaks 30 sekundit. Ülejäänud sõnumid, mille puhul võlg on suurem kui 30 sekundit tuleb sisestada hilinemisega tagasi Sissetulevasse Sõnumibussi kasutades algoritmi poolt kalkuleeritud võlga.

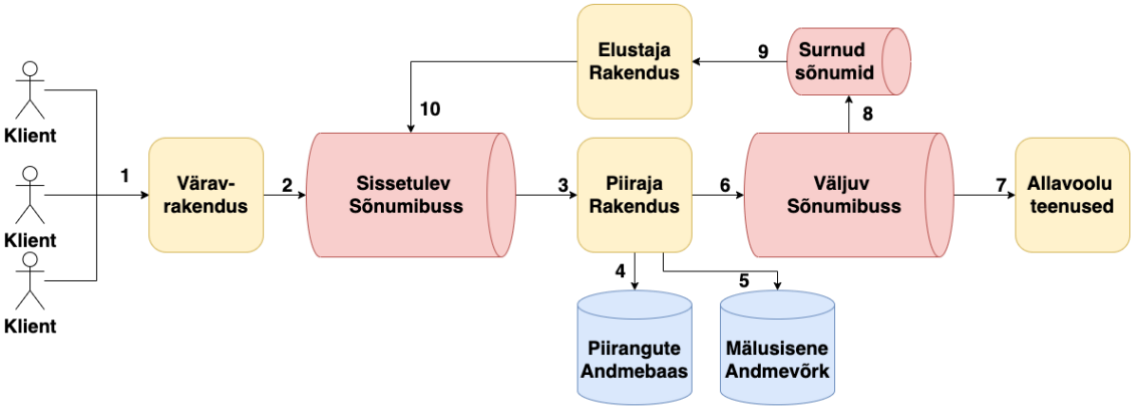
Selleks, et kaitsta Väljuvat Sõnumibussi liiga suure hulga kogunenud sõnumite eest, peavad allavoolu teenused hülgama sõnumid, mis on sõnumibussis olnud kauem kui 60 sekundit. Hüljatud sõnumid liiguvad SQS surnuaeda, kust Sõnumite Elustaja teenus neid pärib. Elustaja Teenuse eesmärk on võtta hüljatud sõnumeid Väljuvast Sõnumibussist, ning liigutada need sõnumid tagasi Sissetulevasse Sõnumibussi kasutades eksponentsiaalset hilinemist. Sõnumitel on võimalik sattuda SQS surnuaeda mitmeid kordi ning seetõttu peab iga sõnumiga kaasas olema kordaja, mis näitaks mitu korda on sõnumit juba elustatud. Elustamiskordaja järgi otsustab teenus, kui pika hilinemisega sõnum Sissetulevasse Sõnumibussi liigutatakse.

Süsteemi loogika on kujutatud visuaalselt järgmisel diagrammil.



Joonis 6. Süsteemi loogika diagramm.

Joonisel 7 on kujutatud süsteemi komponentide seosed. Lisaks on nummerdatud sõnumite liikumise trajektoor ja kirjeldatud komponentide vaheline suhtlus.



Joonis 7. Süsteemi komponentide kavand.

Järgnevas loetelus on kirjeldatud Joonisel 7 nummerdatud komponentide seosed.

1. Klient edastab sõnumi HTTP päringu kujul Väravrakendusele
2. Väravrakendus sisestab sõnumi Sissetulevasse Sõnumisiini
3. Piiraja Rakendus pärib regulaarselt Sissetulevast Sõnumisiinist sõnumeid
4. Piiraja Rakendus pärib sõnumi saatja piiranguid Piirangute Andmebaasist
5. Piiraja Rakendus kalkuleerib kliendi piirangu kasutades Mälusisest Andmevõrku
6. Piiraja Rakendus sisestab sõnumi Väjuvasse Sõnumisiini
7. Allavoolu teenused saavad sõnumit Väljuvast Sõnumisiinist pärida
8. Hüljatud sõnumid liiguvad Surnud Sõnumisiini
9. Elustaja Rakendus pärib regulaarselt Surnud Sõnumisiinist sõnumeid
10. Elustaja Rakendus sisestab sõnumid tagasi Sissetulevasse Sõnumisiini

5 Rakendus

Kavandi elluviimiseks oli vajalik rakendus kõigepealt lokaalselt realiseerida ning ette valmistada Docker konteinerid. Realiseerimisele järgnes süsteemi üles seadmine AWS keskkonnas kasutades eelnevalt valmistatud konteinereid.

Autori poolt tehtud tehnoloogia valikute põhjendused:

- Programmide realiseerimiseks kasutab autor Java programmeerimis keelt, kuna Java on autori tööl kasutatav programmeerimis keel.
- Veebirakenduste raamistikuks valis autor Dropwizard tehnoloogia, kuna Dropwizard on autorile veebirakenduste raamistikest kõige paremini tuntud.
- Veebirakenduse ehitamiseks valis autor Maven tarkvara, kuna on autorile kõige paremini tuttav.
- Veebirakenduste konteinerite süsteemiks vali autor Docker tarkvara, kuna Docker on autori tööl kasutatav tarkvara.
- Koodi versioonihalduseks kasutas autor GitHub tarkvara, kuna autor on versioonihaldus platvormidest GitHubiga kõige tuttavam.

5.1 Arenduskäik

Käesolevas peatükis kirjeldab autor süsteemi arenduskäigu lokaalset osa. Tutvustatakse programmeerimisega seotud valikuid ning rakenduste konteineriseerimist.

5.1.1 Väravrakenduse arendus

Väravrakenduse arendamiseks kasutas autor Java programmeerimisekeelt ning Dropwizard raamistiku. Rakenduse lähtekood on saadaval GitHub keskkonnas https://github.com/jakobjaks/queueing-gateway.

Väravrakenduse API kirjeldamiseks kasutati OpenAPI 3.0.1 spetsifikatsiooni, mis võimaldas kiiret testimist ning serveri lõppsõlmede mugavat kasutajatele tutvustamist.

Väravrakendusele lisati tervisekontrolli lõppsõlm. Selline lõppsõlm võimaldab rakenduse kasutajatel regulaarselt kontrollida süsteemi kättesaadavust, ilma et oleks vaja klientide sõnumeid edastada. Eelkõige on selliste tervisekontrollide kasutajateks koormusjaoturite süsteemid ning ülesvoolu kasutajad.

Süsteemi konteineriseerimiseks kirjutati valmis Dockerfile, mis koosnes kõigepealt süsteemi ülesehitamisest Maven tarkvara abil, ning JAR faili loomisest. Seejärel installeeriti Java ja JAR fail koos Docker konteinerisse.

5.1.2 Piiraja rakenduse arendus

Väravrakenduse arendamiseks kasutas autor Java programmeerimisekeelt ning Dropwizard raamistiku. Rakenduse lähtekood on saadaval GitHub keskkonnas https://github.com/jakobjaks/queueing-limiter.

Väravrakendus sõltub oma töös neljast erinevast süsteemist. Seetõttu oli vajalik jagada Väravrakenduse töö ära paljude lõimude vahel, et mitte lasta ühel allavoolu süsteemil seada ohtu terve rakenduse töö. Sõnumibussist lugemiseks määrati 2 lõimu, kuna iga lõim teeb SQS vastu regulaarselt päringuid siis oli mõistlik hoida lõimede arv pigem väike. SQS toote hind sõltub otseselt päringute arvust ning madala liiklusega perioodil võivad liiga paljud tühjad päringud arveid suurendada.

Sõnumibussist saadud sõnumite käsitsemiseks määrati kuni 6 lõimu. Käsitsemiseks ei ole vaja lõimude arvu väiksena hoida, kuna käsitsemisel tehakse päringuid ainult vajaduse põhiselt.

Selleks et tagada süsteemi asünkroonset tööd kirjutati kood kasutades Java CompletableFuture tehnoloogiat, mis võimaldab vabastada süsteemi lõime kui mõni allavoolu teenuse klient ei vasta õigeaegselt.

Piirangute hoiustamiseks kasutati Bucket4j raamistiku. Bucket4j võimaldab ühendust erinevate väliste vahemälu süsteemidega, mis põhinevad JCache API (JSR 107) spetsifikatsioonil [5]. JCache API kasutus ja algoritmi realiseerimine programmikoodina töös välja toodud Lisas 3.

5.1.3 Elustaja rakenduse arendus

Elustaja arendamiseks kasutas autor Java programmeerimisekeelt ning Dropwizard raamistiku. Rakenduse lähtekood on saadaval GitHub keskkonnas https://github.com/jakobjaks/queueing-reviver.

Elustaja rakendus sõltub oma töös kolmest sõnumisiinist. Rakendus pärib kahest sõnumisiinist regulaarsete intervallidega sõnumeid. Kätte saadud sõnumitele kalkuleerib rakendus eksponentsiaalse ooteaja, millega edastatakse sõnum Sissetulevasse sõnumisiini. Elustaja rakendus kasutab tööks asünkroonset SQS HTTP klienti, mille eesmärk on vältida suluseisu programmi lõimudel.

5.1.4 Testrakenduse arendus

Testrakenduse arendamiseks kasutas autor Java programmeerimisekeelt ning Dropwizard raamistiku. Rakenduse lähtekood on saadaval GitHub keskkonnas https://github.com/jakobjaks/queueing-consumer.

Testrakendus käitub süsteemis kui sõnumitöötluses allavoolu asuvad teenused. Testrakenduse funktsioon on pärida Väljuvast Sõnumisiinist sõnumeid nii kiiresti kui võimalik ning edastada seireandmeid. Rakenduse edastatud seireandmetega on võimalik järeldada väljuvate sõnumite tempot.

5.1.5 Hazelcast süsteemi ülesseadmine

Hazelcasti süsteemi üles seadmine lokaalselt viidi ellu kasutades Hazelcasti loojate poolt pakutavad avaliku Docker Image’it [35]. Selleks et kohandada Hazelcasti kasutama Bucket4j Java klasse, oli vaja lisada originaalsele Hazelcasti poolt pakutava Docker Image’ile lisaks veel Bucket4j JAR failid. Seda oli võimalik teha kasutades Dockeri Image laiendamist tööriista.

Järgnev kood on Hazelcasti laiendamiseks koostatud Dockerfile faili sisu.

```
FROM hazelcast/hazelcast:4.2

ENV HZ_LIB $HZ_HOME/lib
WORKDIR $HZ_LIB

ENV HZ_VERSION 4.2
ENV BUCKET4J_VERSION 6.2.0
ENV JCACHE_VERSION 1.1.0
ADD hazelcast.xml $HZ_HOME

ADD https://repo1.maven.org/maven2/javax/cache/cache-
api/$JCACHE_VERSION/cache-api-$JCACHE_VERSION.jar $HZ_HOME
ADD https://repo1.maven.org/maven2/com/github/vladimir-
bukhtoyarov/bucket4j-core/$BUCKET4J_VERSION/bucket4j-core-
$BUCKET4J_VERSION.jar $HZ_HOME
ADD https://repo1.maven.org/maven2/com/github/vladimir-
bukhtoyarov/bucket4j-jcache/$BUCKET4J_VERSION/bucket4j-jcache-
$BUCKET4J_VERSION.jar $HZ_HOME
ADD https://repo1.maven.org/maven2/com/github/vladimir-
bukhtoyarov/bucket4j-hazelcast/$BUCKET4J_VERSION/bucket4j-hazelcast-
$BUCKET4J_VERSION.jar $HZ_HOME
ADD https://repo1.maven.org/maven2/javax/cache/cache-
api/$JCACHE_VERSION/cache-api-$JCACHE_VERSION.jar $HZ_HOME/lib
ADD https://repo1.maven.org/maven2/com/github/vladimir-
bukhtoyarov/bucket4j-core/$BUCKET4J_VERSION/bucket4j-core-
$BUCKET4J_VERSION.jar $HZ_HOME/lib
ADD https://repo1.maven.org/maven2/com/github/vladimir-
bukhtoyarov/bucket4j-jcache/$BUCKET4J_VERSION/bucket4j-jcache-
$BUCKET4J_VERSION.jar $HZ_HOME/lib
ADD https://repo1.maven.org/maven2/com/github/vladimir-
bukhtoyarov/bucket4j-hazelcast/$BUCKET4J_VERSION/bucket4j-hazelcast-
$BUCKET4J_VERSION.jar $HZ_HOME/lib

USER root
RUN chmod -R +r ${HZ_HOME}
ENV JAVA_OPTS -Dhazelcast.config=/opt/hazelcast/hazelcast.xml -
DLOGGING_LEVEL=DEBUG

CMD ["/opt/hazelcast/start-hazelcast.sh"]
EXPOSE 5701
```

Joonis 8. Hazelcast Dockerfaili sisu.

Lisaks oli veel tarvis muuta Hazelcasti vaikimisi sätteid, et võimaldada süsteemil testkeskkonnas töötamist. Lisati AWS süsteemis töötamist võimaldavad võrgusätted ning tervisekontrolli lõime avaldamise sätted. Hazelcaati sätete faili sisu on välja toodud Joonisel 9.

```
<network>
    <join>
        <multicast enabled="false">
        </multicast>
        <aws enabled="true">
        </aws>
    </join>
    <rest-api enabled="true">
        <endpoint-group name="HEALTH_CHECK" enabled="true"/>
    </rest-api>
</network>
```

Joonis 9. Hazelcast konfiguratsiooni faili sisu XML formaadis.

5.1.6 Sõnumite andmemudel

Sõnumite andmemudeli koostamisel võeti arvesse nii süsteemseid nõuded kui ka kliendi poolseid nõudeid. Andmemudel Java programmeerimiskeeles on toodud välja Joonisel 10.

- Sõnum koosnes väga primitiivsest sõnest, ehk kliendi poolt edastatud andmetest. Programmis oli välja nimi „content“.
- Sõnumil oli kaasas kliendi identifikaator, indikaator oli kujutatud sõnena. Programmis oli välja nimi „identifier“.
- Sõnumil oli enda unikaalne identifikaator, mis oli UUID[1] formaadis ning kasutas sõne andmetüüpi. Programmis oli välja nimi „id“.
- Sõnumil oli kaasas number, mis näitas mitu korda on sõnum süsteemis hüljatud ning uuesti elustatud. Programmis oli välja nimi „requeueCounter“.

```
private String id;
private String content;
private String identifier;
private int requeueCounter;
```

Joonis 10. Sõnumi andmemudel Java keeles.

[1] https://tools.ietf.org/html/rfc4122

5.1.7 Rakenduste konteineriseerimine

Autori poolt programmeeritud rakendused põhinevad kõik täpselt samade raamistikel, mis võimaldab neid ka sarnaselt konteineriseerida. Rakenduste konteineriseerimiseks kasutati Dockerfile tööriista.

Dockerfile loomine koosnes kahest etapist. Esimese etapi eesmärk oli ehitada valmis rakendus JAR faili kujul, kasutades Maven projektiehitus tööriista.

Teise etapi eesmärk oli ehitada valmis Docker konteiner. Kõigepealt lisatakse konteinerisse Java JDK (Java Development Kit), mis võimaldab JAR failide käivitamist. Seejärel lisatakse konteinerisse eelnevalt valmistatud JAR fail koos konfiguratsiooni failiga. Kõige viimasena kirjeldatakse failis käsk, millega süsteemi käivitada. Joonisel 11 on kirjeldatud Piiraja rakenduse Dockerfile sisu.

```
FROM maven:3.6.0-jdk-11-slim AS build
COPY src /usr/src/app/src
COPY pom.xml /usr/src/app
RUN mvn -f /usr/src/app/pom.xml clean install

FROM openjdk:11
COPY config.yml /var/queueing-limiter/
COPY --from=build /usr/src/app/target/queue-limiter-1.0.jar
/var/queueing-limiter/
EXPOSE 8082:8083
WORKDIR /var/queueing-limiter
CMD ["java", "-jar", "-Done-jar.silent=true -
Dcom.sun.management.jmxremote.port=9090", "queue-limiter-1.0.jar",
"server", "config.yml"]
```

Joonis 11. Piiraja rakenduse Dockerfile.

5.2 Pilvekeskkond

Rakenduse juurutamiseks pilvekeskkonda kasutati peamiselt kahte laialtlevinud pilveteenust, milleks olid GitHub ja AWS ECS. Need kaks teenust võimaldasid omavahelist integreerimist ning pideva integratsiooni torustiku ülesseadmist. Kogu ehitatud pilvesüsteemi võrgutopoloogia on välja toodud 4. lisas.

5.2.1 GitHub Actions

GitHub Actions pakkus võimalust juurutada Docker konteinereid versioonihaldus keskkonnast automaatselt AWS ECS pilve klastrisse. Selle tööriista kasutamiseks oli

kõigepealt vaja läbi teha AWS ECS klastri üles seadmine ning konfigureerimine, õpetus selleks asus GitHub Actions lehel [36]. AWS pilvekeskkonda juurutamine toimus automaatselt iga kord kui rakendusel avaldati uus versioon.

5.2.2 AWS ECS

Kogu süsteemi haldamiseks ehitati üles AWS ECS klaster nimega QueueCluster. Igale konteinerteenusele ehitati oma ECS Task Definiton ehk konteineri juurutamise seade. Lisaks ehitati igale Task Definition'ile oma ECS Service ehk ühte tüüpi konteinerite kogumi haldamise seade.

Programmeeritud mikroteenused nagu Väravrakendus, Piiraja rakendus ja Elustaja rakendus kasutasid EC2 pilveservereid. EC2 pilveserverid võimaldasid kergelt ligipääsu teenustele ning neid oli lihtne hallata. Ligipääsu oli tihti vaja, kuna teenused olid autori poolt kirjutatud ning autoril oli vaja leida vigu ja neid parandada.

Hazelcasti konteiner seati üles kasutades AWS Fargate pilveservereid. Hazelcasti puhul ei olnud oluline SSH ligipääs serveritele, sest süsteem on tootena müügis ning suure töökindlusega. Lihtsamate logide ligipääsu võimaldab Fargate AWS konsoolis.

5.2.3 AWS Network Load Balancer

Võrgukoormusjaoturi eesmärk oli jaga ära koormus Hazelcasti konteinerite vahel. Koormusjaotur võimaldas algoritmi päringute võrdselt edasi suunamist, ning pakkus Piiraja Rakendusele staatilist IP aadressi, millega suhelda.

5.2.4 AWS Application Load Balancer

Väravrakendus on süsteemi ainuke komponent, mis puutub otseselt kokku välismaailmaga ehk klientidega. On oluline, et kui klastris eksisteerib mitmeid Väravrakenduse konteinereid korraga, siis jaotatakse klientide liiklus nende vahel ära õiglaselt ja võrdselt. Seetõttu võeti kasutusele AWS Application Load Balancer, ehk teenuse koormusjagaja.

5.2.5 Piirangute andmebaas

Piirangute andmebaasiks valitud Amazon DynamoDB üles seadmine toimus AWS veebirakenduse kaudu. Tabeli nimeks valiti „rate_limiter_limits". Peamiseks jaotusvõtmeks valiti sõne nimega „identifier". Peale peamise jaotusvõtme määramist ei

olnud tabelis vaja rohkem andmetüüpe kirjeldada. Ülejäänud andmed ning nende tüübid kirjeldati kasutaja poolt andmebaasi objekti loomise hetkel.

5.2.6 Sõnumisiinid

Sõnumisiinid ehitati AWS veebirakenduses kasutades valdavalt vaikimisi sätteid. Mõlemad sõnumisiinid kasutavad SQS Standard sõnumisiini tüüpi. Mõlemale sõnumisiinile lisati külge veel surnud sõnumite siin, mille eesmärk on püüda kinni ajapiirangu ületanud ning probleemsed sõnumid.

Tabelis 10 on välja toodud mõlema sõnumisiini sätted.

Tabel 10. SQS Sõnumisiinide seaded.

	Sissetulevad sõnumid	**Väljuvad sõnumid**
Sõnumisiini nimi AWS'is	rate_limiter_entry_queue	rate_limiter_exit_queue
Sõnumite eluiga (*retention*)	1 päev	1 päev
Tüüp (*type*)	Standard	Standard

6 Loodud süsteemi testimine ja kuluanalüüs

Käesolevas peatükis viiakse läbi eelmises peatükis realiseeritud süsteemi koormustest, et uurida süsteemi võimekust suurt võrguliiklust reguleerida. Peatüki teises pooles tehakse koormustestiks kasutatud teenuste ja serverite kohta kuluanalüüs, et leida süsteemi teoreetiline ööpäeva töö maksumus.

6.1 Koormustest

Koormustestimine on tarkvarasüsteemi testimise metoodika, mille eesmärk on hinnata süsteemi võimekust tulla toime erinevate koormustega. Koormustesti peamine eesmärk on katsetada rakenduse võimekust eeldatavate päris maailma stsenaariumitega. Lisaks võimaldavad koormustestid kindlaks teha süsteemi võimekuse piire ning koostada sellega SLA lepinguid [37].

Koormustestis edastatakse süsteemile sõnumeid kiiremini, kui kliendile seatud piirang seda lubaks. Koormustesti hüpotees on, et väljuvate sõnumite määr vastab pidevalt kliendile seatud piirangule.

6.1.1 Koormuse genereerimine

Koormuse genereerimiseks kasutati Yandex ettevõtte poolt pakutavat tasuta tarkvara nimega Tank. Tanki üles seadmiseks kasutati kahte konfiguratsiooni faili ning Docker tarkvara.

Esimene fail, nimega load.yaml, on YAML formaadis esitatud konfiguratsiooni sätete kogum Tank koormusgeneraatorile. Failis on kirjeldatud koormuse muster ning aeg, lisaks veel IP aadress, mille vastu koormust genereeritakse, koormuse tüüp ning konsooli kuvamise sätted. Faili sisu on välja toodud Joonisel 12.

```
phantom:
  address: 52.45.194.124:8080
  ammofile: ammo.txt
  ammo_type: uripost
  load_profile:
    load_type: rps
    schedule: line(600, 600, 3m)
console:
  enabled: true # enable console output
telegraf:
  enabled: false
```

Joonis 12. Tank konfiguratsioonifaili load.yaml sisu.

Teine fail, nimega ammo.txt, on koormusgeneraatori loodud päringute sisu defineerimise jaoks. Fail sisaldab päringu päiseid, päringu pikkust baitides ning päringu sisu. Fail on kujutatud Joonisel 13.

```
[Host: 18.208.189.225:8080]
[User-Agent: curl/7.64.1]
[accept: */*]
[Content-Type: application/json]
62 /push-to-queue
{
  "content": "Hello, World",
  "identifier": "load-test3"
}
```

Joonis 13. Tank päringute konfiguratsioonifaili ammo.txt sisu.

6.1.2 Koormuse mõõdikud

Koormuse mõõtmiseks ning seireandmete kogumiseks kasutati Prometheus süsteemi koostöös Grafana seire andmete kuvamise tööriistaga.

Prometheus on avatud lähtekoodiga süsteemi järelevalve tööriist, mis arendati algselt ettevõttes SoundCloud. Antud töös kasutatud Prometheuse lahendus koosneb kahest komponendist. Esimene komponent on Prometheus server, mille eesmärk on kokku koguda seireandmeid. Teine komponent on rakendustes kasutatud teegid, mille eesmärk on seireandmeid avalikustada [38].

Seireandmete kogumiseks kasutati Dropwizard Metrics teeki, mis võimaldab mõõta sündmuste toimumise hulka kindlal ajaperioodil [39]. Seireandmete avalikustamiseks

jaoks kasutati Prometheuse loojate poolt valmistatud teeke, mis võimaldasid integratsiooni Dropwizard Metrics teegiga [40]. Prometheuse teegi kaudu avalikustati eraldi võrguport, mille eesmärk oli lubada Prometheus serverile ligipääs rakenduse poolt kogutud seireandmetele.

Prometheuse üles seadmiseks kasutati Prometheus.yml konfiguratsiooni faili, mille sisu on välja toodud Joonisel 14. Faili lisati kõikide ECS klastri poolt kasutatud AWS EC2 serverite avalikud IP4 aadressid, et võimaldada Prometheusel seireandmeid koguda. Ülejäänud parameetrid failis on vaikimisi väärtused.

```
global:
  scrape_interval:     15s
  evaluation_interval: 15s
rule_files:
  # - "first.rules"
  # - "second.rules"
scrape_configs:
  - job_name: prometheus
    static_configs:
      - targets: ['54.173.111.36:9091', '54.173.111.36:9092',
'54.173.111.36:9093', '18.206.230.226:9091', '18.206.230.226:9092',
'18.206.230.226:9093', '54.146.246.113:9091', '54.146.246.113:9092',
'54.146.246.113:9093', '54.90.230.57:9091', '54.90.230.57:9092',
'54.90.230.57:9093', '52.91.39.24:9091', '52.91.39.24:9092',
'52.91.39.24:9093', '3.80.83.7:9091', '3.80.83.7:9092', '3.80.83.7:9093', ]
```

Joonis 14. Prometheus süsteemi konfiguratsioonifaili sisu YAML formaadis.

Prometheuse käivitamiseks kasutati Prometheus Docker Image'it. Joonisel 15 on kujutatud Prometheuse käivitamiseks kasutatud käsklus. Käsus on sätestatud konfiguratsioonifaili asukoht ja konteinerist avatud võrguportide kaardistus.

```
docker run \
    -p 9090:9090 \
    -v /Users/jroots/thesis/prometheus/prometheus.yml:
/etc/prometheus/prometheus.yml \
    prom/prometheus
```

Joonis 15. Prometheus Dockerfile'i käivitamise käsk.

Grafana käivitamiseks kasutati Grafana Docker Imagei'it. Grafana käivitamiseks kasutatud käsklus on kujutatud Joonisel 16 ning seal on kirjeldatud Grafanaga suhtlemiseks vajalike võrguportide kaardistus.

```
docker run -d -p 3000:3000 grafana/grafana
```

Joonis 16. Grafana Dockerfile'i käivitamise käsk.

6.1.3 Koormustesti tulemused

Koormustestiks üles seadud klastri konfiguratsioon:

- Kasutati 6 EC2 t2.xlarge tüüpi serverit
- Väravrakenduse jaoks kasutati 5 virtuaalset konteinerit
- Piiraja rakenduse jaoks kasutati 4 virtuaalset konteinerit
- Testrakenduse jaoks kasutati 4 virtuaalset konteinerit
- Elustaja rakenduse jaoks kasutati 1 virtuaalset konteinerit
- Andmevõrgu jaoks kasutati 3 ECS Fargate virtuaalset konteinerit
- DAX klastri jaoks kasutati 1 r5.xlarge tüüpi serverit
- Üks rakenduskoormusjaotur
- Üks võrgukoormusjaotur

Koormustesti liiklust genereeriti autori isiklikust arvutist. Enamus saabuva liikluse tempo määrast jäi vahemiku 350 kuni 450 päringut sekundis. Väravrakenduse liiklus on kujutatud graafikuna Lisa 6 peal. Väravrakenduse puhul on märgata stabiilselt kasvavat sõnumite edastus tempot, mis on tingitud ilmselt Yandex Tank päringute genereerimise metoodikast.

Piiraja rakenduse liiklust on kujutatud graafikuna Lisas 5. Piiraja rakenduse liiklus on testi alguses ebaühtlasem kui Väravrakendusel, sest Piiraja rakendus sõltub Väravrakenduse sõnumite edastuskiirusest ning SQS teenuse kättesaadavusest. Enamus liikluse tempo määrast jääb Piiraja rakendusel 350 kuni 450 sõnumi vahele sekundis.

Lisas 7 on kujutatud väljuvate sõnumite liiklust graafikuna. Andmed pärinevad Testrakendusest. Väljuvatel sõnumitel on väga ühtlane 300 sõnumit sekundis tempo määr. Üksikud tempo kõikumised on hiljem tasakaalustatud vastupidiste kõikumistega.

Käesolevas peatükis läbiviidud koormustest tõestas, et lõputöö teoreetilises osas planeeritud ning praktilises osa välja ehitatud süsteem on võimeline sõnumite tempo piiramiseks isegi suurel koormusel.

6.2 Kuluanalüüs

Süsteemi potentsiaalsete kasutajate koormusnõuded võivad olla väga erinevad. Autor kasutab kuluanalüüsiks koormustestis kasutatud koormusmäära ning arvutab nende andmetega süsteemi ööpäeva töökulu.

Koormustestis genereeriti keskmisel 400 sõnumit sekundis, minutis tähendab see 24000 sõnumit, tunnis 1 440 000 sõnumit ja ööpäevas 34 560 000 sõnumit.

Tabelis 11 on välja toodud pilveteenuste loetelu, nende tüüp, kuluühik ja ööpäeva kulu. Teenuste ja nende kuluühikute andmed on välja toodud töö 2. peatükis. Kulud on esindatud dollarites, kuna teenusepakkuja väljastab arveid dollarites.

Tabel 11. Võrgutaristu hinnatabel.

Teenus	**Kogus**	**Tüüp**	**Kuluühik**	**Ööpäeva kulu ($)**
EC2 server	6	t2.xlarge	0,2016 $ / tunnis	29,03
DynamoDB andmebaas	1		0,283 $ / miljoni päringu kohta	0,00
DAX klaster	1	r5.xlarge	0,509 $ / tunnis	12,22
ECS Fargate	3	0.5 vCPU	0,02913 $ / tunnis	2,10
Application Load Balancer	1		0,0252 $ / tunnis	18,40
Network load Balancer	1		0,0252 $ / tunnis	18,40
SQS	2	Standard	0,0000004 $ / sõnumi kohta	27,65
Kokku				107,80

Süsteemi ööpäeva töökulu on ligikaudu 107,80 $.

7 Kokkuvõte

Telekommunikatsioonivaldkonnas on klientide jaoks tähtsal kohal alati kiirus ja kättesaadavus. Liiga pikalt hilinenud või mitte kohale toimetatud meeldetuletussõnum võib tähendada unustatud arstivastuvõtuaega või kaotatud klienti.

Eesmärgiks oli luua süsteemi prototüüp, mis oleks võimeline reguleerima klientide poolt edastatud sõnumite võrguliiklust rakenduskihis vastavalt kindlatele piirangutele. Piirangute määramine klientidele võimaldab kaitsta ettevõtte süsteeme ülekoormuse eest ning tagada klientidele stabiilset teenuse kiirus.

Töö käigus analüüsiti erinevaid võrgusuhtlus mustreid, et leida kõige skaleeruvamad ja sobivad andmeedastus metoodikad. Analüüsiti võrguliikluse reguleerimise algoritme ning nende sobivust, milles võeti arvesse ka realiseerimise lihtsust läbi olemasolevate tööriistade. Lisaks analüüsiti mälusiseste andmevõrkude jõudlusi, et leida reguleerimise algoritmi jaoks parim tarkvara. Analüüside tulemusena koostati süsteemist kavand, kus kirjeldati andmevoogu ja komponentide suhtlust.

Praktilises osas ehitati välja kavandi põhjal süsteemi prototüüp. Kõigepealt kirjutati valmis kolm mikroteenust, mis käitusid süsteemis kui sõnumite töötlejad ning üks mikroteenus testimise jaoks. Seejärel konfigureeriti Hazelcast andmevõrk konteineriseeritud kujul. Kogu süsteem juurutati Amazon Web Services pilvetaristus ning seati üles vajalikud andmebaasid, sõnumisiinid ja koormusjaoturid.

Autor testis süsteemi võimekust piiranguid hoida ja koormust taluda koormustestidega. Süsteem läbis koormustesti edukalt ning klientide sõnumitele rakendati määratud piirangud. Süsteemist väljuv liiklus vastas nõuetele. Koormustestiks kasutatud võrgutaristule tehti ööpäeva töökulu arvestamiseks hinnaanalüüs.

Püstitatud eesmärgid said täidetud ning süsteem võib lähiaastatel prototüübi faasist liikuda päris arendustiimi kätte.

Kasutatud Kirjandus

[1] M. Kaur, „AN OVERVIEW OF QUALITY OF SERVICE COMPUTER NETWORK,“ *Indian Journal of Computer Science and Engineering,* nr 2, 2011.

[2] CISCO, "Quality of Service," 14 10 2019. [Online]. Available: https://www.cisco.com/c/en/us/support/docs/quality-of-service-qos/qos-policing/19645-policevsshape.html. [Accessed 15 3 2021].

[3] A. S. Tanenbaum ja D. J. Wetherall, Computer Networks. 5th Edition., India: Pearson Education Inc, 2011.

[4] Cloudflare ja J. Desgats, „How we built rate limiting capable of scaling to millions of domains,“ [Võrgumaterjal]. Available: https://blog.cloudflare.com/counting-things-a-lot-of-different-things/. [Kasutatud 22 4 2021].

[5] „Bucket4j,“ [Võrgumaterjal]. Available: https://github.com/vladimir-bukhtoyarov/bucket4j. [Kasutatud 26 3 2021].

[6] M. Fowler, „Microservices,“ [Võrgumaterjal]. Available: https://martinfowler.com/articles/microservices.html#:~:text=In%20short%2C%20the%20microservice%20architectural,often%20an%20HTTP%20resource%20API.. [Kasutatud 22 3 2021].

[7] Red Hat, Inc., „What does an API gateway do?,“ [Võrgumaterjal]. Available: https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do#:~:text=An%20API%20gateway%20is%20an,and%20return%20the%20appropriate%20result.. [Kasutatud 20 3 2021].

[8] Amazon, „Overview of Amazon Web Services,“ 11 19 2020. [Võrgumaterjal]. Available: https://d1.awsstatic.com/whitepapers/aws-overview.pdf. [Kasutatud 23 3 2021].

[9] Amazon, „EC2 Pricing Calculator,“ [Võrgumaterjal]. Available: https://calculator.aws/#/createCalculator/EC2. [Kasutatud 5 5 2021].

[10] Amazon, „Amazon Fargate pricing,“ [Võrgumaterjal]. Available: https://aws.amazon.com/fargate/pricing/. [Kasutatud 4 5 2021].

[11] Amazon, „Amazon Simple Queue Service,“ [Võrgumaterjal]. Available: https://aws.amazon.com/sqs/. [Kasutatud 22 3 2021].

[12] Amazon, „Amazon DynamoDB,“ [Võrgumaterjal]. Available: https://aws.amazon.com/dynamodb/. [Kasutatud 22 3 2021].

[13] Amazon, „Amazon DynamoDB API Reference,“ [Võrgumaterjal]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/Welcome.html. [Kasutatud 22 03 2021].

[14] Amazon, „Pricing for On-Demand Capacity,“ [Võrgumaterjal]. Available: https://aws.amazon.com/dynamodb/pricing/on-demand/. [Kasutatud 23 3 2021].

[15] Amazon, „Amazon DynamoDB Accelerator (DAX),“ [Võrgumaterjal]. Available: https://aws.amazon.com/dynamodb/dax/. [Kasutatud 22 3 2021].

[16] Amazon, „What is a Network Load Balancer?,“ [Võrgumaterjal]. Available: https://docs.aws.amazon.com/elasticloadbalancing/latest/network/introduction.html. [Kasutatud 5 4 2021].

[17] Amazon, „What is an Application Load Balancer?,“ [Võrgumaterjal]. Available: https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html. [Kasutatud 5 4 2021].

[18] Amazon, „Elastic Load Balancing Pricing,“ [Võrgumaterjal]. Available: https://aws.amazon.com/elasticloadbalancing/pricing/. [Kasutatud 4 May 2021].

[19] Oracle, „What Is a Relational Database?,“ [Võrgumaterjal]. Available: https://www.oracle.com/database/what-is-a-relational-database/. [Kasutatud 18 3 2021].

[20] Microsoft, „Relational vs. NoSQL data,“ [Võrgumaterjal]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/relational-vs-nosql-data. [Kasutatud 18 3 2021].

[21] Hazelcast, „What Is an In-Memory Data Grid?,“ [Võrgumaterjal]. Available: https://hazelcast.com/glossary/in-memory-data-grid/. [Kasutatud 29 3 2021].

[22] Hazelcast, „Cloud-native data and compute platform,“ [Võrgumaterjal]. Available: https://hazelcast.org/imdg/. [Kasutatud 29 3 2021].

[23] Hazelcast, „JCache / Java Cache,“ [Võrgumaterjal]. Available: https://hazelcast.com/glossary/jcache-java-cache/. [Kasutatud 29 3 2021].

[24] The Apache Software Foundation., „In-Memory Data Grid,“ [Võrgumaterjal]. Available: https://ignite.apache.org/use-cases/in-memory-data-grid.html. [Kasutatud 30 3 2021].

[25] Oracle Inc., „Coherence Community,“ [Võrgumaterjal]. Available: https://coherence.community/. [Kasutatud 29 3 2021].

[26] Kong Inc., „How to Design a Scalable Rate Limiting Algorithm,“ [Võrgumaterjal]. Available: https://konghq.com/blog/how-to-design-a-scalable-rate-limiting-algorithm/. [Kasutatud 10 3 2021].

[27] A. Bhayani, „System Design: Sliding window based Rate Limiter,“ 5 4 2020. [Võrgumaterjal]. Available: https://www.codementor.io/@arpitbhayani/system-design-sliding-window-based-rate-limiter-157x7sburi. [Kasutatud 10 3 2021].

[28] IBM, „Main features and benefits of message queuing,“ [Võrgumaterjal]. Available: https://www.ibm.com/docs/en/ibm-mq/9.1?topic=queuing-main-features-benefits-message. [Kasutatud 17 3 2021].

[29] VMWare Inc., „Java Client API Guide,“ [Võrgumaterjal]. Available: https://www.rabbitmq.com/api-guide.html. [Kasutatud 21 3 2021].

[30] C. Richardson, „Pattern: API Gateway / Backends for Frontends,“ [Võrgumaterjal]. Available: https://microservices.io/patterns/apigateway.html. [Kasutatud 17 3 2021].

[31] J. Schabowsky, „REST vs Messaging for Microservices – Which One is Best?,“ [Võrgumaterjal]. Available: https://solace.com/blog/experience-awesomeness-event-driven-microservices/. [Kasutatud 1 4 2021].

[32] D. Setrakyan, „Benchmarking Data Grids: Apache Ignite vs Hazelcast, Part II,“ [Võrgumaterjal]. Available: https://www.gridgain.com/resources/blog/benchmarking-data-grids-apache-ignite-vs-hazelcast-part-ii. [Kasutatud 6 4 2021].

[33] Hazelcast, „GridGain/Apache Ignite vs Hazelcast,“ [Võrgumaterjal]. Available: https://hazelcast.com/resources/benchmark-gridgain-apache-ignite-1-5/#bm2. [Kasutatud 2 4 2021].

[34] G. Luck, „Fake Benchmark: Corrected Benchmark shows Hazelcast 3.6 is much faster than GridGain/Apache Ignite 1.5,“ 18 2 2016. [Võrgumaterjal]. Available: https://hazelcast.com/blog/fake-benchmark/. [Kasutatud 6 4 2021].

[35] „Hazelcast DockerHub,“ [Võrgumaterjal]. Available: https://hub.docker.com/r/hazelcast/hazelcast. [Kasutatud 29 3 2021].

[36] GitHub, „Deploying to Amazon Elastic Container Service,“ [Võrgumaterjal]. Available: https://docs.github.com/en/actions/guides/deploying-to-amazon-elastic-container-service?learn=deploy_to_the_cloud. [Kasutatud 4 4 2021].

[37] A. Fang, „Dynein: Building an Open-source Distributed Delayed Job Queueing System,“ 10 12 2019. [Võrgumaterjal]. Available: https://medium.com/airbnb-engineering/dynein-building-a-distributed-delayed-job-queueing-system-93ab10f05f99. [Kasutatud 16 3 2021].

[38] Amazon, „Message Queue,“ [Võrgumaterjal]. Available: https://aws.amazon.com/message-queue/. [Kasutatud 16 3 2021].

[39] Redis, „Introduction to Redis,“ [Võrgumaterjal]. Available: https://redis.io/topics/introduction. [Kasutatud 22 3 2021].

[40] Amazon, „Message Queue Benefits,“ [Võrgumaterjal]. Available: https://aws.amazon.com/message-queue/benefits/. [Kasutatud 16 3 2021].

[41] Amazon, „Amazon SQS message timers,“ [Võrgumaterjal]. Available: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-message-timers.html. [Kasutatud 23 3 2021].

[42] C. Godden-Payne, „What Is Event Driven Architecture, and When Should I Use It?,“ 21 3 2020. [Võrgumaterjal]. Available: https://levelup.gitconnected.com/what-is-event-driven-architecture-and-when-should-i-use-it-1ea9987b85d. [Kasutatud 14 3 2021].

[43] Amazon, „Amazon SQS pricing,“ [Võrgumaterjal]. Available: https://aws.amazon.com/sqs/pricing/. [Kasutatud 22 3 2021].

[44] Amazon, „Using the Amazon SQS message group ID,“ [Võrgumaterjal]. Available: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/using-messagegroupid-property.html. [Kasutatud 24 3 2021].

[45] The Apache Software Foundation, „Distributed Key-Value Store,“ [Võrgumaterjal]. Available: https://ignite.apache.org/use-cases/key-value-store.html. [Kasutatud 30 3 2021].

[46] Microsoft, „What is a Transaction?,“ [Võrgumaterjal]. Available: https://docs.microsoft.com/en-gb/windows/win32/ktm/what-is-a-transaction?redirectedfrom=MSDN. [Kasutatud 25 3 2021].

[47] Google, „Cloud Tasks,“ [Võrgumaterjal]. Available: https://cloud.google.com/tasks. [Kasutatud 14 3 2021].

Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Mina, Jakob Roots

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose „Tarkvarasüsteemi võrguliikluse reguleerimise lahendus teleturundusettevõtte näitel“, mille juhendaja on Priit Raspel
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

17.05.2021

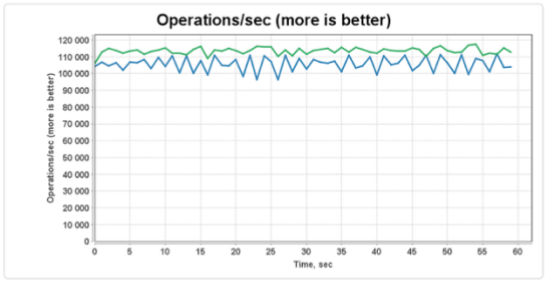
1 Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.

Lisa 2 GridGain jõudlusvõrdlus

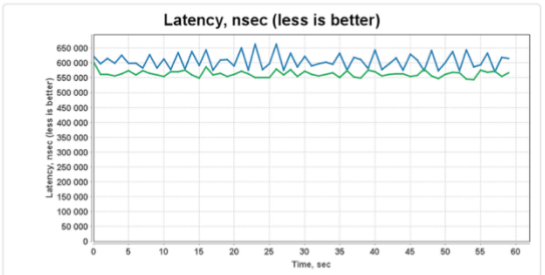
Jõudlustulemuste graafikud on võetud artiklist „Benchmarking Data Grids: Apache Ignite vs Hazelcast, Part II“ [32].

Color	Benchmark	Configurations
	IgnitePutBenchmark	ignite-atomic-put-1-backup
	HazelcastPutBenchmark	hz-atomic-put-1-backup

ThroughputLatencyProbe



	Avg	Min	Max	SD
	113,449.50	106,077.00	117,443.00	2,030.91
	105,591.52	96,409.00	111,956.00	4,225.69



	Avg	Min	Max	SD
	563,311.91	543,798.01	601,551.54	10,639.81
	606,203.54	570,531.05	662,679.54	25,062.77

Color	Benchmark	Configurations
	IgnitePutGetBenchmark	ignite-atomic-put-get-1-backup
	HazelcastPutGetBenchmark	hz-atomic-put-get-1-backup

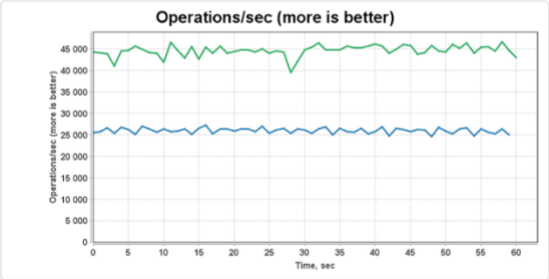
ThroughputLatencyProbe

	Avg	Min	Max	SD
	76,768.33	72,405.00	79,510.00	1,334.76
	74,066.85	69,456.00	77,649.00	2,127.17

	Avg	Min	Max	SD
	832,845.97	803,852.61	882,629.00	15,354.97
	863,920.07	822,976.21	920,226.35	25,344.74

Color	Benchmark	Configurations
	IgnitePutTxBenchmark	ignite-tx-put-1-backup
	HazelcastPutTxBenchmark	hz-tx-put-1-backup

ThroughputLatencyProbe

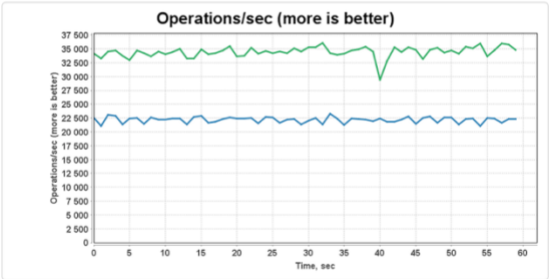


	Avg	Min	Max	SD
	44,588.98	39,528.00	46,612.00	1,295.36
	25,958.32	24,552.00	27,167.00	630.69

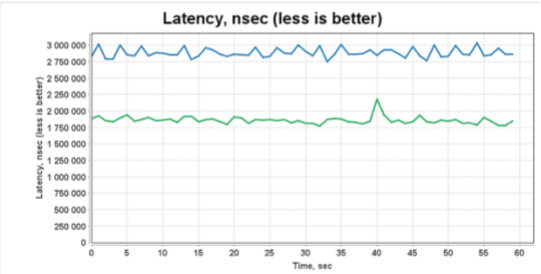
	Avg	Min	Max	SD
	1,435,865.56	1,372,135.56	1,594,051.51	43,942.56
	2,466,361.42	2,354,669.60	2,605,848.09	59,966.59

Color	Benchmark	Configurations
	IgnitePutGetTxBenchmark	ignite-tx-put-get-1-backup
	HazelcastPutGetTxBenchmark	hz-tx-put-get-1-backup

ThroughputLatencyProbe



	Avg	Min	Max	SD
	34,430.12	29,399.00	36,062.00	1,001.80
	22,185.87	21,071.00	23,305.00	529.98



	Avg	Min	Max	SD
	1,859,938.39	1,773,768.31	2,176,456.20	57,817.37
	2,885,730.49	2,746,150.59	3,033,809.97	71,036.37

Lisa 3 Hazelcast jõudlusvõrdlus

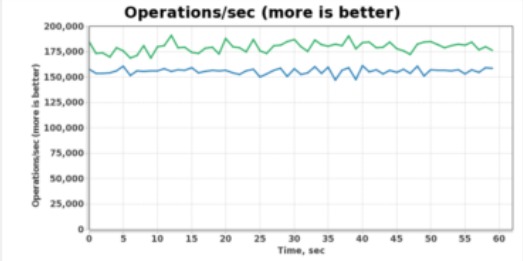
Jõudlustulemuste graafikud on võetud artiklist „GridGain/Apache ignite vs Hazelcast“ [33].

Yardstick

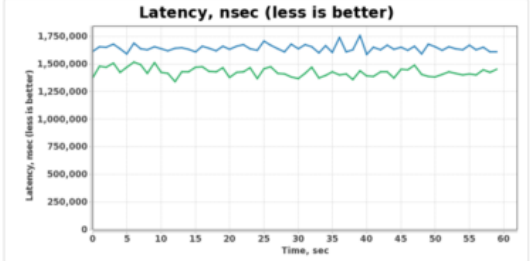
Benchmark Comparison Results

Color	Benchmark	Configurations
	HazelcastPutJcacheBenchmark	hz-jcache-put-3.6-EA
	IgnitePutBenchmark	ignite-full-syncatomic-put-1-backup-1.4.1

ThroughputLatencyProbe



Avg	Min	Max	SD
179,568.33	168,851.00	190,782.00	5,041.38
155,757.25	147,060.00	161,154.00	2,959.46



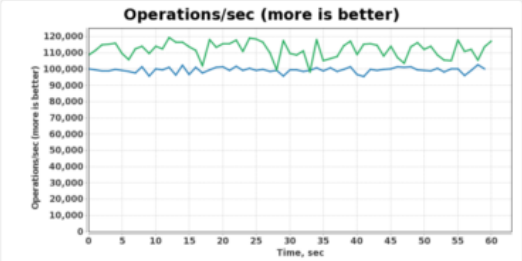
Avg	Min	Max	SD
1,425,996.59	1,340,742.45	1,514,856.81	40,249.02
1,644,003.01	1,586,960.89	1,756,147.92	32,433.89

Yardstick

Benchmark Comparison Results

Color	Benchmark	Configurations
	HazelcastPutGetJcacheBenchmark	hz-jcache-put-get-3.6-EA
	IgnitePutGetBenchmark	ignite-full-syncatomic-put-get-1-backup-1.4.1

ThroughputLatencyProbe



	Avg	Min	Max	SD
	111,997.38	98,377.00	119,243.00	4,930.86
	99,411.22	95,286.00	102,625.00	1,627.75

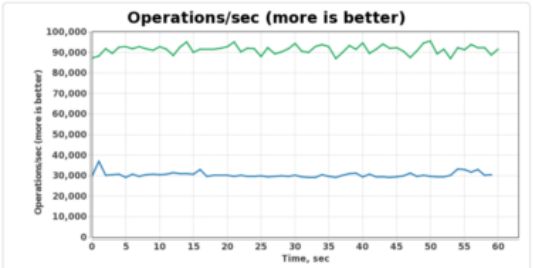
	Avg	Min	Max	SD
	2,288,613.67	2,146,167.95	2,601,313.86	104,725.79
	2,575,413.98	2,494,973.05	2,704,224.53	42,252.43

Yardstick

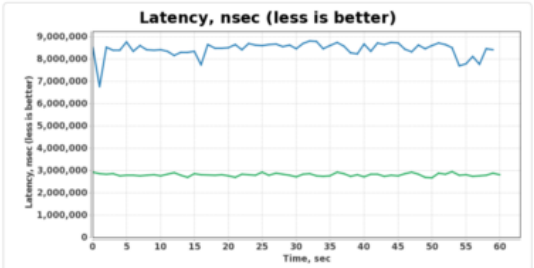
Benchmark Comparison Results

Color	Benchmark	Configurations
	HazelcastPutTxBenchmark	hz-tx-put-1-backup-3.6-EA
	IgnitePutTxBenchmark	ignite-full-synctx-put-1-backup-1.4.1

ThroughputLatencyProbe



	Avg	Min	Max	SD
	91,486.22	86,902.00	95,496.00	2,064.18
	30,384.77	29,082.00	37,000.00	1,282.83



	Avg	Min	Max	SD
	2,800,542.28	2,673,255.65	2,934,290.51	62,490.21
	8,435,672.27	6,769,394.41	8,797,649.95	332,174.10

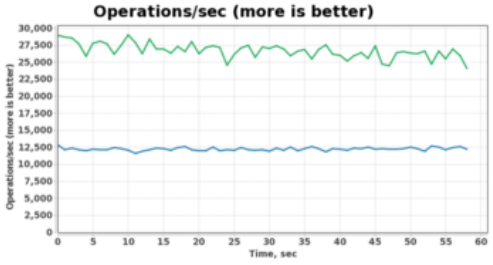
Yardstick

Benchmark Comparison Results

Color	Benchmark	Configurations
	HazelcastPutGetTxPessimisticBenchmark	hz-sync-pess-tx-put-get-1-backup-3.6-EA
	IgnitePutGetTxBenchmark	ignite-full-syncpessimistic-tx-put-get-1-backup-1.4.1

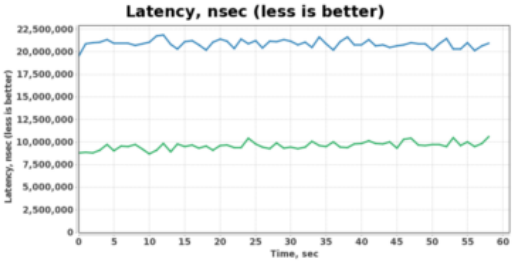
ThroughputLatencyProbe

Operations/sec (more is better)



	Avg	Min	Max	SD
	26,717.00	24,081.00	28,992.00	1,102.39
	12,253.36	11,606.00	12,807.00	232.62

Latency, nsec (less is better)



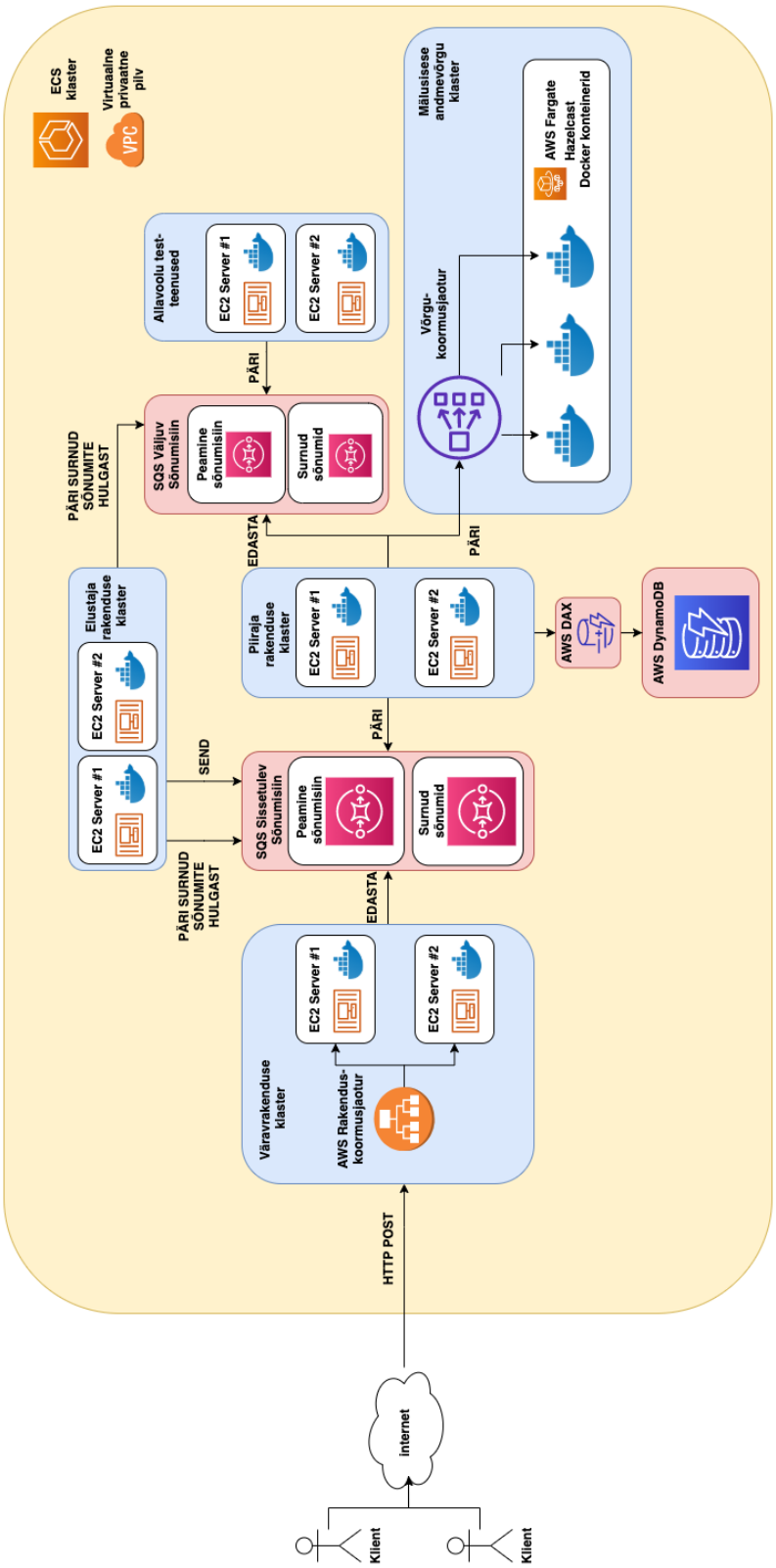
	Avg	Min	Max	SD
	9,601,239.63	8,723,897.61	10,619,701.48	422,546.13
	20,908,249.93	19,541,355.73	21,855,127.51	437,809.26

Lisa 3 Bucket4j algoritmi kasutuse kood

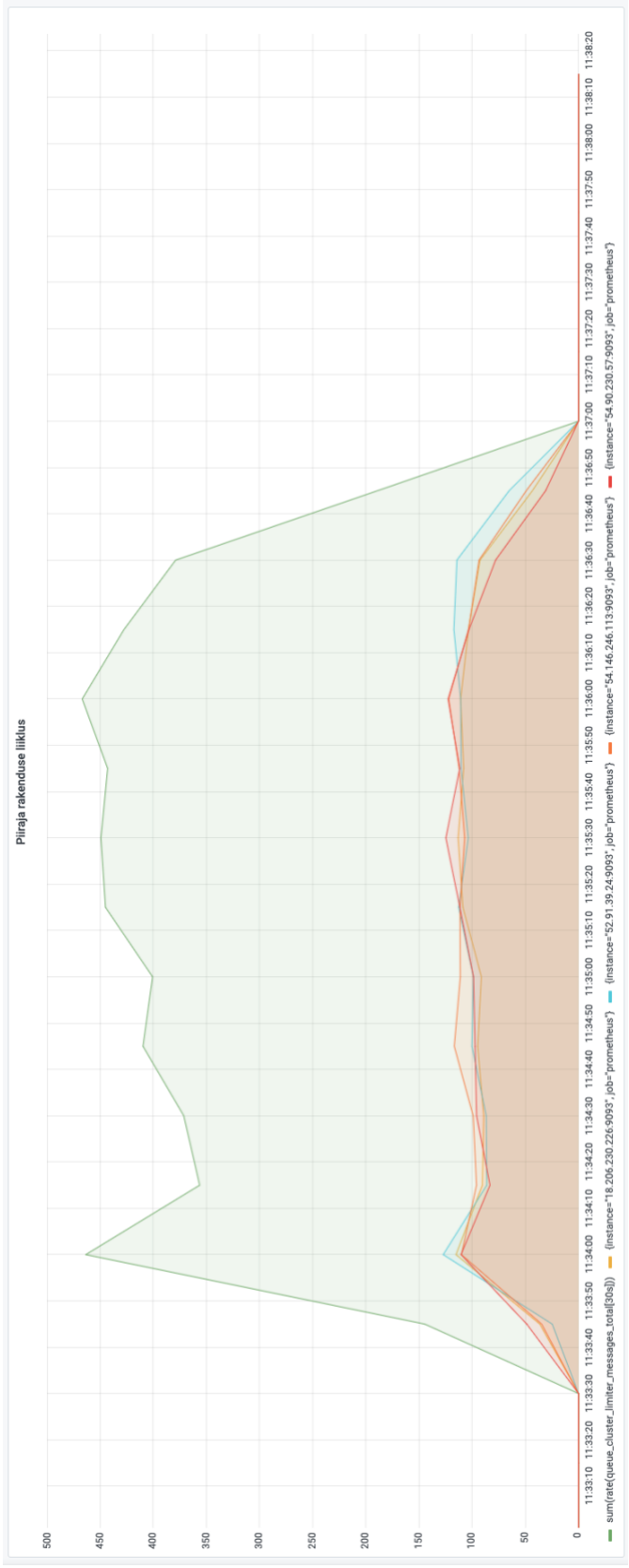
```
public CompletableFuture<Long> võtaŽetoon(Message message,
                                          int piirang) {
    if (andmevõrk == null) {
        //Alusta ühendust Hazelcast andmevõrguga
        startConnection();
    }
    return CompletableFuture.supplyAsync(() -> {
        var identifier = message.getIdentifier();
        long jääkAeg;
        //Proovi kätte saada kliendi piirangute ämber
        var ämberO = buckets.getProxy(identifier);
        if (ämberO.isPresent()) {
            var ämber = ämberO.get();
            var piirangud = map.get(identifier)
                               .getConfiguration()
                               .getBandwidths();
            //Võrdle kliendi ämbri mahtu ja andmebaasist päritud
                sätestatud mahtu
            if (piirangud[0].getCapacity() != piirang) {
                Bandwidth uusPiirang = Bandwidth.simple(
                                       piirang,
                                       Duration.ofSeconds(1));
                BucketConfiguration newConfiguration = Bucket4j
                                       .configurationBuilder()
                        .addLimit(uusPiirang)
                        .build();
                //Uuenda kliendi ämbri mahtu
                ämber.replaceConfiguration(newConfiguration,
                             TokensInheritanceStrategy.AS_IS);
            }
            tokenCounter.inc();
            jääkAeg = TimeUnit.NANOSECONDS.toSeconds(ämber
                               .estimateAbilityToConsume(1)
                               .getNanosToWaitForRefill());
            //Kontrolli, et jääkaeg oleks väiksem kui lubatud
            if (jääkAeg < 30) {
                //Võtaž etoon ämbrist
                ämber.consumeIgnoringRateLimits(1);
            }
        } else {
            //Loo uus piirangute ämber
            var bucket = Bucket4j.extension(Hazelcast.class)
                      .builder()
                      .addLimit(
                   Bandwidth.simple(piirang, Duration.ofSeconds(1)))
                      .build(map, identifier,
                   RecoveryStrategy.THROW_BUCKET_NOT_FOUND_EXCEPTION);
```

```
            bucket.tryConsume(1);
            jääkAeg = TimeUnit.NANOSECONDS.toSeconds(bucket
                                .estimateAbilityToConsume(1)
                                .getNanosToWaitForRefill());
        }
        //Tagasta aeg järgmise žetooni jõudmiseni ämbrisse
        return jääkAeg;
    });
}
```

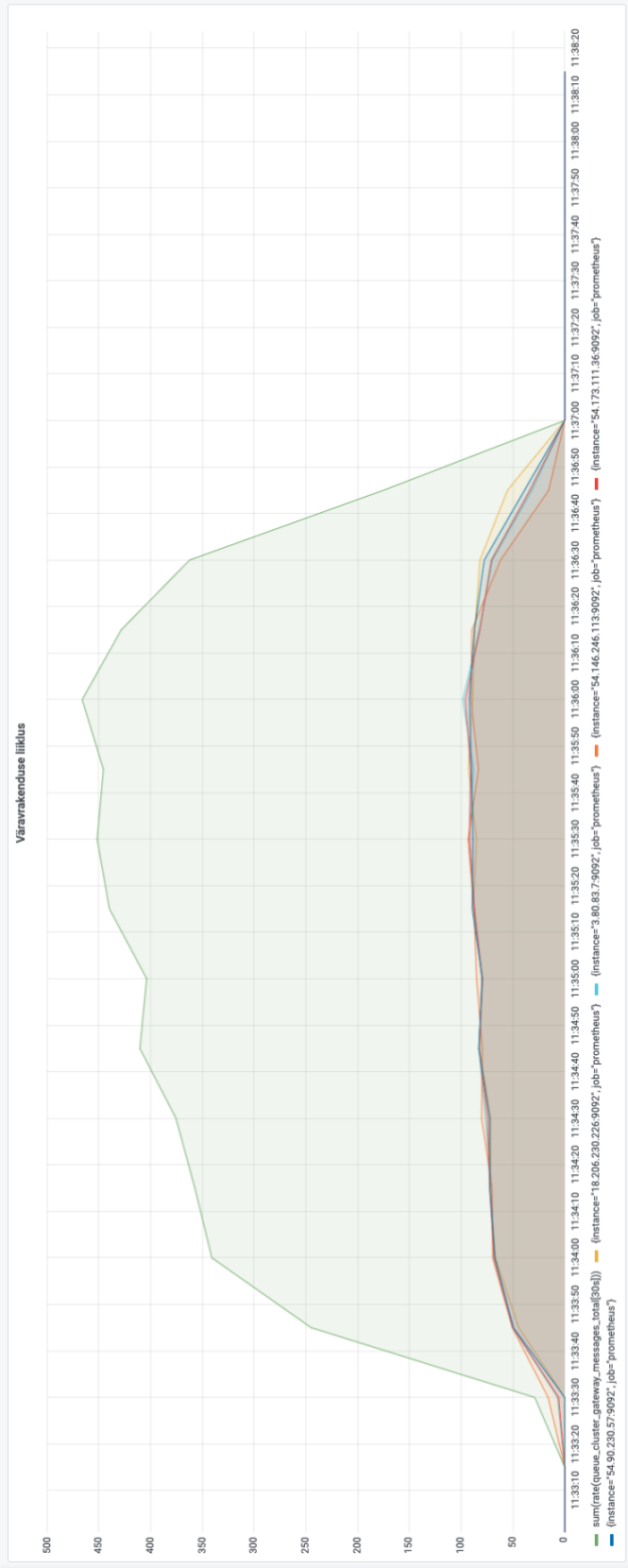
Lisa 4 Süsteemi võrgutopoloogia



Lisa 5 Piiraja rakenduse liiklus



Lisa 6 Väravrakenduse liiklus



Lisa 7 Piiraja rakenduse liiklus