TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Siim Milli

Vahemälu lahendus mikroteenuste kihile telekommunikatsiooniettevõtte näitel

Bakalaureusetöö

Juhendaja: Lauri Anton
Bakalaureusekraad

Tallinn 2023

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: Siim Milli

24.04.2023

Annotatsioon

Seoses eSIM tehnoloogia järjest laialdasema levikuga on kasvanud eSIM tehnoloogiat toetavate seadmete arv ning koos sellega on tekkinud mobiilseadmete tootjatel vajadus rakendada erinevaid mobiilside haldamise funktsioone integreeritult mobiilsideoperaatoriga otse kasutaja seadmes. Individuaalseid seadmeid on võrkudes küllaltki suur hulk, mis toob kaasa ka levinud päringutele suure mahu. Mobiilsideoperaatoritega integreerumiseks on lahenduses kasutusel mikroteenuste kiht.

Antud bakalaureuse töö eesmärk on luua mikroteenuste kihis lahendus levinud päringute ning nende vastuste ajutiseks salvestamiseks. Eesmärgi saavutamiseks analüüsiti mitmeid erinevaid vahemälu lahenduste rakendusviise ning valiti välja sobiv kooslus. Seejärel kirjeldati ja visandati välja valitud lahenduse kasutamist töös kajastatud probleemide korral.

Antud lahendus on rakendatud testkeskkonnas, kus seda on võimalik kasutada kolme turu teenustel, mis paiknevad samas keskkonnas. Lahenduse testimise käigus tuli välja märgatav päringu protsessimise ajaline võit, kui kirje vahemälus juba olemas oli. Samuti saab prognoosida küllaltki suurt päringute mahu vähendamise võimekust lähtesüsteemidele. Antud lahendust on plaanis edasi analüüsida vastavalt ärinõutele ja riskidele ning seejärel seda päris liiklusega keskkonnas kasutusele võtta

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 40 leheküljel, 11 peatükki, 33 joonist, 3 tabelit.

Abstract

Cache Solution for a Microservice Layer on the Example of a Telecommunication Company

With the increasingly widespread distribution of eSIM technology in the consumer devices, there’s a lot of new integrated services coming out by the device vendors. Idea behind these services is to apply the eSIM technology over the air connection delivery options integrated into native user interfaces of the device. This means that the device types which have a service enabled with the mobile network operator in a region start to ping the technical systems a lot and this has increasing trend as there’s more and more devices with such support coming out. For the connection towards the operators backends systems there’s a microservice layer in between which hides the complexity of the different source systems.

With the recently launched services the load for the operator’s backend systems have gone up multiple times and this has brought out problems where the load has been too high for some systems. There’s a good amount of recurring request happening for rather immutable data which could be handled by caching the responses in microservice layer for a period to reduce the amount of the requests done towards the downstream backend systems.

This thesis main goal is to implement a caching solution for the microservice layer to optimise the request count towards the other systems. Solution design was achieved by analysing the options, approaches, strategies, and technologies available to store data in a memory. Solution components are implemented and deployed in the test environment where the initial testing has bee carried. During the testing it was clear that there are performance gains available by the developed solution and as well the amount of requests could be reduced if the proper time to live is set. Plan forward is to analyse more on business requirements and possible risks about serving stale data.

The thesis is in Estonian and contains 40 pages of text, 11 chapters, 34 figures, 3 tables.

Lühendite ja mõistete sõnastik

Alpine Linux	Väikse Linux distributsioon.
API	*Application Programming Interface*; Rakendusliides.
Backend	Rakenduse serveri poolne osa.
Bash	Käsurea keel
Cache Aside	Vahemällu andmete salvestamise lähenemis viis, kus rakendus suhtleb vahemäluga eraldiseisvalt.
Cache hit	Andmete leidmine vahemälust.
Cache miss	Andmeid ei leitud vahemälust.
Client Server Cache	Kliendi lähedal asuv eraldiseisev vahemälu komponent.
ConfigMap	Kubernetese objekt mis võimaldab salvestada konfiguratsiooni teavet rakendusest eraldi.
Consistent Hashing	Algoritm, mida kasutatakse andmete hajutamiseks arvutivõrgus, tagades ühtlase jaotumise.
DB-Engines	Veebisait, kus saab võrrelda andmehoidlaid.
Docker	Platvorm rakenduste konteineriseerimiseks.
Docker Hub	Teenus rakenduste *Docker* piltide jagamiseks ja haldamiseks.
Embedded Cache	Vahemälu rakenduse osana.
eSIM	Mobiil seadmesse integreeritud *SIM* kaart.
Hashslot	Redise kasutatav algoritm, andmete hajus jagamisel instantside vahel.
Hazelcast	Avatud lähtekoodiga operatiivmälu andmevõrk.
IP	Internet Protocol; Masina aadress internetis
Java	Programmeerimiskeel.
Java	Objektorienteeritud programmeerimiskeel.
Jedis	*Java* klient teek *Redisele.*
Kubernetes	Avatud lähtekoodiga konteiner orkestratsiooni platvorm.
Lettuce	*Java* klient teek *Redisele.*
Load Balancer	Koormuse haldur, mis jagab töökoormust mitme instantsi vahel.

Master	Primaarne instants Redise kobar süsteemis.
Memcached	Avatud lähtekoodiga operatiivmälus andmete hoidmise tehnoloogia.
MOD	Matemaatiline tehe, mis leiab jäägi kahe täisarvu jagamisel.
MSISDN	*Mobile Station Integrated Services Digital Network*; Telefoni number rahvusvahelisel tasemel, mis unikaalselt identifitseerib mobiilside lepingut.
N	Number.
NoSQL	Mitte relatsiooniline andmete salvestamise viis.
PersistentVolumeClaim	Kubernetese objekt, mis broneerib salvestus ruumi.
PersistentVolumes	Kubernetese objekt, mis esindab tükikest salvestus ruumis kobar süsteemis.
Pod	Väikseim *Kubernetese* loogiline üksus.
Rancher	Kasutajaliides mitmete Kubernetese kobar süsteemide haldamiseks.
Read Through	Vahemällu andmete salvestamise viis, kus vahemälu nõudlusel ise küsib andmed andmeallikast ning salvestab.
Redis	Avatud lähtekoodiga operatiivmälu andmehoidla.
Redis Alpine	*Alpine Linux* baasil *Redise Docker* pilt
RedisTemplate	*Spring Data Redis* klass, mis tagab *Redisega* suhtluse
Refresh Ahead	Vahemällu andmete salvestamise viis, kus vahemälu iseseisvalt aktiivselt uuendab andmeid.
REST	*Represntational State Transfer*; Arhitektuuri stiil veebirakendustes, mis võimaldab ühtset rakendusliideste disaini.
Sentinel	Redise tehnoloogia, mis jälgib ja juhib *Redise* instantse
Service	Kubernetese objekt, mis võimaldab teenuste avastamist kobar süsteemis.
Sidecar Cache	Konteiner platvormide erine vahemälu rakendamise viis, kus vahemälu instants on samas konteiner platvormi loogilises üksuses, kus rakendus ise on.
SIM	*Subscriber Identity Module*; Kiipkaart mis hoiab identifitseerimisteavet, mida kasutatakse tellija tuvastamiseks ning mobiilsidevõrku autentimiseks.
Slave	Sekundaarne instants Redis kobar süsteemis.
Spring	*Java* platvormi raamistik
Spring Data Redis	*Spring* raamistik *Redise* toetusega.
StatefulSet	Kubernetese töövoog olekupõhiste rakenduste juhtimiseks

VolumeClaimtTemlpates	Kubernetese konfiguratsiooni osa, mis määrab rakenduse instantsile püsiva mälu.
Write Through	Vahemällu andmete salvestamise viis, kus vahemälu vastutab kirjutamise operatsiooni käskluse edastamise eest andmeallikasse.
Yaml	Keel konfiguratsiooni failide jaoks.

Sisukord

1 Sissejuhatus 14
2 Hetke lahenduse ülevaade 15
2.1 Lahenduse äriline eesmärk 15
2.2 Hetke lahenduse lihtsustatud kirjeldus 15
2.3 Mikroteenuste eesmärk 16
3 Lahendatav probleem 16
3.1 Probleemi kirjeldus 16
3.2 Eesmärk 17
3.3 Lähtetingimused 17
3.4 Metoodika 17
4 Kaardistamine 18
5 Probleemi analüüs 18
5.1 Lähtesüsteemide koormuse probleem 19
5.2 Probleemi päritolu 19
5.3 Viivitus lähtesüsteemis 19
5.4 Vahemälu kasutamise riskid 20
5.5 Nõuded lahendusele 20
5.5.1 Funktsionaalsed nõuded 20
5.5.2 Mittefunktsionaalsed nõuded 20
6 Vahemälu lahenduste analüüs 21
6.1 Andmete laadimise lähenemisviisid 21
6.2 Andmete laadimise strateegiad 21
6.2.1 Cache Aside 21
6.2.2 Read Through 22
6.2.3 Write Through 23
6.2.4 Refresh Ahead 23
6.3 Vahemälu asukoht 24
6.3.1 Embedded Cache 24
6.3.2 Client-Server Cache 25

6.3.3 Sidecar Cache 25
6.4 Vahemälu kehtetuks tunnistamine 26
6.5 Vahemälu tühjendamine 26
7 Vahemälu tehnoloogiad 27
7.1 Redis 27
7.1.1 Rakendus mudelid 27
7.1.2 Andmete püsivus 29
7.2 Memcached 30
7.2.1 Rakendus mudelid 30
7.2.2 Andmete püsivus 31
7.3 Hazelcast 31
7.3.1 Rakendus mudelid 31
7.3.2 Andmete püsivus 32
8 Vahemälu lahenduse valik ja kirjeldus 32
8.1 Andmete laadimise strateegia valik 32
8.2 Vahemälu asukoha valik 33
8.3 Tehnoloogia valik 33
8.4 Vahemälu lahenduse kirjeldus 35
8.5 Vahemälu lahendus eraldi seisva teenusena 35
8.6 Vahemälu rakendamine koormus probleemi korral 36
8.7 Vahemälu rakendamine andmete varajasel kättesaamisel 37
9 Lahenduse Arendus 38
9.1 Redis kobar süsteemi rakendamine Kuberneteses 38
9.1.1 Redis konteiner pilt 38
9.1.2 Kubernetes Configmap 39
9.1.3 Kubernetes Statefulset 40
9.1.4 Kubernetes Service 40
9.1.5 Redis instantside kasutuselevõtt 41
9.1.6 Redis kobar süsteemi moodustamine 42
9.2 REST API mikroteenus Redis kliendiga 44
9.2.1 Redis Java konfiguratsioon 44
9.2.2 Redis Repository 46
9.2.3 API kontrollerid 47
9.2.4 Vahemälu teenuse projekti struktuur 48

10 Tulemused 49
10.1 Valmidus tase ja testimine 49
10.2 Mõõtmistulemused 49
10.3 Nõuete täitmine 51
10.4 Edasised plaanid 51
11 Kokkuvõte 52
Kasutatud kirjandus 53
Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks 56
Lisa 2 – Kubernetese Statefulset 57

Jooniste loetelu

Joonis 1. Tervik lahenduse lihtsustatud ülevaade 15
Joonis 2. Andmete lisamise strateegia cache aside 22
Joonis 3. Andmete lisamise strateegia read through 22
Joonis 4. Andmete lisamise strateegia write through 23
Joonis 5. Andmete laadimise strateegia refresh ahead 23
Joonis 6. Teenuse sees asuv privaatne vahemälu 24
Joonis 7. Teenuse sees asuv jagatud vahemälu kobar süsteem 25
Joonis 8. Eraldiseisev vahemälu server 25
Joonis 9. Rakendusega samas konteiner platvormi rakendus üksuses asuv vahemälu .. 26
Joonis 10. Redis kobar süsteem 35
Joonis 11. API kontroller vahemälu lahendusele 36
Joonis 12. Vahemälu kasutamine koormus probleemi korral 36
Joonis 13. Õnnestunud toote tellimusel lisatakse tellimuse andmed vahemällu. 37
Joonis 14. Toote tellimuse staatusele vastav mikroteenus saab toote tellimuse kohta küsida vahemälust. 37
Joonis 15. Redis konteiner pildi kohalikku hoidlasse laadimine 38
Joonis 16. Kubernetese configmap Redise jaoks 39
Joonis 17. Kubernetes Service Redis seadistuse jaoks 41
Joonis 19. Redis instantside rakendamine Kubernetese käsureal 41
Joonis 19. Redis instantsid Kuberneteses 41
Joonis 20. Redis instantsid Rancher kasutajaliideses 42
Joonis 21. Redis kobar süsteemi info käskluse tulemus ennem 42
Joonis 22. Redis kobar süsteemi instantsid esimese instantsi vaatest 42
Joonis 23. Redis kobar süsteemi seadistuse käsklus ja vastus 43
Joonis 24. Redis kobar süsteemi info käsklus peale kobar süsteemi konfiguratsiooni .. 44
Joonis 25. Spring Redis data sõltuvus 44
Joonis 26. Jedis kliendi sõltuvuse lisamine 45
Joonis 27. Redis kobar süsteemiga ühenduse loomine Spring teenusest 45
Joonis 28. Redis template seadistus 45

Joonis 29. Geneeriline Repository klass 46
Joonis 30. Kirje lisamise API kontroller 47
Joonis 31. Kirje eemaldamise kontroller 47
Joonis 32. Kirje otsimise kontroller 48
Joonis 33. Vahemälu teenuse projekti struktuur 48
Joonis 34. Lahenduse testimine 49

Tabelite loetelu

Tabel 1. Mõõtmis tulemused ennem 18
Tabel 2. Mõõtmis tulemused ennem 50
Tabel 3. Mõõtmis tulemused pärast test keskkonnas 50

1 Sissejuhatus

Tänapäeval mobiilside võimekuse haldamine mobiilseadmetes on muutumas järjest rohkem integreeritud kogemuseks kasutaja individuaalses mobiilseadmes. Seda protsessi veab ja võimaldab *eSIM* tehnoloogia, kus *SIM* kaart on viidud füüsiliselt seadme sisse ning mida saab hallata eemalt. Ehk mobiilside võimekuse saavutamiseks, ei ole vaja enam füüsilist plastikul *SIM* kaarti, mille kasutaja oma seadmesse panema peab, vaid *eSIM* on võimalik üle õhu seadmesse saata. See progress on toonud kaasa suurel hulgal erinevaid mobiilsideoperaatoriga integreeritud teenuseid seadme tootjatelt, et pakkuda kasutajatele mugavat seadmest lahkumata kogemust, et saavutada mobiilside võimekus.

Selline integreeritud lähenemine, eeldab toetavatelt süsteemidelt teistsugust lähenemist, kui see oli füüsiliste SIM kaartide ajastul, kus oli protsesside realiseerimiseks rohkem aega ning mis ei olnud otse seotud seadmete tootja kasutajakogemusega. Lisaks on kõik mõnda integreeritud võimekust omavad seadmed pidevas ühenduses operaatorite tehniliste tugi-süsteemidega, mis mobiiltelefonitele mõeldud lahenduste puhul on toonud kaasa plahvatusliku koormuse kasvu võimekust võimaldava info küsimisel.

Mobiilseadmes integreeritud teenuseid on rakendatud mobiilsideoperaatori kõikidel turgudel ühise lahendusena. Turgude süsteemide keerukuse peitmiseks on lahenduses mikroteenuste integratsioon kiht. Töö eesmärk on antud mikroteenuste kihis rakendada vahemälu lahendust, mis osaliselt vähendaks koormust turgude süsteemidele. Ning lisaks, et tekitada võimekus teatud andmeid ennem turu süsteemidest kätte saada, kui need sealt saadaval on.

Selle jaoks autor analüüsib vahemälu rakendamise lahendusi mikroteenuste arhitektuuris ja võrdleb võimalike lahenduste sobivust antud mikroteenuste arhitektuurile koos terve lahenduse eripäradega. Peale seda koostatakse võimaliku lahenduse ülevaade ning rakendatakse antud lahendust ühe test turu raames.

2 Hetke lahenduse ülevaade

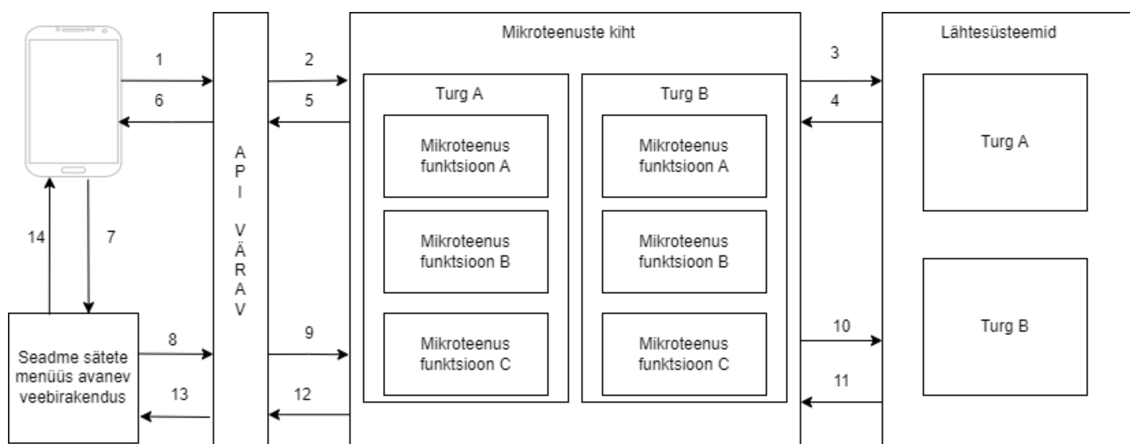
Antud peatükk annab lihtsustatud ülevaate tervik lahendusest, kus antud töös kajastatav mikroteenuste kiht asub. Lihtsustatud ülevaate eesmärk on anda hiljem püstitatavale probleemile lisa sisu.

2.1 Lahenduse äriline eesmärk

Lahenduse äriline eesmärk on pakkuda kasutajate seadmetes funktsioone mobiilside pakettide ning *SIM* ja *eSIM* kaartide haldamiseks. Funktsioonid saavad olla näiteks uue mobiilside paketi *eSIM* seadmesse paigaldamine, olemas oleva paketi lõpetamine või SIM kaardi vahetus uude *eSIM* seadmesse.

2.2 Hetke lahenduse lihtsustatud kirjeldus

Tervik lahendus koosneb *API* väravast, mikroteenuste kihist, veebirakendusest ja välistest lähtesüsteemidest. *API* värav tegeleb peamiselt süsteemi autoriseerimise ja päringu õigesse mikroteenusesse marsruutimisega, mikroteenused tegelevad lähtesüsteemiga integreerumisega, veebirakendus tegeleb kasutaja interaktsiooniga ning välised lähtesüsteemid on andmete omanikud. Joonis 1 esitab lihtsustatud ülevaate.



Joonis 1. Tervik lahenduse lihtsustatud ülevaade

2.3 Mikroteenuste eesmärk

Mikroteenuste kiht on integratsiooni kihiks mobiilseadmetele ning veebirakendusele sügavamale turgude lähtesüsteemidesse. Antud integratsiooni kihi eesmärk koos juurde kuuluva *API* väravaga on peita mobiilseadmete ja veebirakenduse eest lähtesüsteemide keerukus. Nii mobiilseadmed kui ka veebirakendus saavad sama eesmärgiga funktsiooni alati üheselt pärida, kasutades sama andmemudelit mitmel erineval turul.

3 Lahendatav probleem

Järgnevalt kirjeldatakse probleemi, antakse ülevaade eesmärgist ja lähtetingimustest ning tuuakse välja metoodika, kuidas tulemuseni jõuda.

3.1 Probleemi kirjeldus

Seoses mobiilseadmete tootjate mitmete mobiilsideoperaatoriga integreeritud teenuste rakendamisega mitmetel mobiilsideoperaatori turgudel ning antud võimekust omavate mobiilseadmete kasvuga on toimunud kõrge päringute kasv süsteemile. Antud koormuse kasv on kohati probleemiks lähtesüsteemidele. Suure osa päringuid teevad mobiilseadmed ilma kasutaja poolse tegevuseta, et seadme pärilikus kasutajaliideses liideses mõnda mobiilsideoperaatoriga integreeritud funktsiooni võimekust kuvada. Päringud on alati *SIM* kaardi põhised ning lisaks funktsioonide võimekusele sisaldab vastus ka *SIM* kaardiga seotud muud informatsiooni.

Teiseks võib esineda olukordi, kus ühe ahelas varem paikneva mikroteenuse poolt uuendatud info mobiilsideoperaatori lähtesüsteemis ei ole sama lähtesüsteemi teisest komponendist koheselt kätte saadav sama mikroteenuste keskkonna teisele ahelas hiljem paikenvale mikroteenusele. See kahjustab kasutaja kogemust mõnevõrra mobiilseadme pärilikus kasutajakogemuses, kuid ei tekita funktsionaalset kahju.

3.2 Eesmärk

Peamine eesmärk on vähendada kõige kõrgema koormusega mikroteenuse päringute mahtu lähtesüsteemi. Selleks, et vähendada päringute mahtu lähtesüsteemi, on vaja juba küsitud infot talletada mikroteenuste kihile lähemal. Vahemälu kasutamine on levinud viis vähendamaks päringute arvu teistesse süsteemidesse [1]. Eesmärk on analüüsida erinevaid vahemälu lahendusi ning valida välja sobiv ning rakendada antud lahendust mikroteenuste kihile ühe turu raames. Lisaks päringute mahu vähendamisele tõstab vahemälu kasutamine ka teenuste kihi jõudlust ning kättesaadavust [1]. Lisaks on eesmärgiks, et sama vahemälu lahendust saaks ka rakendada muudele funktsioonidele.

3.3 Lähtetingimused

Loodav vahemälu lahendus peab vähendama lähtesüsteemidesse tehtavate päringute mahtu tõstes samal ajal mikroteenuste kihi jõudlust. Lahendus peab olema universaalselt kasutatav kõikidele lahenduses olevatele mikroteenustele. Lahendus peab omama aegumise loogikat või vastavat seadistust real ajas võimaldama. Lisaks talletamisele peab olema võimalus ka nõudlusel kustutamine. Kirjete hoidmise maht peab olema kergesti kasvatatav.

3.4 Metoodika

Korrastatud lähenemine antud lõputöö koostamiseks sisaldab järgnevaid väiksemaid protsessi osasid, mille tulemusel saavutatakse lõpp tulemus.

- Probleemi kirjeldus
- Hetke jõudluse ja koormuse kaardistamine.
- Probleemi analüüs
- Nõuded
- Vahemälu lahenduste ja tehnoloogiate analüüs
- Valida välja sobi lahendus ja tehnoloogia.
- Lahenduse analüüs, mis kasutab vahemälu

- Arendada ja seadistada nõuetele vastav vahemälu lahendus
- Saavutatud tulemuse kaardistamine, võrdlemine ennem saadud tulemustega ning hinnangu andmine tulemustele.

4 Kaardistamine

Autor kaardistab hetke koormuse päringute ahelas alati esimesel kohal paikneva mikroteenuse raames kolmel operaatori turul. Valim annab ülevaate erineva jõudlusega lähtesüsteemidest ning samuti kajastab erinevat koormust turgudel. Koormuse maht on tingitud teenustest, mis antud turul võimaldatud on, ning vastavat võimekust omavate kliendi seadmete hulgast. Antud mikroteenus pakub kahte rakendusliidest, üks mida kasutab mobiilseade otse, et saada kliendi *SIM* kaardi *MSISDN* ja binaarsed väärtused funktsioonide kohta, mis antud *SIM* kaardi omanikule lubatud on. Teist rakendusliidest kasutab veebirakendus. Päringud mobiilseadmetelt tekitavad üle 99.9% päringutest. Päringud mobiilseadmetelt peaksid juhtuma ühe korra 24 tunni jooksul, kui ei ole kasutaja poolset tegevust. Kuid autor on tähele pannud, et klientidel on seadmeid, mis pärivad palju tihedamalt. Tabel 1 esitab arvestuse 24 tunni kohta.

Tabel 1. Mõõtmis tulemused ennem

Turg	**Päringute maht**	**Päringud lähtesüsteemi**	**Unikaalsed päringud**	**Keskmine vastuse aeg**	**Õnnestumise %**
A	1,742,357	1,742,357	1,069,589	364ms	99.853%
B	986,201	986,201	687,928	222ms	99.696%
C	149,843	149,843	106,545	1701ms	99.581%

5 Probleemi analüüs

Antud peatükis esitatakse probleemi tekke põhjust, kirjeldatakse probleemi ning analüüsitakse lahenduse riske.

5.1 Lähtesüsteemide koormuse probleem

Mikroteenuste kihi kontekstis esinev probleem seisneb liigsete päringute tegemises lähtesüsteemidele, millest andmeid kogutakse. Lähtesüsteem antud kontekstis on enamasti järgmine teenus sügavamale konkreetse turu süsteemidesse.

Sisenemispunkt ja ahel edasi ei ole autori kontrolli all. Mida rohkem päringuid seda rohkem on võrguliiklust ja koormust, eriti kui samu andmeid päritakse sageli lühikese ajajooksul. Selline koormus võib mõjutada kogu süsteemi jõudlust ning põhjustada ebamugavusi lõppkasutajatele. Üheks päringute mahu vähendamise lahenduseks on vahemälu kasutamine, mille kasutamine kliendi ja lähesüsteemi vahel aitab vähendada koormust lähesüsteemile [2].

5.2 Probleemi päritolu

Koormuse probleem lähtesüsteemidele tekkis turgudel, kus hakati toetama lahendust mida toetavaid seadmeid on turgudel oluliselt rohkem. See tähendab, et päringute maht kasvas hüppeliselt võrreldes turgudega, kus antud lahendust veel ei ole. Tabel 1 näitab päringute mahu erinevust Turg A ja Turg B toetavad antud lahendust ning Turg C ei toeta.

5.3 Viivitus lähtesüsteemis

Teiseks probleemiks on kohatised viivitused lähtesüsteemides. Peale uue tellimuse sooritamist läbi mobiilseadme päriliku kasutaja liidese on kasutaja kogemuse jaoks vajalik koheselt peale tellimust seadmelt tulevale tellimuse staatuse päringule vastata uue tellimuse informatsiooniga. Tellimuse saatus võib olla antud hetkel – edenemisel. Antud päring jääb minutilise intervalliga seadmelt tulema kuni tagastatakse – aktiivne. Viivitus uue tellimuse informatsiooni kätte saamisel pärsib kasutajakogemust pärilikus kasutaja liideses. Kui edenemisel ja aktiivne staatuste puhul näeb kasutaja vastavaid staatuseid ka pärilikus kasutajaliideses siis olukorras, kus tellimuse info pole üldse kättesaadav olnud ei näe kasutaja hetkeks mitte midagi oma uue tellimuse kohta. Vahemälu kasutatakse ka süsteemi robustsuse suurendamiseks lubades toiminguid, kui info lähtesüsteemist pole kätte saadav [2].

5.4 Vahemälu kasutamise riskid

Arvutiteaduses on ainult kaks rasket asja – vahemälu kehtetuks tunnistamine ja asjade nimetamine [7]. Vahemälu kasutamine üks riskidest on aegunud andmete kasutamine[3]. See on ka riskiks antud lahenduses, kui hakata kasutama vahemälu mikroteenuse kihis, et vähendada tehtavate päringute mahtu lähtesüsteemidele. Mikroteenused ei tea, kui kasutatavad andmed muutunud on.

5.5 Nõuded lahendusele

5.5.1 Funktsionaalsed nõuded

Vahemälu lahendus peab vastama järgmistele funktsionaalsetele nõuetele.

- Peab võimaldama salvestada sageli kasutatavat teavet.
- Peab võimaldama salvestada teavet mitmelt erinevalt mikroteenuselt.
- Peab võimaldama küsida salvestatud teavet.
- Peab võimaldama andmete aegumise sätestamist ja kehtetuks tunnistamist.
- Peab võimaldama sündmuse või nõudluse põhist andmete kehtetuks tunnistamist.

5.5.2 Mittefunktsionaalsed nõuded

Vahemälu lahendus peab vastama järgnevatele mittefunktsionaalsetele nõuetele.

- Peab olema laiendatav järgmistele teiste turgude mikroteenustele.
- Peab olema laiendatav uute vajaduste tekkel.
- Peab olema lihtsalt skaleeritav andmete ja päringute mahu kasvamisel.
- Peab olema kõrge kättesaadavusega.
- Peab olema hea jõudlusega.

6 Vahemälu lahenduste analüüs

Selles peatükis autor uurib erinevaid andmete vahemällu laadimise lähenemisviise ja salvestamise strateegiaid ning ka erinevaid kohti, kus vahemälu rakendada. Näiteks mikroteenusesse sisseehitatud osana või eraldiseisva teenusena. Lisaks uuritakse erinevaid vahemälu tehnoloogiaid nagu populaarsed *Redis* ja *Memcached*.

6.1 Andmete laadimise lähenemisviisid

Kaks peamist lähenemisviisi andmete laadimiseks vahemällu on eellaaditud ja laisklaaditud. Eellaaditud juhul teatakse andmeid ennem ja laetakse andmed ennem päringu saamist vahemällu, mis juhul on andmed salvestatud ennem nõudlust. Laisklaaditud vahemäludes laetakse andmed vahemällu nõudlusel, seega esimese päringu peale küsitakse andmed andmeallikast ning seejärel salvestatakse vahemällu [4].

Antud mikroteenuste keskkond ei ole kunagi teadlik kõikidest andmetest. Esiteks on mitmed lähtesüsteemid erinevate turgude raames, kus andmed asuvad ning mis ei ole autori kontrolli all. Teiseks on vaja ainult andmeid teatud *SIM* kaartidest, mis parasjagu asuvad mobiilseadmetes, millel on võimekus mikroteenuste keskkonnale päringuid tekitada. Seega antud mikroteenuste kihis on võimalik kasutada ainult laisklaaditud lähenemisviisi.

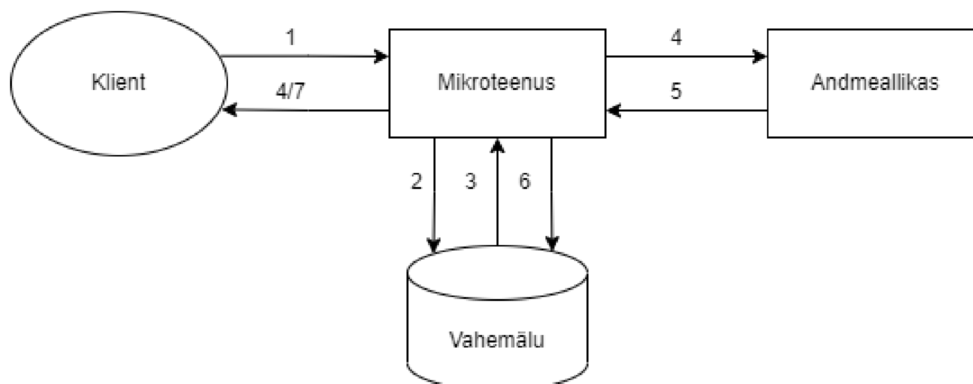
6.2 Andmete laadimise strateegiad

Andmete laadimisel vahemällu, on meil valida mitmete strateegiate vahel millest osad lähenemised on ennetavad ehk eellaaditud ja osad reaktiivsed ehk laisklaaditud [5].

6.2.1 Cache Aside

Tegemis on ühe levinuma strateegiaga, kus mikroteenus suhtleb nii vahemäluga kui ka välise teenusega. Selle strateegia puhul andmed laisklaetakse vahemällu. Päringu saades mikroteenus kõigepealt küsib vahemälust, kui vahemälus on vastus olemas on meil *cache hit* ning vastus tagastatakse koheselt. Kui vahemälus antud päringule vastust ei ole on meil *cache miss*. Kui vahemälust vastust ei leitud, peab mikroteenus küsima

andmeallikast, mis järel salvestatakse kirje vahemällu ning tagastatakse vastus kliendile [5]. Joonis 2 annab ülevaate strateegiast.

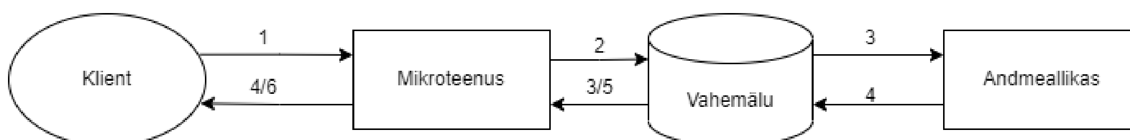


Joonis 2. Andmete lisamise strateegia cache aside

Antud strateegia töötab väga hästi suure lugemiskoormuse puhul andmetele, mida ei muudeta tihti [6]. Samuti on see strateegia vastupidav vahemälu tõrgetele, juhul kui toimub vahemälu täielik tõrge, saab teenus endiselt andmeid lähtesüsteemist [5]. Lisaks peab vahemälu lahendus tegelema ainult kirjete lisamise ja otsimisega ning rakendus tegeleb välise teenusega ühenduse rakendamisega [7].

6.2.2 Read Through

Siinkohal suhtleb mikroteenus ainult vahemäluga, ning vahemälu suhtleb andmeallikaga ehk vahemälu on mikroteenuse ja andmeallika vahel. Tegemist on samuti laisklaetud lähenemisviisiga. Mikroteenus saadab päringu alati vahemälu komponendile, sarnaselt eelmisele strateegiale ka siin tagastatakse vastus koheselt, kui vahemälul oli see olemas. Juhul, kui vahemälus vastavad andmestiku ei olnud, saadab vähemälu päringu andmeallikale. Seejärel vastus salvestatakse ning vahemälu vastab mikroteenusele [5]. Tegemist on murede lahususe põhimõttega, kus mikroteenus suhtleb ainult vahemäluga. Ühenduse rakendamine ja andmete sünkoriseerimine andmeallikaga on vahemälu ülesanne [7]. Joonis 3 kirjeldab strateegiat.

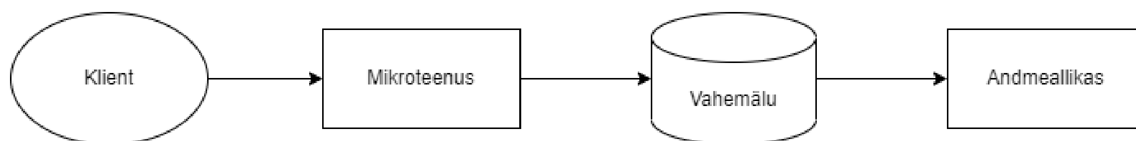


Joonis 3. Andmete lisamise strateegia read through

See strateegia on kasulik koos kasutamiseks järgnevalt kirjeldatava *write through* strateegiaga juhul, kui meil on andmed mida on kasulik hoida vahemälus ning millel juhtuvad muudatused, kuid need muudatused peavad olema sama komponendi kontrolli all. Nõrk koht selle strateegia puhul on vahemälu rike, mis juhul ei ole võimalik enam üldse päringutele vastata [5].

6.2.3 Write Through

Sarnane eelmisele strateegiale, kus lugemis operatsiooni puhul käis liiklus läbi vahemälu andmeallikani siis siin toimub sama tegevus kirjutamise operatsioonil. Ehk kui rakenduse uuendab või lisab andmeid, siis seda tehakse läbi vahemälu, kus vahemälu salvestab uue kirje ning seejärel uuendab andmeallika [5]. Joonis 4 kirjeldab seda protsessi.

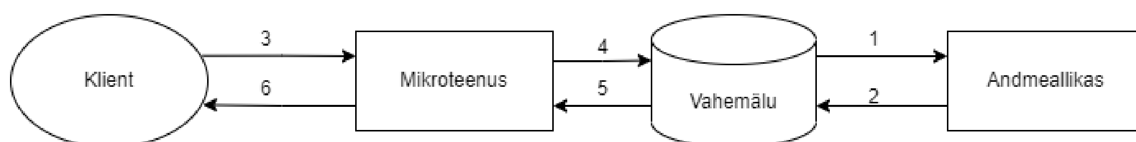


Joonis 4. Andmete lisamise strateegia write through

Ainult kirjutamise operatsioonil üksi ei ole mõtet, ehk antud kirjeid tuleks ka lugemisoperatsiooni puhul kasutada, mis tõttu seda tihti kasutatakse koos eelneva peatüki strateegiaga.

6.2.4 Refresh Ahead

See on ennetav ehk eellaaditud andmetega vahemälu strateegia, mis laeb või värskendab andmed vahemälus ennem, kui neid küsitakse. Siinjuhul vahemälu lahendus küsib kõigepealt andmed andmeallikalt ning salvestab vahemälus ja seejärel kliendi päringu korral saab kohe vastata vahemälus oleva informatsiooniga [6]. Joonis 5 annab ülevaate.



Joonis 5. Andmete laadimise strateegia refresh ahead

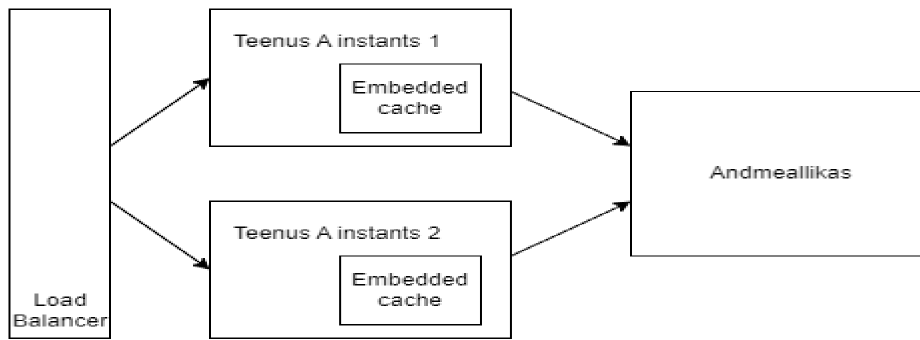
Eellaaditud andmete puhul on andmed vahemälus saadaval ennem, kui neid vaja on ning seeläbi tagavad alati kiire jõudluse ahela kriitilisel hetkel [7].

6.3 Vahemälu asukoht

Üks olulisi tegureid on vahemälu asukoht süsteemis. On olemas mitmed võimalusi kus vahemälu komponenti rakendada, igaühel neist on omad eelised ning puudused.

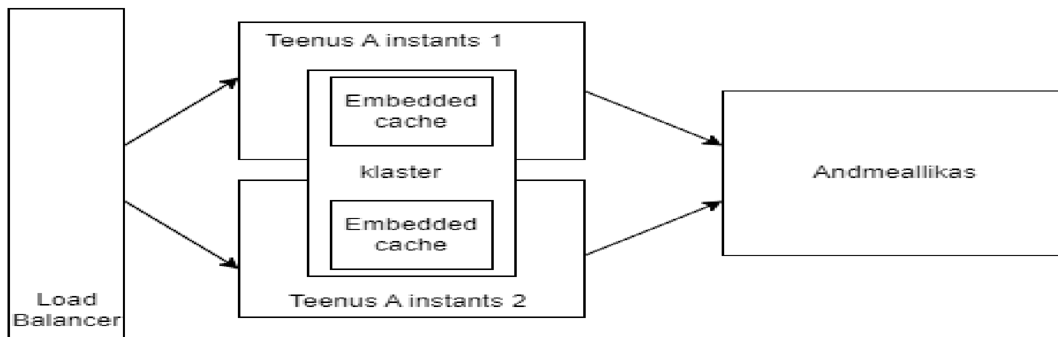
6.3.1 Embedded Cache

Tegemist on ühe lihtsama asukohaga vahemälule, kus vahemälu asub teenuse instantsi sees ning andmed laetakse otse teenuse mällu. Päringu teenindamisel otsitakse kõigepealt vahemälust ning seejärel juhul, kui vahemälust ei leitud, küsitakse andmeallikalt ning lisatakse kirje teenuse privaatsesse vahemällu [2]. Joonis 6 annab ülevaate.



Joonis 6. Teenuse sees asuv privaatne vahemälu

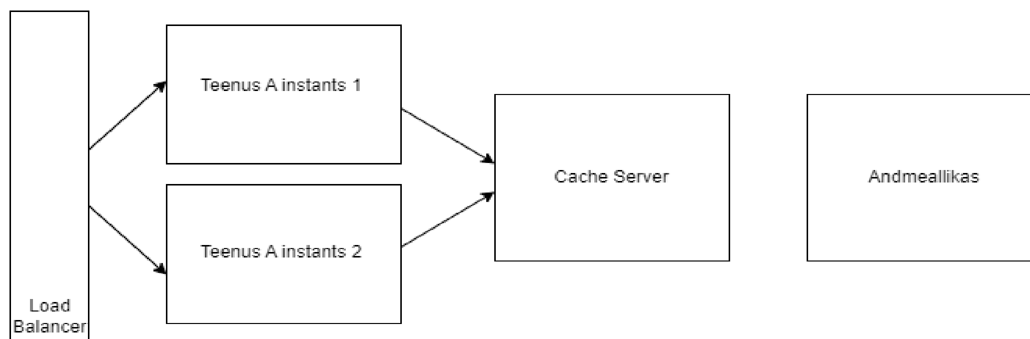
Antud lahendust on lihtne rakendada, *Spring* puhul piisab selleks eri annotatsioonist meetodile, mis andmeallikalt andmeid küsib [8]. Suurim probleem antud lahenduse juures on olukord, kui meil on *N* teenuse instantsi oma privaatse vahemäluga ning süsteemi tuleb päring, mis korra on juba teenindatud siis tõenäosus, et vahemälus kirje olemas on *1/N* [9]. Selle probleemi lahendamiseks on võimalik kasutada ka jagatud sissehitatud vahemälu, mille tulemusel vahemälud rakenduste instantsides moodustavad ühise jagatud vahemälu kobara. Selle jaoks on tehnoloogiad mille kasutamine *Spring* raamistikul on küllaltki lihte [8]. Küll aga vahemälu on endiselt teenuse osa, jagab mikroteenusega ressursse ja rakenduse taas käivitusel andmed kaovad. Joonis 7 annab ülevaate.



Joonis 7. Teenuse sees asuv jagatud vahemälu kobar süsteem

6.3.2 Client-Server Cache

Printsiip siin on murede eraldamine ehk teenuse ja vahemälu üksteisest eraldamine [10]. Vahemälu on süsteemi iseseisev eraldi komponent, kus kõik lugemise ja kirjutamise operatsioonid teenustelt toimuvad üle võrgu päringuga kasutades vahemälu klienti. Joonis 8 annab ülevaate.

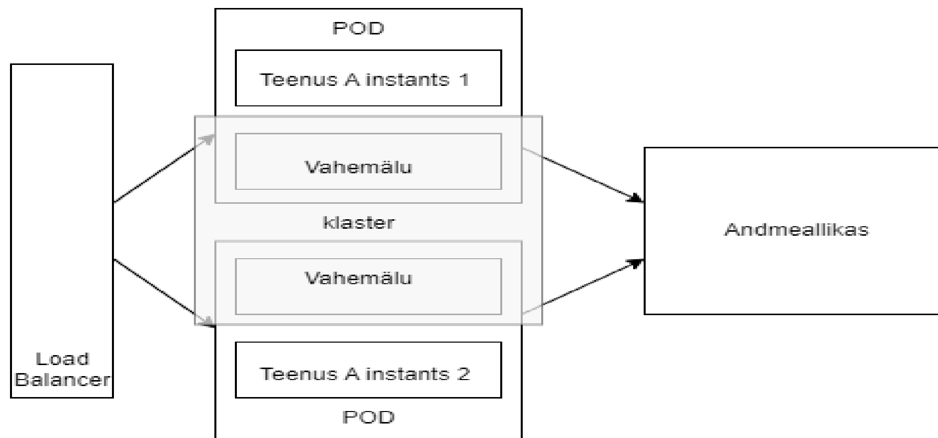


Joonis 8. Eraldiseisev vahemälu server

Eraldiseisvus tagab vahemälu eraldiseisva konfigureerimise ehk saab eraldi seisvalt mälu või instantse juurde lisada. Samuti teenused, mis vahemälu kasutavad on uuendatavad vahemälu lahendusest eraldi seisvalt, mis tähendab, et ressursside jaotus on individuaalne ja teenuse taas käivitus ei mõjuta andmeid vahemälus [8].

6.3.3 Sidecar Cache

Tegemist on konteiner platvormide nagu *Kubernetese* spetsiifilise lahendusega, kus rakendus ühikut kutsutakse *pod* mille sees on tavaliselt üks konteiner rakendusega, kuid sinna on võimalik ka kaasa lisada teisi lisa funktsioonidega konteinereid. Sel juhul nii rakendus, kui ka vahemälu asuvad alati samas füüsilises masinas [8]. Joonis 9 annab ülevaate.



Joonis 9. Rakendusega samas konteiner platvormi rakendus üksuses asuv vahemälu

See lahendus on segu kahest eelnevast, kus on sarnasusi nii eraldiseisva kui ka teenuse instantsi sees olevate lahendustega. Teenuse instantsi sees oleva vahemäluga sarnaneb see lahendus, kuna asub samas loogilises ruumis tänu millele on ühendusel rakenduse ja vahemälu vahel madal latentsus ning samuti jagatakse ressursse. Eraldi seisva vahemäluga serveriga sarnaneb see kuna kirjutamiseks ja lugemiseks kasutatakse vahemälu klienti [8].

6.4 Vahemälu kehtetuks tunnistamine

Mingi aja möödudes andmed alati muutuvad ning vältimatult hakkavad allika tõesusest eemale triivima. Kui on toimunud muudatus andmetega algallikas, aga vahemälu endiselt hoidis eelmist väärtust siis see võib viia tahtmatute vigadeni rakenduse töös. Kehtetuks tunnistamine tähendab andmete vahemälust eemaldamist. Üks enimlevinud viise on kirjele aegumise määramine, pärast sätestatud aja möödumist kirje automaatselt eemaldatakse mälust. Järgmise päringu puhul samale kirjele, see küsitakse uuesti andmeallikast ning salvestatakse mällu. Aegumis-aja sättimine peab olema hästi läbi mõeldud vastavalt äri vajadustele ja andmete muutumise sagedusele. Kui liiga pikaks aeg sätestada võib juhtuda, et tagastatakse vananenud andmeid, kui liiga lühikeseks aeg sätestada, tehakse liigseid päringuid andmeallikale [3].

6.5 Vahemälu tühjendamine

Vahemälu tühjendamise reeglid on sätestavad, mis peab juhtuma, et vahemälu suurus ei ületaks mälu limiiti. Selle saavutamiseks hakatakse mälus elemente kustutama vastavalt

defineeritud reeglistikule. On mitmeid erinevaid algoritme mille alusel elemente kustutada. Erinevad lahendused võivad toetada erinevaid algoritme. Üks enim levinumaid lahendusi on kustutada kirjed, mida viimati kasutati kõige rohkem aega tagasi ning teine levinud lahendus on kustutada kirjed, mida kõige harvem kasutatakse[3].

7 Vahemälu tehnoloogiad

On mitmeid erinevaid populaarseid vahemälu tehnoloogiad, mida mikroteenuste juures kasutatakse. Kolm kõige levinumat on Redis, Memcached ja Hazelcast. Need kõik on operatiivmälus andmeid hoidvad lahendused, millel on oma tugevused ja nõrkused.

7.1 Redis

Redis on üks kõige levinumaid vabavaraline *NoSQL* võti väärtus paaride vahemällu salvestamise tehnoloogiad. Enamasti kasutatakse *Redist* mõne andmeallika nagu andmebaasi või muu rakenduse rakendusliidese ees, et aidata parandada rakenduse jõudlust. *Redis* on disainitud, et pakkuda rakendustele paremat jõudlust tihedalt kasutatud andmete küsimisel [12]. Redis toetab ainult *Cache Aside* strateegiat [15].

7.1.1 Rakendus mudelid

Redis toetab nii ühe instantsi kui ka mitme instantsi mudeleid. Ühe instantsi mudelit on kõige lihtsam rakendada, see aitab kiirelt teenuse jõudlust ja kiirust kasvatada. Ühe instantsi rakendamisel on ka mitmed puudused, näiteks kui instantsi tabab rike ning see ei ole saadaval, siis kõik päringud andmeid vahemälust saada ebaõnnestuvad, mis kahjustab süsteemi üldist jõudlust ja kiirust. Lisaks on ühe instantsi mudel ainult vertikaalselt skaleeritav, mis tähendab, et kui on vaja mahtu kasvatada siis seda saab teha ainult instantsile mälu juurde lisades. Kui piisavalt mälu anda, siis teatud lihtsamatel juhtudel on selline lahendus väga võimas [14].

Teine levinud seadistus on kõrge kättesaadavuse rakendus, kus rakendatakse teist sekundaarset instantsi. Sekundaarseid instantse saab olla rohkem kui üks. Kirjutamise

operatsioonil põhi instantsi jagab see käskude koopiad sekundaarsetele instantsidele laiali. Sekundaarseid instantse saab kasutada lugemis operatsioonidel koormuse jaotamisel ja põhi instantsi rikkel saab sekundaarne rolli üle võtta [14]. Kõrge kättesaadavuse rakendamiseks on olema *Redis Sentinel*, mis on disainitud instantside haldamise lihtsustamiseks. *Redis Sentinel* jälgib pidevalt instantside kättesaadavust, saadab teateid juhtumitest, tegeleb rikkejuhtimisega ja pakub teenuse avastamist [14]. Täpsem seletus järgnevalt:

- Jälgimine – Jälgib, et instantsid töötaksid ettenähtult [16].
- Teated – Teavitab *API* kaudu teisi rakendusi ja süsteemi administraatoreid juhtumitest, kui mõne jälgitava instantsiga on midagi valesti [16].
- Rikkejuhtimine – Kui primaarne instants ei ole kättesaadav ja piisav hulk *Sentinel* protsessesse seda tõeseks peavad siis algatatakse tõrkesiirdeprotsess, kus üks sekundaarne sätestatakse primaarseks ning ülejäänud sekundaarsed sätestatakse uue primaarse vastu [16].
- Konfiguratsioonihaldus – Teenuse avastamise meede klientidele avastamiseks milline instants hetkel primaarne on [16].

Soovitatav on *Redis Sentineli* kasutada paaritu arvu instantsidena, mis tagab, et rikkejuhtimise hääletusel ei tekiks viigi seisu. Kõrge kättesaadavuse ja *Redis Sentineli* rakendamisel saab siiski ainult vertikaalselt kasvada, ehk ei toimu andmete eraldamist instantside vahel ning ainus viis kasvada on lisada serveritele ressursse juurde [14].

Kolmas viis *Redis*e rakendamiseks on kobar süsteem, kus on mitmed primaarsed instantsid ning igal primaarsel on ka sekundaarsed instantsid. *Redis* kobar süsteemi rakendamise puhul jaotatakse andmed mitmete primaarsete instantside vahel automaatselt laiali. Seda protsessi kutsutakse andmete killustamiseks ehk igal primaarsel instantsil on osa andmeid. Selline rakendamine tagab horisontaalse mahu suurendamise võimekuse, kus instantside võimsuse suurendamise asemel saab lisada juurde uusi instantse. Andmete killustamiseks kasutab redis algoritmilist killustamist, kus olenemata sisend võtme suurusest konverteeritakse see alati sama suurusega räsiks ning seejärel rakendatakse *MOD* funktsiooni, kus *MOD* väärtus on primaarsete instantside arv. Seeläbi saavutatakse alati sama tulemus, ning sisend võti on alati samas

instantsis. Instantside juurde lisamisel andmete ümber jagamise lihtsustamiseks instantside vahel kasutab *Redis* lahendust *hashslot.* Siin sisend võtmed kaardistatakse *hashslot*'de vastu, mis võimaldab instantside juurde lisamisel kõikide andmete ükshaaval ümber liigutamise asemel liigutada *hashslot*'id. Redis kasutab 16384 *hashslot*'i ning sisend võtmete jagamiseks instantside vahel jagatakse kõigepealt *hashslot*'id instantside arvuga, mille tulemusel saavutatakse vahemikud, millisesse instantsi mis *hashslot* kuulub. Seejärel otsustamaks, kuhu konkreetne sisend kaardistada, kasutatakse sama eelevalt mainitud deterministlikku räsi funktsiooni sisendi peal ning seejärel *MOD* funktsiooni, kus *MOD* väärtus on *hashslot*'de koguarv [14].

Kõikide instantside jälgimiseks kasutab *Redis* kobar süsteem lahendust, kus kõik instantsid pidevalt suhtlevad omavahel, arusaamaks millised on saadaval ning millistel hetkel probleeme esineb. Kui piisaval hulgal instantse on nõus, et üks primaarne instants ei ole kättesaadav siis edendatakse vastava primaarse sekundaarne uueks primaarseks [14].

7.1.2 Andmete püsivus

Redis on operatiivmälus andmete hoidmise lahendus, ehk instantsi töö lõppemisel kaovad ka andmed. Paljudel kasutusjuhtumite jaoks ei ole andmete kaotamine kriitiline aga juhuks, kui on siis saab kasutada andmete kopeerimist vastupidavale andmekogule [17]. Redis pakub järgenvaid lahendusi:

- *Redis* andmebaas – Siin tehakse teatud eelnevalt sätestatud ajavahemiku järel andmetest läbilõige ning salvestatakse. Suurim miinus on võimalik andmete kadu kahe ajas tehtud läbilõike vahel [17].

- Faili lisamine – Siin logitakse iga kirjutamise operatsiooni käsklus faili. Need käsklused saab uuesti tööle panna, et taastada andmekogu [17].

- Mitte püsivus – Siin juhul andmete püsivust ei kasutata. Levinud vahemälu kasutusjuhtudel [17].

- *Redis* andmebaas ja faili lisamine – Kahe kombineeritud meetod [17].

7.2 Memcached

Memcached on vabavaraline võti väärtus paaride operatiivmälus hoidmise tehnoloogia, mida on kerge kasutada ning mis on kõrge jõudlusega. Selle tehnoloogia puhul on kliendil palju suurem vastutus, kuna *Memcached backend* on ainult võti väärtus paaride andmekogu mille vastutada on kirjutamise ja lugemise operatsioonid ning millal mälu uuesti kasutada või millal mälust kirje välja visata [18].

7.2.1 Rakendus mudelid

Memcached toetab samuti nii ühe, kui mitme instantsi rakendus mudeleid. Ainult, et *Memcached* puhul ei ole serverid üksteisest teadlikud ning rakenduse klient peab rakendama loogikat, et saavutada mitme instantsi rakendus mudeleid [18]. Seega *Memcached* rakendus mudel on osaliselt rakendatud kliendi ja osaliselt vahemälu serveri poolel. Kliendi kompetentsi kuulub serveri valik, kuhu sisend võti kuulub ja mida teha, kui server pole kättesaadav. Serveri kompetentsi kuulub teadmine, kuidas lugeda kirjeid, kuidas kirjutada ning kuidas kirjeid kustutada, kui mälu on juurde vaja[19]. Tüüpiline Memcached rakendamine sisaldab:

- Klient tarkvara, millele antakse list instantsidest [19].
- Kliendi poolne räsi algoritm, mis valib serverit vastavale sisendile [19].
- Serveri tarkvara mis salvestab võti väärtus paare serveri enda räsi tabelisse[19].
- Serveri algoritmid, mis otsustavad millal ja kuidas mälu vabastada [19].

Seega siin ühe või mitme instantsi rakendamine vahemälu serveri poolt ei erine. Kogu erinevus ühe või mitme instantsi rakendamisel on kliendi tarkvara poolel.

Ühe instantsi rakendamine on küllaltki lihtne ning annab kohesest tulemust teatud määrani. Kuna Memcached on mitmelõimelise arhitektuuriga siis arvutusvõimsuse vertikaalne kasvamine on efektiivne [20].

Mitme instantsi rakendamine serveri poolelt tähendab lihtsalt mitmel korra ühe instantsi rakendamist. Instantsid ei ole teadlikud üksteisest ja töötavad eraldi seisvalt [19]. Seega kõrge kättesaadavuse saavutamiseks peab andmete replikatsioon sekundaarsetele

instantsidele ja instantside kättesaadavuse kontroll ning ümberlülitamine sekundaarsele olema kliendi rakenduse poolel [21].

Kobar süsteemi rakendamisel on samuti kogu loogika kliendi poolel. Lisatakse rohkem, kui üks instants ning iga klient rakendus peab kasutama sama algoritmi võtmete paigutamiseks või lugemiseks instantside vahel, soovitatav algoritm on *Consistent Hashing* [19]. Kuna see tagab võimalikul väikse arvu võtmeid, mis tuleb instantside vahel ümber jagada, kui instantse juurde lisatakse või eemaldatakse [22]. Klient peab teadma instantside kogu arvu ning nende aadresse. Sellise rakendamise puhul on kõik andmed instantsides ühtse hajutatud räsi tabeline [19].

7.2.2 Andmete püsivus

Memcached ei paku andmete püsivuse strateegiat. *Memcached* rakendab vaikimisi andmete kustutamist alati sama kõige harvem kasutatud algoritmi alusel. Mis ei ole ideaalne igas olukorras. Võib juhtuda olukord, kus kirjele on lisatud aegumine ühe tunni pärast, kuid ei ole võimalik kindel olla, et antud kirje seal ka tunni jooksul olemas on [23].

7.3 Hazelcast

Hazelcast on vabavaraline hajus operatiivmälus võti väärtus paaride salvestamise tehnoloogia. Hazelcast toetab mitmeid salvestamise strateegiaid nagu *Cache Aside* ja ka *Read and Write Through.* Lisaks toetab *Hazelcast* väga hästi kasvamise viise ja pakub instantside automaatset avastamist ning automaatset andmete sünkronisatsiooni [24].

7.3.1 Rakendus mudelid

Hazelcast toetab nii ühe kui ka mitme instantsi rakendus mudeleid ning seda nii *Embedded cache* kui ka *Client-Server* asukoha strateegiana. *Embedded cache* puhul iga instants sisaldab nii rakendust, andmeid vahemälus kui ka *Hazelcast* teenuseid. *Client-Server* puhul on rakendantud vastutuste eraldamist ning vahemälu andmed ja *Hazelcast* teenused on eraldatud rakenduse enda loogikast ning ressursid on eraldi juhitavad [25].

Hazelcast jagab kogu mälu ruumimälu 271 segmendiks, kus iga segment võib omada tuhandeid kirjeid olenevalt süsteemi ressursside võimekusest. Igal mälu segmendil võib olla mitu koopiat, mis on jagatud ülejäänud instantside vahel laiali. Üks koopiatest on

primaarne ning ülejäänud on varukoopiad. Instantsi kuhu kuulub primaarne segment, kutsutakse segmendi omanikuks. Ühe instantsi rakendamisel kuuluvad kõik 271 segmenti sellele ühele instantsile, seahulgas varukoopiad [25]. Ehk sellise seadistuse puhul ei ole kõrge kättesaadavus rakendatud.

Järgmiste instantside lisamisel või eemaldamise jagatakse mälu segmendid nende vahel võrdselt laiali ning sama tehakse ka varukoopiatega. Näiteks kui rakendatakse kahte instantsi, siis pooled mälu segmendid kuuluvad ühele ning pooled teisele. Varukoopiad jagatakse vastupidiselt ehk esimene instants hoiab teise instantsi varukoopiaid. Instantside juurde lisamisel jagatakse mälu segmendid ümber ning *Hazelcast* jagab primaarsed ja varukoopia mälu segmendid uute instantside vahel laiali. Siin kasutatakse ka *Consistent Hashing* algoritmi, mis tagab minimaalse arvu mälu segmente mida tuleb horisontaalse kasvamise puhul liigutada [25].

7.3.2 Andmete püsivus

Hazelcast toetab tasulises versioonis püsimällu andmete kirjutamist vahemälust ning taastamise operatsioone nii instantsile kui tervele kobar süsteemile [26]. Mälu täissaamisel on vaikeväärtuseks mitte kustutamine, kuigi on võimalk sätestada ka algoritme, mis kas kõige harvem kasutatud või kõige rohkem aega tagasi kasutatud algoritmide alusel hakkavad kirjeid kustutama [27].

8 Vahemälu lahenduse valik ja kirjeldus

Selles peatükis autor valib eelnevalt analüüsitud salvestamise strateegia, vahemälu asukoha ja tehnoloogia vastavalt kirjeldatud probleemile ning nõuetele. Lisaks kirjeldatakse plaanitavat vahemälu terviklahendust ning ka plaanitavat kasutust probleemide lahendamisel.

8.1 Andmete laadimise strateegia valik

Selles ökosüsteemis, kus antud töös kajastatav mikroteenuste kiht asub, puudub võimekus iga turu kõiki aktiivseid *SIM* kaarte ning vajalikku infot nende kohta

eellaadida ning selleks puudub ka vajadus, kuna aktiivseid *SIM* kaarte turgudel on täna veel oluliselt rohkem, kui mobiilseadmeid, mis on võimelised päringuid mikroteenus kihti saatma. Sellest tulenevalt on ainult laisklaaditud lahendused kaalumisel.

Antud töös on kajastatud laisklaaditud lähenemisviise, kus ühel juhul lähtesüsteemi integratsiooniga tegeleb vahemälu lahendus ja teisel juhul teenus ise. Antud mikroteenuste kihi mikroteenuste eesmärk on tegeleda integratsiooni keerukuse peitmisega. Turge ja süsteeme on küllaltki suur hulk seega antud vastutus peaks jääma integratsioon mikroteenustele. Lisaks võib mikroteenusel olla sama lähtesüsteemi vastu muid funktsioone, mis ei vaja vahemälu kasutust ja osade lähtesüsteemide andmemudel võib sisaldada liigset infot, mis vähemällu salvestusel kasutaks liigselt mälu. Vahemälus peaks olema miinimum kogus infot, et funktsiooni tagamine võimalik oleks. Seega üsna palju loogikat on vaja endiselt hoida mikroteenuse poolel. Rakenduse orkestreeritavaid laadimise strateegiaid on ainult üks, mis on *Cache Aside*. See tagab ka võimekuse kiiresti laieneda sama lahenduse kasutamisel teiste turgude teenustelele ning võimekuse antud lahendust kasutada muudel vajadustel, mis on ühtlasi ka üks nõuetest.

8.2 Vahemälu asukoha valik

Üks nõuetest antud vahemälu lahendusele on selle kasutamise võimekus terves mikroteenuste kihis. Üks variantidest oleks kasutada teenuste sissehitatud lahendust igas teenuses, mis vahemälu vajab. Selle rakendamine teenuses on küllaltki lihtne. Kuid andmete maht võib küllaltki suur olla, ning kasvamise vajadused erinevad rakenduse ja vahemälu vahel. Sissehitatud lahenduse puhul ei ole eraldi kasvamine võimalik. Lisaks võib olla vajadus kasutada vahemälu kahe erineva teenuse vahel, mis juhul peaks rakendama uusi rakendusliideseid teenustes, mis serveerivad enda talletatud infot teisele teenusele või vastupidi. Antud mikroteenuste kihile universaalse lahenduseana sobib kõige paremini *Client-Server* lahendus, kus vahemälu on eraldiseisev komponent olemas olevas arhitektuuris. Selline lähenemine tagab ka kiire kasvamise võimekuse uutel turgudel, kasutades sama seadistust.

8.3 Tehnoloogia valik

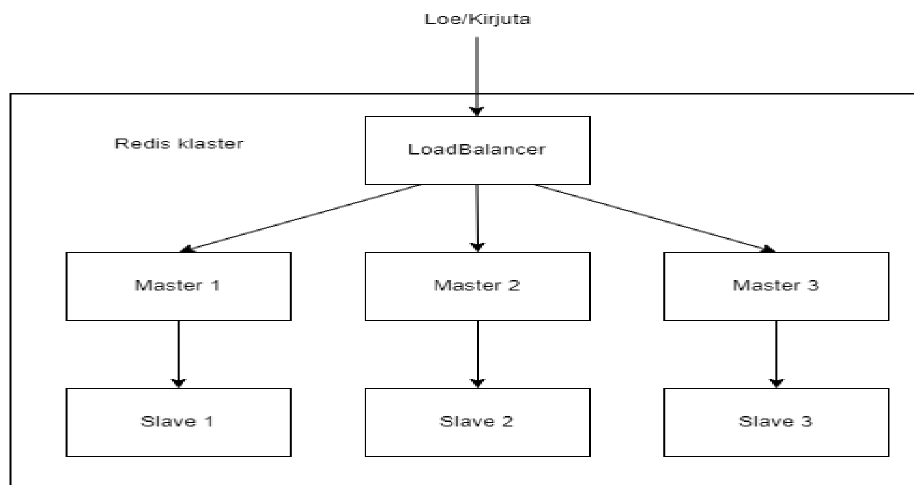
Kõik antud valikus olevat tehnoloogiad võimaldavad rakendada *Cache Aside* strateegiat koos *Client Server* eraldiseisva vahemäluna. Kõik need tehnoloogiad on levinud ning

tagavad sarnase tulemuse. Kasvamise mudel on kõige piiratum *Memcached* puhul, kus on suurem vastutus klient rakenduse koodil, lisaks *Memcached* rakendab ka automaatset andmete välja viskamise loogikat mälu täitumisel, mis ei ole antud lahenduse puhul kasulik. Edasi *Redis* ja *Hazelcast* mõlemad on horisontaalselt kasvavad ning samuti mõlemad toetavad erinevaid andmete väljatõstmise lahendusi kaasa arvatud, et andmeid ei visata välja ja tagastatakse viga. *Redis* vajab rohkem seadistust, et hajus lahendus püsti panna, kui Hazelcast on automaatselt hajus.

Redis toetab kõiki kasutusjuhtumeid ning omab väga head dokumentatsiooni. Lisaks Redis on *db-engines* järjestuses ülekaalukalt number üks võti väärtus paaride tehnoloogia [28]. Kuigi *Hazelcast* tundub esmapilgul väga hea lahendusena siis *Redis* tänu oma populaarsusele ja lisaks heale dokumentatsioonile omab ka laialdast kajastust erinevates kolmandate osapoolte artiklites, mis teeb rakendamise lihtsamaks. Redis on tehnoloogia millega edasi liigutakse ning mille seadistust detailsemalt hiljem kajastatakse.

8.4 Vahemälu lahenduse kirjeldus

Antud probleeme proovitakse lahendada järgmise seadistusega, kus on eraldi seisev vahemälu komponent, mis kasutab *Redis* tehnoloogiat. Täpsemalt proovitakse rakendada *Redis* kobar süsteemi, mis tagab nii süsteemi skaleeritavuse, kui ka kõrge kättesaadavuse. Sellisel juhul saab kerge vaevaga lisada uusi klient mikroteenuseid antud vahemälu lahendusele ning ressursside puudu jäämisel saab lisada instantse juurde ehk kasvada horisontaalselt. *Redis* kobar süsteem tagab ka vajaliku kõrge kättesaadavuse, kuna igal primaarsel instantsil on alati olemas sekundaarne, mis automaatselt primaarseks edutatakse vea puhul. Joonis 10 annab ülevaate.

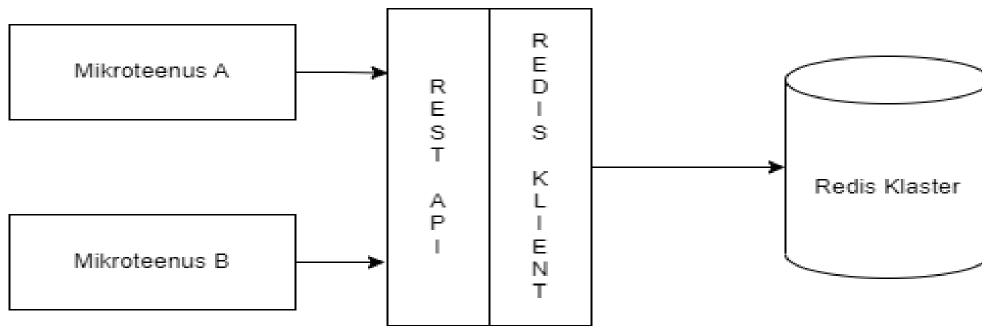


Joonis 10. Redis kobar süsteem

Siin rakendatakse Redis kobar süsteemi miinimum seadistust, mis koosneb kolmest primaarsest ehk *master* instantsist ja kolmest sekundaarsest ehk *slave* instantsist.

8.5 Vahemälu lahendus eraldi seisva teenusena

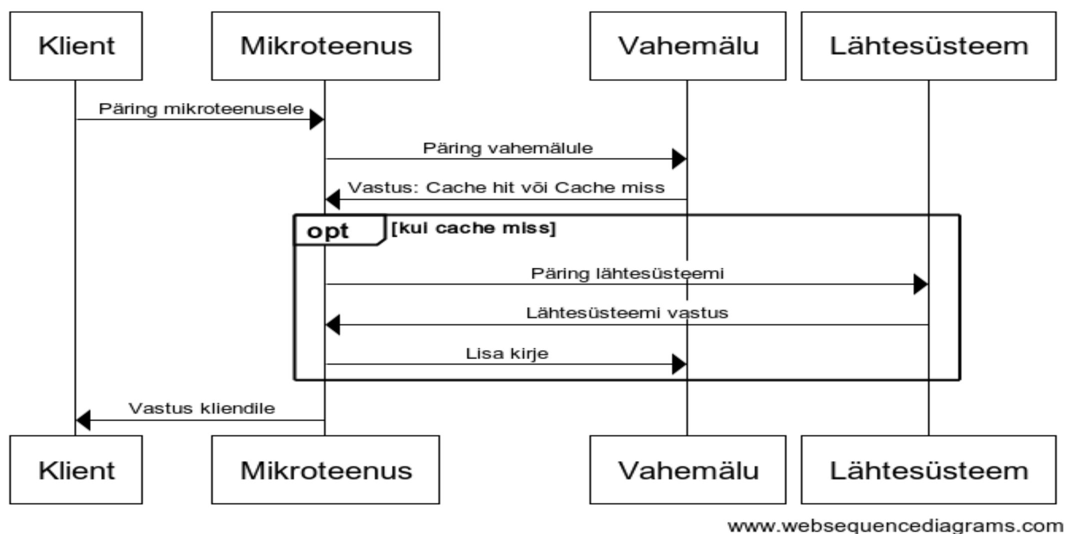
Antud mikroteenuste kihis igal mikroteenusel on geneeriline *REST API* klient. Vältimaks *Redis* kliendi rakendamist igas teenuses, kus vahemälu kasutada vaja on ning lisaks, et pakkuda nõudel eemaldamise funktsionaalsust saab rakendada eraldi mikroteenuse, mis sisaldab *Redis* klienti ning pakub *REST API* kontrollereid teistele mikroteenustele. Joonis 11 annab ülevaate.



Joonis 11. API kontroller vahemälu lahendusele

8.6 Vahemälu rakendamine koormus probleemi korral

Esimene probleem oli liigne koormus lähtesüsteemidele ning eesmärk vähendada koormust vahemälu lahenduse kasutamise abil. Joonis 12 annab ülevaate tegevuste järjestusest.

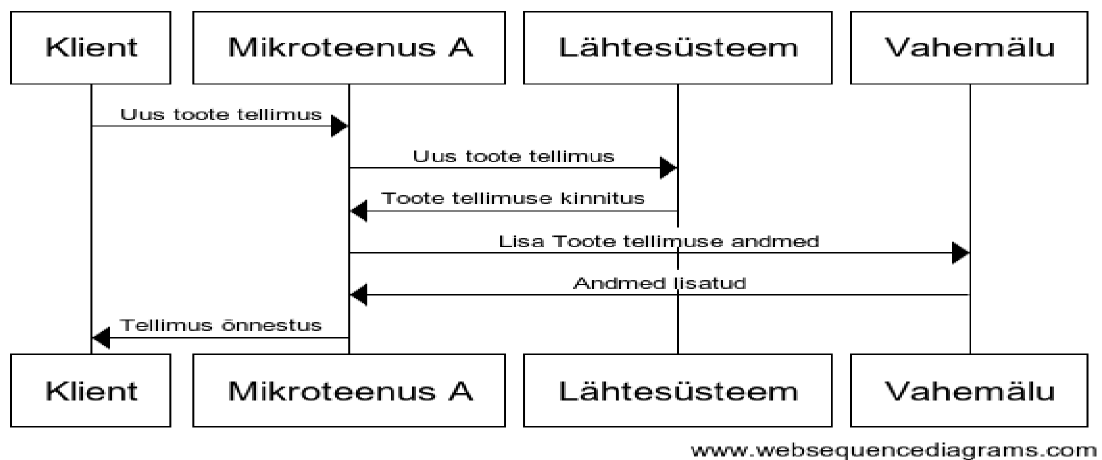


Joonis 12. Vahemälu kasutamine koormus probleemi korral

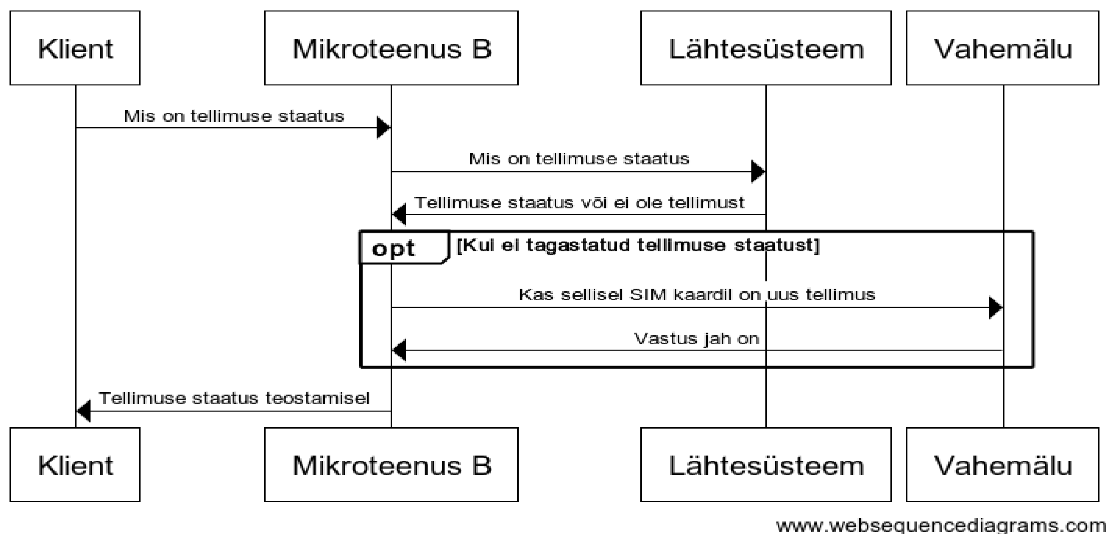
Sellel järjestus diagrammil on selgelt näha, kuidas antud probleemi puhul rakendatakse kõrval seisvat vahemälu lahendust laisklaaditult.

8.7 Vahemälu rakendamine andmete varajasel kättesaamisel

Teine probleem seisnes vajaduses teiste väliste süsteemide sünkronisatsiooni probleemide korral andmed iseseisvalt varem kätte saadavaks teha. Selleks saab rakendada sama vahemälu lahendust viisil, kus üks mikroteenus lisab andmed hetkel, kui need on teada vahemällu ning teine mikroteenus päringu saamisel olukorras, kus informatsioon tellimuse kohta ei olnud lähtesüsteemist kättesaadav, saab kasutada ajutiselt andmeid vahemälust ning kliendile vastata. Joonis 13 kirjeldab info vahemällu lisamist ning Joonis 14 kirjeldab sama informatsiooni lugemist.



Joonis 13. Õnnestunud toote tellimusel lisatakse tellimuse andmed vahemällu.



Joonis 14. Toote tellimuse staatusele vastav mikroteenus saab toote tellimuse kohta küsida vahemälust.

9 Lahenduse Arendus

Mikroteenuste kiht on kasutusel ettevõtte poolt juhitud Kubernetese konteiner teenuse platvormil. Seega nii *Redis* kobar süsteem, kui ka mikroteenus, mis teistele teenustele vahemälu teenust üle *API* pakub tuleb rakendada samal *Kubernetese* platvormil. Mikroteenused on kõik *Spring* raamistikul, seega ka *Redis* klienti ja *REST API* pakkuv mikroteenus kasutab sama tehnoloogiat. Mikroteenuse osas kajastatakse *Redis* kliendi seadistamist ja kirjeldatakse kontrollereid, mikroteenuse platvormil rakendamise kirjeldamine ei kuulu antud töö juurde. *Redis* kobar süsteemi puhul kajastatakse selle rakendamist *Kubernetese* platvormil.

9.1 Redis kobar süsteemi rakendamine Kuberneteses

Tulenevalt platvormist, kus *Redist* rakendatakse on vaja kohalikku konteinerpiltide hoidlasse *Redis* konteiner pilt laadida, mida instantside loomisel kasutatakse. *Redis* instantside rakendamiseks *Kubernetese* kobar süsteemis on vaja luua konfiguratsiooni faile *yaml* formaadis. Mille rakendamisel *Kubernetes* automaatselt sätestab vaja minevad ressursid ja *Redis* instantsid. Hiljem on vaja käsureal luua *Redis* instantsidest kobar süsteem ning luua *master slave* suhted [29].

9.1.1 Redis konteiner pilt

Rakenduste konteiner pilte saab alla laadida maailma suurimast konteinerpiltide hoidlast *Docker Hub* [30]. Järgnevate käsklustega saab avalikust konteiner hoidlast alla laadida viimase *Redis alpine* versiooni, mis on rajatud *Alpine Linux* versioonil. *Alpine Linux* on oluliselt väiksema suurusega, kui enamus muud baas konteiner pildid [31]. Ning sama versiooni kohalikku hoidlasse üle laadida. Joonis 15 näitab kasutatavaid käsklusi.

```
docker pull redis:alpine
docker push docker-local.jfrog.company.io/es/folder/redis:alpine
```

Joonis 15. Redis konteiner pildi kohalikku hoidlasse laadimine

Seejärel on meil olemas *Redise* konteiner pilt, mida saab kasutada *Redis* instantside loomisel antud *Kubernetese* platvormil.

9.1.2 Kubernetes Configmap

Kubernetese Configmap on *Kubernetese* kobar süsteemis olev avalik võti väärtus paaride hoidla. *Pod'i*de loomisel kasutatakse konfiguratsioone, keskkonna põhiseid muutujaid või käsklusi vastavast *Configmap'*st. See tagab keskkonna põhise konfiguratsiooni lahti sidumise rakendusest [32].

Redise Configmap'i on meil vaja lisada *Redise* konfiguratsioon ning lisaks käsklus, mis dünaamiliselt uuendab *IP* (*Internet Protocol*) aadressid *Redis* instantside konfiguratsiooni failis hetkel loodava instantsi aadressiga [29]. Joonis 16 kajastab faili sisu.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: common-cache-config
data:
  update-node.sh: |
    #!/bin/sh
    REDIS_NODES="/data/nodes.conf"
    sed -i -e "/myself/ s/[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-
9]\{1,3\}\.[0-          9]\{1,3\}/${POD_IP}/" ${REDIS_NODES}
    exec "$@"
  redis.conf: |+
    cluster-enabled yes
    cluster-require-full-coverage no
    cluster-node-timeout 15000
    cluster-config-file /data/nodes.conf
    cluster-migration-barrier 1
    appendonly yes
    protected-mode no
```

Joonis 16. Kubernetese configmap Redise jaoks

Configmap'i esimene andme kirje on *bash* käsklus, mis lisab loodava instantsi andmed konfiguratsiooni faili ning teine on *Redise* konfiguratsioon, mis sätestab loodava instantsi seadistuse. Antud seadistusega luuakse *Redise* instants, millel on kobar süsteemi toetus sees ning millel on andmete püsivus tagatud käskluste faili kirjutamisega [35]. Selle *Configmap'i* lisame tekstina *yaml* faili, kuhu hiljem lisame teised failid ning seejärel rakendame kõik korraga *Kubernetese* kobar süsteemis.

9.1.3 Kubernetes Statefulset

Statefulset on *Kubernetese* objekt mis juhib olekupõhiseid rakendusi. Sarnaselt *Kubernetese* tavalisele rakenduse rakendamisele, juhib *Statefulset* *Pod*'e, mis baseeruvad samal spetsifikatsioonil, kuid siinjuhul luuakse igale *Pod*'le püsiv identifikaator, mida säilitatakse alati ehk iga *Pod* on siin sama seadistuse põhine, kuid ei ole äravahetamiseni sarnane teisega [33]. Lisa 2 kajastab faili sisu.

Antud seadistus defineerib instantside arvu, milleks on kuus. Siin viidatakse eelnevalt konteinerpiltide hoidlasse lisatud *Redise* rakendusele, mille baasil luuakse iga *Redis* instants konteiner. Defineeritakse konteineri pordid, kasutamiseks klientidele ning samuti, mis pordi kaudu *Redise* instantsid omavahel suhtlevad tagamaks kobar süsteemi töö. Samuti defineeritakse andmeruumi ligipääsud *Pod*'idele, mis tagavad ligipääsu väljaspool *Pod*'i andmetele ja konfiguratsiooni failidele ning sätestatakse ka ressursid igale konteinerile, mis on piiritletud, kui palju ressursse on vaja konteineri loomisel minimaalselt ning mis on maksimum limiit, mida konteiner kasutada saab. Lisaks on defineeritud *volumeClaimTemplates*, mis pakub *Kubernetese* kobar süsteemis olevat püsivat mäluruumi kasutades *PersistentVolumes* [33]. *PersistentVolumes* on osa andmeruumist, mis on ennem loodud Kubernetese administraatori poolt või luuakse dünaamiliselt kasutades administraatori poolt ette defineeritud viisi. Seadistus siin kasutab ennem defineeritud andmeruumi klassi *nas*, mille põhjal luuakse andmeruum Kubernetese kobar süsteemis ning igale *Redis Pod*'le luuakse sidus kinnitus ühele andmeruumile ehk *PersistentVolumeClaim*, mille maht on 200GiB, mis on defineeritud miinimum *nas* klassis [34]. Selle konfiguratsiooni lisame samuti tekstina *yaml* faili.

9.1.4 Kubernetes Service

Service ehk teenus on Kuberneteses selleks, et abstraktsel viisil avalikustada võrgurakendus, mis töötab mitmel *Pod*'il. *Pod* ise ei ole püsiv ressurss. Teenus sätestab, kuidas võrguliiklus rakenduse *Pod*'de poole liigub. Klient ei pea teadma, milline *Pod* parasjagu antud päringut töötles. Järgnev teenuse konfiguratsioon avalikustab Redise instantsid Kubernetese kobar süsteemi sisemisel *IP* aadressil, mis tähendab, et see on ligipääsetav ainult sama kobar süsteemi seest [34]. Selle osa lisame ka olemasolevasse *yaml* faili. Joonis 17 kajastab antud faili sisu.

```
apiVersion: v1
kind: Service
metadata:
  name: common-cache
spec:
  type: ClusterIP
  ports:
    - port: 6379
      targetPort: 6379
      name: client
    - port: 16379
      targetPort: 16379
      name: gossip
  selector:
    app: common-cache-store-selector
```

Joonis 17. Kubernetes Service Redis seadistuse jaoks

9.1.5 Redis instantside kasutuselevõtt

Instantside kasutuselevõtuks, tuleb eelpool loodud konfiguratsioonid käivitada *Kubernetese* käsurea tööriistal (*kubectl*) või üles laadida *Rancher*'is kasutades graafilist liidet [29]. Järgnevalt käivitatakse, antud konfiguratsioonid käsurealt. Joonis 18 kuvab käsu eelnevalt loodud konfiguratsioonide rakendamiseks.

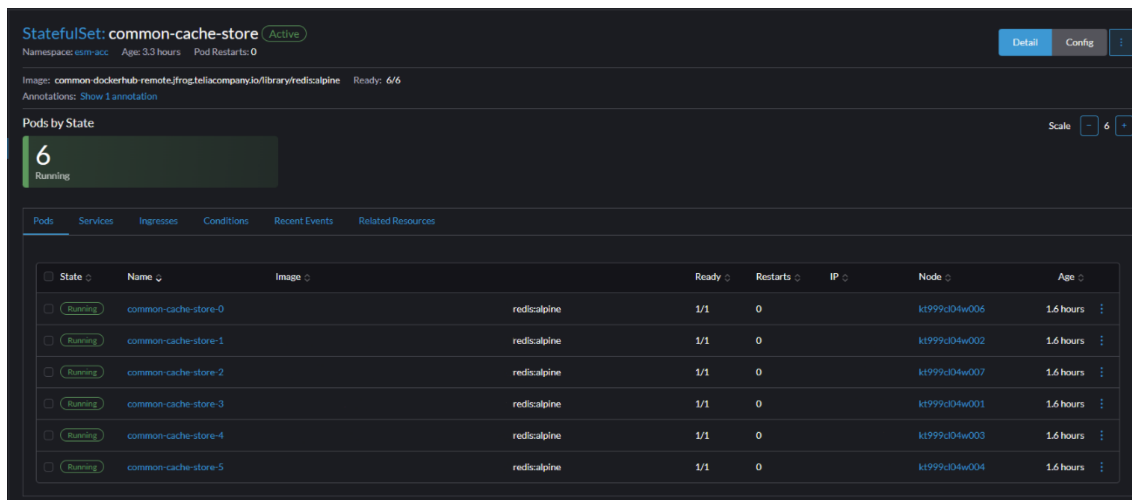
```
 C:\Users\siim.milli\.kube>kubectl apply -f redis-cluster-config.yml
configmap/common-cache-config created
service/common-cache created
statefulset.apps/common-cache-store created
```

Joonis 18. Redis instantside rakendamine Kubernetese käsureal

Sellega on nüüd olemas *Redis* instantsid. Seda saab kontrollida käsurealt või *Rancher* kasutaja liidese kaudu. Joonis 19 näitab saadud tulemust käsurealt ning Joonis 20 kasutajaliideses.

```
 C:\Users\siim.milli\.kube>kubectl get pods
NAME                            READY   STATUS    RESTARTS     AGE
common-cache-store-0            1/1     Running   0            83m
common-cache-store-1            1/1     Running   0            83m
common-cache-store-2            1/1     Running   0            83m
common-cache-store-3            1/1     Running   0            83m
common-cache-store-4            1/1     Running   0            84m
common-cache-store-5            1/1     Running   0            84m
```

Joonis 19. Redis instantsid Kuberneteses



Joonis 20. Redis instantsid Rancher kasutajaliideses

9.1.6 Redis kobar süsteemi moodustamine

Esmalt saame käsurealt mõne *Redis Pod*'i seest vaadata, mis on hetke seis *Redis* kobar süsteemil. Joonis 21 ja Joonis 22 kuvavad infot Redis kobar süsteemi kohta ennem seadistust.

```
C:\Users\siim.milli\.kube>kubectl exec -it common-cache-store-0 -- redis-cli
127.0.0.1:6379> cluster info
cluster_state:fail
cluster_slots_assigned:0
cluster_slots_ok:0
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:1
cluster_size:0
cluster_current_epoch:0
cluster_my_epoch:0
cluster_stats_messages_sent:0
cluster_stats_messages_received:0
total_cluster_links_buffer_limit_exceeded:0
```

Joonis 21. Redis kobar süsteemi info käskluse tulemus ennem

```
127.0.0.1:6379> cluster nodes
c4b310d7f3b40aaabb5d9d3e34ede10155d4d8c1 xxx.xx.x.xxx:6379@16379
myself,master - 0 1682947940432 0 connected
```

Joonis 22. Redis kobar süsteemi instantsid esimese instantsi vaatest

Kobar süsteemi rakendamiseks on vaja anda käsklus läbi mõne *Redis* instantsi koos hetke *Pod*'ide *IP* aadressidega. Kobar süsteemi loomine ja vastavate *IP* aadresside ette andmine peaks olema võimalik ka dünaamiliselt ühe käsklusega, kuid seda ei

õnnestunud tööle saada. Seega hetke *Pod*'ide *IP* aadressid tuli leida eraldi käsklusega ning seejärel staatiliselt kobar süsteemi loomis käsklusele kaasa anda [29]. Joonis 23 kuvab kobar süsteemi loomise käsku ning vastavat tulemust. Joonis 24 kuvab kobar süsteemi infot peale seadistust.

```
C:\Users\siim.milli\.kube>kubectl get pods -l app=common-cache-store-selector
-o jsonpath='{.items[*].status.podIP}'
'xxx.xx.x.xxx xxx.xx.x.xxx xxx.xx.x.xxx xxx.xx.x.xxx xxx.xx.x.xxx
xxx.xx.x.xxx'
C:\Users\siim.milli\.kube>kubectl exec -it common-cache-store-0 -- redis-cli
--cluster create --cluster-replicas 1 xxx.xx.x.xxx:6379 xxx.xx.x.xxx:6379
xxx.xx.x.xxx:6379 xxx.xx.x.xxx:6379 xxx.xx.x.xxx:6379 xxx.xx.x.xxx:6379
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica xxx.xx.x.xxx:6379 to xxx.xx.x.xxx:6379
Adding replica xxx.xx.x.xxx:6379 to xxx.xx.x.xxx:6379
Adding replica xxx.xx.x.xxx:6379 to xxx.xx.x.xxx:6379
Can I set the above configuration? (type 'yes' to accept): yes
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join
>>> Performing Cluster Check (using node xxx.xx.x.xxx:6379)
M: c4b310d7f3b40aaabb5d9d3e34ede10155d4d8c1 xxx.xx.x.xxx:6379
   slots:[0-5460] (5461 slots) master
   1 additional replica(s)
M: c87034faa532eaaca0cea672c15db30f617b8b18 xxx.xx.x.xxx:6379
   slots:[5461-10922] (5462 slots) master
   1 additional replica(s)
M: c4dba49515bebbe10c8d5524deea7b41a101e1d7 xxx.xx.x.xxx:6379
   slots:[10923-16383] (5461 slots) master
   1 additional replica(s)
S: 699a65ce8ba9fed80e1794c8098be317586b0a82 xxx.xx.x.xxx:6379
   slots: (0 slots) slave
   replicates c87034faa532eaaca0cea672c15db30f617b8b18
S: 57cfba4ebb6a298705b8a506d98a9b69838e6f48 xxx.xx.x.xxx:6379
   slots: (0 slots) slave
   replicates c4b310d7f3b40aaabb5d9d3e34ede10155d4d8c1
S: c360f8381cdb7d1d6a9689345dfdd3b0d4ed3588 xxx.xx.x.xxx:6379
   slots: (0 slots) slave
   replicates c4dba49515bebbe10c8d5524deea7b41a101e1d7
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

Joonis 23. Redis kobar süsteemi seadistuse käsklus ja vastus

```
C:\Users\siim.milli\.kube>kubectl exec -it common-cache-store-0 -- redis-cli
127.0.0.1:6379> cluster nodes
c87034faa532eaaca0cea672c15db30f617b8b18 xxx.xx.x.xxx:6379@16379 master - 0
1682961205000 2 connected 5461-10922
c4dba49515bebbe10c8d5524deea7b41a101e1d7 xxx.xx.x.xxx:6379@16379 master - 0
1682961205000 3 connected 10923-16383
c4b310d7f3b40aaabb5d9d3e34ede10155d4d8c1 xxx.xx.x.xxx:6379@16379
myself,master - 0 1682961204000 1 connected 0-5460
699a65ce8ba9fed80e1794c8098be317586b0a82 xxx.xx.x.xxx:6379@16379 slave
c87034faa532eaaca0cea672c15db30f617b8b18 0 1682961206642 2 connected
57cfba4ebb6a298705b8a506d98a9b69838e6f48 xxx.xx.x.xxx:6379@16379 slave
c4b310d7f3b40aaabb5d9d3e34ede10155d4d8c1 0 1682961205639 1 connected
c360f8381cdb7d1d6a9689345dfdd3b0d4ed3588 xxx.xx.x.xxx:6379@16379 slave
c4dba49515bebbe10c8d5524deea7b41a101e1d7 0 1682961204000 3 connected
```

Joonis 24. Redis kobar süsteemi info käsklus peale kobar süsteemi konfiguratsiooni

Nii kobar süsteemi loomise käskluse vastuse, kui järgneva kontroll käskluse järgi näeme, et nüüd on olemas kõrgelt kätte saadav *Redise* kobar süsteem. On olemas kolm master instantsi ja kolm slave instantsi. *Hashslot*'id on jaotatud kolme master instantsi vahel.

9.2 REST API mikroteenus Redis kliendiga

Järgnevalt kirjeldatakse mikroteenust mis luuakse *Java Spring* raamistiku baasil. Antud teenus ühest küljest pakub *REST API* rakendusliidest ning teises küljes ühendub *Redis* kobar süsteemiga.

9.2.1 Redis Java konfiguratsioon

Kõigepealt on vaja lisada *Redis* sõltuvus *Spring* rakendusele. Selleks on esiteks vaja *Spring Data Redis* paketti. See sõltuvus tagab kerge konfiguratsiooni ja ligipääsu *Redisele Spring* rakendustest [36], [38]. Joonis 25 näitab lisatavat paketti.

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-redis</artifactId>
</dependency>
```

Joonis 25. Spring Redis data sõltuvus

Spring Data Redis integreerub kahe populaarse vabavaralise ühenduse pistikuga – *Lettuce* ja *Jedis*. *Jedis* on lihtsama funktsionaalsusega, seega kasutame seda [36], [40]. Joonis 26 näitab kliendi sõltuvuse lisamist.

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
</dependency>
```

Joonis 26. Jedis kliendi sõltuvuse lisamine

Järgnevalt tuleb kasutades lisatud sõltuvusi sätestada rakendusest ühendus *Redis* kobar süsteemi pihta. See toimib sarnaselt ühe instantsi vastu ühendusega [37]. Joonis 27 näitab kobar süsteemiga ühenduse loomist.

```
@Value("#{'${spring.data.redis.cluster.nodes}'.split(',')}")
private List<String> redisClusterNodes;


@Bean
public JedisConnectionFactory redisConnectionFactory() {

    RedisClusterConfiguration config = new
    RedisClusterConfiguration(redisClusterNodes);
    return new JedisConnectionFactory(config);
}
```

Joonis 27. Redis kobar süsteemiga ühenduse loomine Spring teenusest

Redis kobar süsteemida suhtlemisega tegeleb eraldi kobar süsteemi konfiguratsiooni klass. Sinna tuleb kaasa anda nimekiri kõikidest kobar süsteemi instantsidest. Nimekiri on lisatud *Kubernetese* konfiguratsiooni kaarti ning *Redise* seadistuse klassis viidatakse sellele. Järgmiseks on vaja sisenemis klassi *RedisTemplate,* läbi mille toimub *Redis* käskluste andmine *Spring* rakenduse koodist. Tegemist on peamise abstraktsiooniga *Redise* käskudele mida rakendusel vaja on, lisaks tegeleb see klass ühenduse haldusega [36], [37], [39]. Joonis 28 kirjeldab vastavat seadistust.

```
@Bean
public  RedisTemplate<String, Object> redisTemplate() {
  RedisTemplate<String, Object> template = new RedisTemplate<>();
  template.setConnectionFactory(redisConnectionFactory());
  template.afterPropertiesSet();
  return template;
}
```

Joonis 28. Redis template seadistus

9.2.2 Redis Repository

Repository on klass, mis peidab kasutaja eest toimingud aluseks oleva andmehoidla vastu, see sisaldab kõiki *CRUD* meetodeid ning muid lisa meetodeid andmehoidla andmetega tegelemiseks [36], [37], [39]. Joonis 29 annab ülevaate.

```
@Repository
public class RedisRepository {

    private RedisTemplate<String, Object> redisTemplate;
    private HashOperations hashOperations;


    public RedisRepository(final RedisTemplate<String, Object>
    redisTemplate) {
        this.redisTemplate = redisTemplate;
        this.hashOperations = redisTemplate.opsForHash();

    }

    public void create(final String hashmap, final String key, final Object
    object) {
        hashOperations.put(hashmap, key, object);

    }

    public Object get(final String hashmap, final String key) {
        return hashOperations.get(hashmap, key);
    }

    public Map<String, Object> getAll(final String hashmap){
        return hashOperations.entries(hashmap);
    }

    public void delete(final String hashmap, final String key) {
        hashOperations.delete(hashmap, key);

    }

    public void setExpire(final String targetImsi, final Integer
    cacheExpiration) {
        if (Objects.nonNull(cacheExpiration)) {
             redisTemplate.expire(targetImsi, cacheExpiration,
             TimeUnit.SECONDS);
        }
    }
}
```

Joonis 29. Geneeriline Repository klass

RedisTemplate kaudu keerukamate andmestruktuuride nagu objektide salvestamiseks kasutatakse räsi operatsioone. Redise räsid on disainitud kasutama vähem mälu, seega on see väga hea viis keerukamate andmestruktuuride salvestamiseks. Lisaks on siin meetod kirjele aegumisaja lisamiseks. [38], [39].

9.2.3 API kontrollerid

Teistele teenustele *Redis* vahemälu teenusega suhtlemiseks on loodud standardsed *API* kontrollerid lisa, otsi ja kustuta operatsioonideks. Joonis 30, Joonis 31 ja Joonis 32 esitavad loodud kontrollerid, kuhu on lisatud *Repository* klassi sõltuvus.

```
@PostMapping(value = "/stored/subInfo")
public ResponseEntity storeSubInfo(@RequestParam final String requestImsi,
@RequestParam final Integer expiration,  @RequestBody final Object object) {
    LOG.info("Received request to add subinfo for imsi: '{}'", requestImsi);
    try {
        if (StringUtils.isNotBlank(requestImsi)) {
             redisRepository.create(SUBINFO, requestImsi, object);
             redisRepository.setExpire(requestImsi, expiration);
             LOG.info("Added requestImsi {} and subInfo object {}",
             requestImsi, object);
             return ResponseEntity.ok(requestImsi);
        } else {
            return errorHandler.handleInvalidImsiError();
        }
    } catch (Exception ex) {
        return errorHandler.handleRedisGenericException(ex);
    }
}
```

Joonis 30. Kirje lisamise API kontroller

```
@DeleteMapping(value = "/remove/subInfo")
public ResponseEntity removeSubInfo(@RequestParam final String requestImsi) {
    LOG.info("Received request to remove subinfo imsi: '{}'", requestImsi);
    try {
        if (StringUtils.isNotBlank(requestImsi)) {
            redisRepository.delete(SUBINFO, requestImsi);
                    LOG.info("Removed {}", requestImsi);
            return ResponseEntity.ok(requestImsi);
        } else {
            return errorHandler.handleInvalidImsiError();
        }
    } catch (Exception ex) {
        return errorHandler.handleRedisGenericException(ex);
    }

}
```

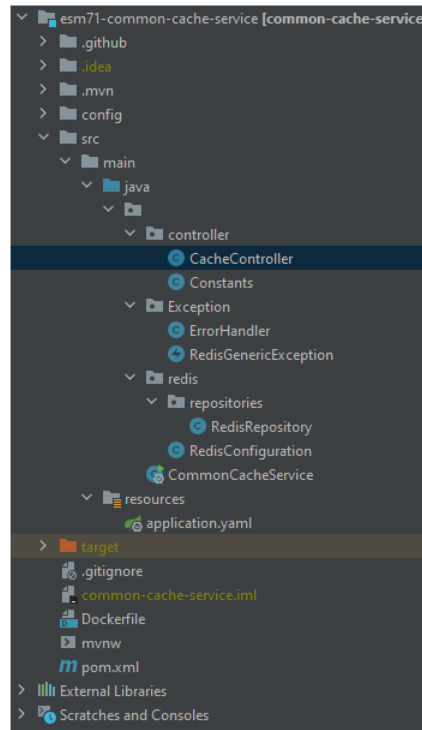
Joonis 31. Kirje eemaldamise kontroller

```java
@GetMapping(value = "/stored/subInfo")
public ResponseEntity getSubInfo(@RequestParam final String requestImsi) {
    LOG.info("Received request to find subInfo By requestImsi {}",
requestImsi);
    try {
        Object subInfo;
        if (StringUtils.isNotBlank(requestImsi)) {
            subInfo = redisRepository.get(SUBINFO, requestImsi);
        } else {
            return errorHandler.handleInvalidImsiError();
        }
        if (Objects.nonNull(subInfo)) {
            LOG.info("Found stored data for requestImsi: '{}'", requestImsi);
            return ResponseEntity.ok(subInfo);
        } else {
            return errorHandler.handleInfoNotFound(requestImsi);
        }
    } catch (Exception ex) {
        return errorHandler.handleRedisGenericException(ex);
    }
}
```

Joonis 32. Kirje otsimise kontroller

9.2.4 Vahemälu teenuse projekti struktuur

Terve uus vahemälu teenus sisaldab küllaltki vähe koodi ning kokku ainult seitset *Java* klassi, millest konkreetselt *Redise* jaoks on kaks. Joonis 33 esitab projekti struktuuri.



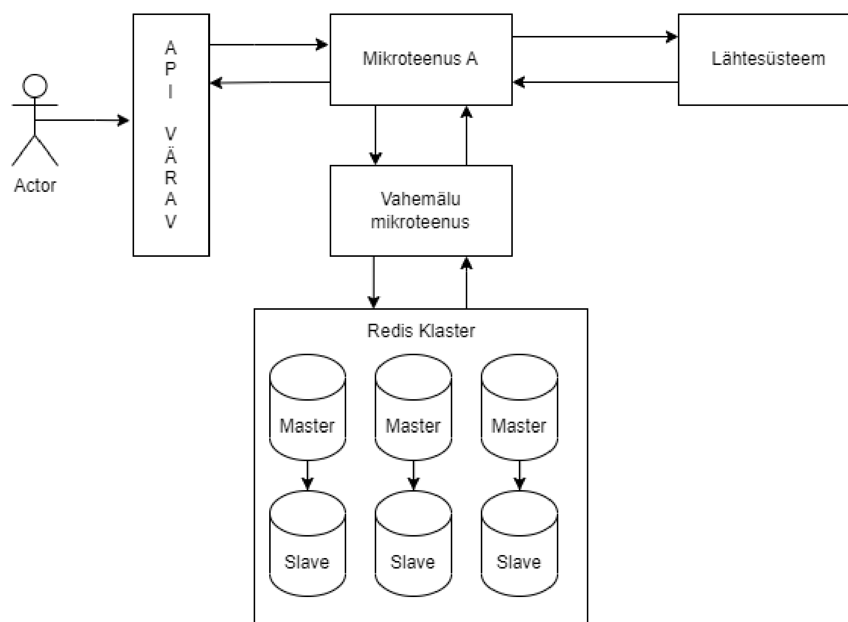
Joonis 33. Vahemälu teenuse projekti struktuur

10 Tulemused

Järgnevalt võetakse kokku tulemused, mis eelnevaga saavutatud on. Antud peatükk kajastab valmiduse taset, mõõtmis tulemusi päringu ajale ning edasisi plaane.

10.1 Valmidus tase ja testimine

Lahendus on rakendatud test keskkonnas nii *Redis* kobar süsteem, kui ligipääsu pakkuv mikroteenus mõlemad on rakendatud ühes test keskkonna *Kubernetese* kobar süsteemis. Lahendust on testitud nii otse manuaalsete päringutega ligipääsu pakkuvale mikroteenusele kui ka läbi teiste klient teenuste lõpust lõpuni integratsiooniga ehk kui suuremaid lahenduse komponente silmas pidada ning igat vahel olevat puhverserverit mitte nimetada, siis järgnev ahel. Joonis 34 annab ülevaate.



Joonis 34. Lahenduse testimine

10.2 Mõõtmistulemused

Järgnevalt tutvustatakse mõõtmistulemusi, mida antud hetkel võimalik saavutada oli, et tutvustada võimekuse kasvu. Tegelik lähtesüsteemi võimekus võib erineda, siin olevatest tulemustest, kuna tegemist on test keskkondadega. Valimis kasutati kümmet

erinevat sisend väärtust ning iga sisendiga korrati päringut 25 korda. Tabel 2 näitab mõõtmistulemusi ennem ning Tabel 3 näitab mõõtmistulemusi pärast test keskkonnas.

Tabel 2. Mõõtmis tulemused ennem

Turg	**Päringute maht**	**Päringud lähtesüsteemi**	**Unikaalsed päringud**	**Keskmine vastuse aeg**	**Õnnestumise %**
A	1,742,357	1,742,357	1,069,589	364ms	99.853%
B	986,201	986,201	687,928	222ms	99.696%
C	149,843	149,843	106,545	1701ms	99.581%

Tabel 3. Mõõtmis tulemused pärast test keskkonnas

Päringute maht	**Keskmine vastuse aeg**	**Keskmine lähtesüsteemi vastuse aeg**	**Keskmine Vahemälu vastuse aeg**
250	92ms	488ms	76ms

Tabel 3. välja toodud tulemustest on näha märgatavat päringute vastuse kiiruse kasvu. Antud suhte juures, kus iga päringu kohta lähtesüsteemi, teenindab mikroteenus vahemälu aegumis aja jooksul veel 24 päringut paraneb jõudlus 5.3 korda. Antud kordaja on siiski spekulatiivne, kuna iga sisendi kohta vahemälu aegumis aja jooksul 25 päringut ei ole. Kui võrrelda siin saadud vastuse aegu suhtega, mis oli algsetes mõõtmistulemustes turg A puhul ja eeldada, et kõik mitte unikaalsed päringud teenindati vahemälust Tabel 3. välja toodud keskmise ajaga ning kõik unikaalsed päringud said vastuse lähtesüsteemist, siin välja toodud keskmise lähtesüsteemi ajaga siis saame järgneva arvutuse teha.

$$((672768 \times 92) + (1069589 \times 488)) \div 1742357 = 335ms$$

Kasutades siin saadud päringute keskmist vastuse aega ning kasutades algmõõtmis tulemuste päringute mahtu saame keskmiseks päringu vastuse ajaks 335ms, mis on 1.33 korda kiirem. Samuti eeldades, et kõik korduv päringud algmõõtmis tulemustest saavad vastuse vahemälus, siis saame eeldada, et koormuse mahtu saab valitud aja tsükli jooksul vähendada umbes 39% võrra.

10.3 Nõuete täitmine

Antud lahendus täidab kõik funktsionaalsed ja mittefunktsionaalsed nõuded. Funktsionaalsete nõuete poolt võimaldab lahendus salvestada sageli kasutatavat teavet, salvestada teavet mitmelt erinevalt mikroteenuselt, küsida salvestatud teavet, andmete aegumise sätestamist ning samuti sündmuse või nõudluse põhist andmete kehtetuks tunnistamist. Mittefunktsionaalsete nõuete pool on antud lahendus laiendatav järgmistele teenustele, on laiendatav uute vajaduste tekkel, on lihtsalt skaleeritav andmete ja päringute mahu kasvul, on kõrge kättesaadavusega ning on hea jõudlusega.

10.4 Edasised plaanid

Edasised plaanid on rakendada lahendust esimesel võimaluse päris keskkonnas kõikide turgude raames. See eeldab veel lahenduse autori meeskonnasisest ülevaatamist, ning turu põhiste tooteomanike ning tehniliste kontaktidega kokku leppimist. Lisaks on vaja veel analüüsida kirje aegumisaega, kuigi tegemist suhteliselt staatiliste andmetega saab seal siiski muutusi esineda näiteks, kui mõne kliendi puhul muudetakse lepingut, mis tagab võimekuse antud lahenduses olevaid teenuseid kasutada ning veel mõningad juhud, kui *SIM* kaart on näiteks deaktiveeritud, kuid endiselt seadmes.

Lisaks on plaanis rakendada antud lahendust lahenduse analüüsis välja toodud teise probleemi korral, kus ahelas varem paiknev teenus saab teada oleva info lisada vahemällu teisele hiljem paiknevale teenuseks kasutamiseks.

11 Kokkuvõte

Töö eesmärgiks oli luua vahemälu lahendus mikroteenuste kihile, mida oleks lihtne laiendada ja järkjärgult teiste teenuste poolt kasutusele võtta. Eesmärgiga vähendada koormust lähtesüsteemidele korduv päringute näol ning tõsta mikroteenuste jõudlust vähendades kulutatud aega võrgu edasi tagasi reisile lähtesüsteemi ning lähtesüsteemi päringu protsessimise aja võrra. Töö käigus analüüsiti mitmeid vahemälu kasutamise strateegiaid ja viise ning erinevaid levinud tehnoloogiad andmete mälus hoidmiseks.

Töö tulemuse valiti välja strateegia, vahemälu asukoht ja andmete mälus hoidmise tehnoloogia ning seejärel uuriti lähemalt, kuidas *Redis* kobar süsteemi *Kubernetese* platvormil rakendada. Lisaks uuriti kuidas *Spring* raamistikul *Redis* kobar süsteemiga suhelda. Analüüsi osa tulemusena leiti seadistus, kuidas *Redis* kobar süsteem *Kuberneteses* seadistada ning rakendati töötav kõrge kätte saadavusega kobar süsteem. Lisaks loodi väike mikroteenus, mis ühest küljest tegeles *Redisega* suhtlemisega ning teisest küljest pakkus lihtsat CRUD rakendusliidest. Antud teenust saab kasutada ükskõik mitme erineva andmemudeliga, seega järgnevate turgude andmemudelite salvestamiseks, ei ole vaja antud teenuse pool muudatusi.

Töö tulemust testiti test keskkonnas täis integratsiooni ahelat. Selle jaoks loodi ühe turu mikroteenuses integratsioon vast loodud vahemälu teenuse vastu. Tulemused toodi välja ning prognoositi potentsiaalset kasutegurit, kui antud lahendust päris keskkonnas kasutada.

Kasutatud kirjandus

[1] Joydip Kanjilal, “Scaling Microservices Architecture using Caching”, 2021 [Veebimaterjal]. Loetud aadressil: https://www.developer.com/design/scaling-microservices-using-cache/

[2] Saeed Anabtawi, “Exploring Caching Patterns for Microservices Architecture”, 2023 [Veebimaterjal], Loetud aadressil: https://www.linkedin.com/pulse/exploring-caching-patterns-microservices-architecture-saeed-anabtawi/

[3] Krzysztof Atłasik, “Caches in Microservice architecture”, 2023 [Veebimaterjal], Loetud aadressil: https://softwaremill.com/caches-in-microservice-architecture/

[4] Joydip Kanjilal, “Consider these key microservices caching strategies”, 2020 [Veebimaterjal], Loetud aadressil: https://www.techtarget.com/searchapparchitecture/tip/Consider-these-key-microservices-caching-strategies

[5] Konstantinos Kalafatis “Application Caching Strategies”, 2022 [Veebimaterjal], Loetud aadressil: https:// linkedin.com/pulse/application-caching-strategies-konstantinos-kalafatis-1f/

[6] Sudheer Sandu “Distributed Caching — The Only Guide You’ll Ever Need”, 2022 [Veebimaterjal], Loetud aadressil: https://medium.com/@sudheer.sandu/distributed-caching-the-only-guide-youll-ever-need-fe152357f912

[7] Nicolas Frankel “A Hitchhiker’s Guide to Caching Patterns”, 2020 [Veebimaterjal], Loetud aadressil: https://hazelcast.com/blog/a-hitchhikers-guide-to-caching-patterns/

[8] Rafal Leszko “Where Is My Cache? Architectural Patterns for Caching Microservices”, 2019 [Veebimaterjal], Loetud aadressil: https://hazelcast.com/blog/architectural-patterns-for-caching-microservices/

[9] Simon Arneaud “A Tale of Three Server Caching Architectures”, 2016 [Veebimaterjal], Loetud aadressil: https://theartofmachinery.com/2016/07/30/server_caching_architectures.html

[10] Nicolas Frankel “From Embedded to Client-Server”, 2021 [Veebimaterjal], Loetud aadressil: https://hazelcast.com/blog/from-embedded-to-client-server/

[11] Rajiv Srivastava “ Cloud Distributed Caching for Microservices”, 2022 [Veebimaterjal], Loetud aadressil: https://cloudificationzone.com/2022/10/29/cloud-distributed-caching-for-microservices/

[12] IBM “What is Redis?”, [Veebimaterjal], Loetud aadressil: https://www.ibm.com/topics/redis

[13] Kamil Wisniowski “ Redis Architecture – (Single Instance, HA, Sentinel, Cluster)”, 2023 [Veebimaterjal], Loetud aadressil: https://cloudinfrastructureservices.co.uk/redis-architecture-single-instance-ha-sentinel-cluster

[14] Mahdi Yusuf “Redis Explained”, 2023 [Veebimaterjal], Loetud aadressil: https://architecturenotes.co/redis/

[15] Hazelcast “ Rethinking Redis?”, [Veebimaterjal], Loetud aadressil: https://hazelcast.org/compare-with-redis/

[16] Redis “High availability with Redis Sentinel”, [Veebimaterjal], Loetud aadressil: https://redis.io/docs/management/sentinel/

[17] Redis “Redis persistence”, [Veebimaterjal], Loetud aadressil: https://redis.io/docs/management/persistence/

[18] Memcached “Overview”, [Veebimaterjal], Loetud aadressil: https://github.com/memcached/memcached/wiki/Overview

[19] Intel Corporation “Configuration and Deployment Guide For Memcached on Intel Architecture”, 2013 [Veebimaterjal], Loetud aadressil: https://cdrdv2-public.intel.com/671319/dec-2013-update-configuration-and-deployment-guide-for-memcached.pdf

[20] Amazon Web Services “Memcached”, [Veebimaterjal], Loetud aadressil: https://aws.amazon.com/memcached/

[21] Melodies Sim “Lessons Learnt from Scaling Memcached in Production”, [Veebimaterjal], Loetud aadressil: https://levelup.gitconnected.com/lessons-learnt-from-scaling-memcached-in-production-86778ab616c7

[22] Juan Pablo Carzolio “A Guide to Consistent Hashing”, [Veebimaterjal], Loetud aadressil: https://www.toptal.com/big-data/consistent-hashing

[23] Hussein Nasser “Memcached Architecture”, 2022 [Veebimaterjal], Loetud aadressil: https://medium.com/@hnasr/memcached-architecture-af3369845c09

[24] Kamso Oguejiofor “Redis vs Hazelcast – What’s the Difference ? (Pros and Cons)”, 2023 [Veebimaterjal], Loetud aadressil: https://cloudinfrastructureservices.co.uk/redis-vs-hazelcast-whats-the-difference/

[25] Hazelcast “Hazelcast Overview”, [Veebimaterjal], Loetud aadressil: https://docs.hazelcast.com/imdg/3.12/hazelcast-overview

[26] Hazelcast “Persisting Data on Disk”, [Veebimaterjal], Loetud aadressil: https://docs.hazelcast.com/hazelcast/5.2/storage/persistence

[27] Hazelcast "Map Configuration", [Veebimaterjal], Loetud aadressil: https://docs.hazelcast.com/cloud/map-configurations

[28] DB-engines "DB-Engines Ranking of Key-value Stores", [Veebimaterjal], Loetud aadressil: https://db-engines.com/en/ranking/key-value+store

[29] Rancher Admin "Deploying Redis Cluster on Top of Kubernetes", 2019 [Veebimaterjal], Loetud aadressil: https://www.suse.com/c/rancher_blog/deploying-redis-cluster-on-top-of-kubernetes/

[30] Docker "Docker Hub", [Veebimaterjal], Loetud aadressil: https://www.docker.com/products/docker-hub/

[31] Docker, "Redis Docker Official Image", [Veebimaterjal], Loetud aadressil: https://hub.docker.com/_/redis

[32] Kubernetes, "ConfigMaps", [Veebimaterjal], Loetud aadressil: https://kubernetes.io/docs/concepts/configuration/configmap/

[33] Kubernetes, "StatefulSets", [Veebimaterjal], Loetud aadressil: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

[34] Kubernetes, "Service", [Veebimaterjal], Loetud aadressil: https://kubernetes.io/docs/concepts/services-networking/service/

[35] Redis, "Documentation", [Veebimaterjal], Loetud aadressil: https://redis.io/docs/

[36] Spring, "Spring Data Redis", [Veebimaterjal], https://docs.spring.io/spring-data/data-redis/docs/current/reference/html/#preface

[37] Redis, "Getting Started with Spring Data Redis", [Veebimaterjal], https://developer.redis.com/develop/java/redis-and-spring-course

[38] Andriy Redko, " Using Redis with Spring", 2013, [Veebimaterjal], https://dzone.com/articles/using-redis-spring

[39] Vipin KP, "Spring Boot with Redis: HashOperations CRUD Functionality", [Veebimaterjal], https://stackabuse.com/spring-boot-with-redis-hashoperations-crud-functionality/

[40] Redis, "Jedis vs. Lettuce: An Exploration", [Veebimaterjal], https://redis.com/blog/jedis-vs-lettuce-an-exploration/

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

Mina, Siim Milli

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "Vahemälu lahendus mikroteenuste kihile telekommunikatsiooniettevõtte näitel" mille juhendaja on Lauri Anton
   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.

14.05.2023

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 – Kubernetese Statefulset

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: common-cache-store
spec:
  serviceName: common-cache
  replicas: 6
  selector:
    matchLabels:
      app: common-cache-store-selector
  template:
    metadata:
      name: common-cache-store
      labels:
        app: common-cache-store-selector
    spec:
      containers:
        - name: common-cache-store
          image: docker-local.jfrog.company.io/es/folder/redis:alpine
          ports:
           - name: client
             containerPort: 6379
           - name: gossip
             containerPort: 16379
          command: [ "/conf/update-node.sh", "redis-server",
                     "/conf/redis.conf" ]
          env:
            - name: POD_IP
              valueFrom:
                fieldRef:
                  fieldPath: status.podIP
          imagePullPolicy: Always
          volumeMounts:
            - name: data
              mountPath: /data
              readOnly: false
            - name: conf
              mountPath: /conf
              readOnly: false
          resources:
            limits:
              memory: "256Mi"
              cpu: "200m"
            requests:
              memory: "128Mi"
              cpu: "100m"
      volumes:
```

```
      - name: conf
        configMap:
          name: common-cache-config
          defaultMode: 0755
    terminationGracePeriodSeconds: 10
volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "nas"
      resources:
        requests:
          storage: 200Gi
```