Arhitektura web aplikacija sa visokim nivoom saobraćaja

By Dragiša Stjepanović

UNIVERZITET U BANJOJ LUCI
UNIVERSITY OF BANJA LUKA

ELEKTROTEHNIČKI FAKULTET

ARHITEKTURA WEB APLIKACIJA SA VISOKIM NIVOOM SAOBRAĆAJA

MASTER RAD

Mentor:
prof. dr Slavko Marić

Kandidat:
Dragiša Stjepanović, dipl. ing. el.

Banja Luka, oktobar 2018. godine

UNIVERZITET U BANJOJ LUCI

ELEKTROTEHNIČKI FAKULTET

ARHITEKTURA WEB APLIKACIJA SA VISOKIM NIVOOM SAOBRAĆAJA

MASTER RAD

Mentor:
prof. dr Slavko Marić

Kandidat:
Dragiša Stjepanović, dipl. ing. el.

Banja Luka, oktobar 2018. godine

UNIVERSITY OF BANJA LUKA

FACULTY OF ELECTRICAL ENGINEERING

WEB APPLICATION ARCHITECTURE WITH HIGH LEVEL OF TRAFFIC

MASTER THESIS

Mentor:
prof. dr Slavko Marić

Candidate:
Dragiša Stjepanović, dipl. ing. el.

Banja Luka, October 2018

Mentor:	prof. dr Slavko Marić, redovni profesor, Univerzitet u Banjoj Luci, Elektrotehnički fakultet
Naslov master rada:	Arhitektura web aplikacija sa visokim nivoom saobraćaja
Rezime:	U radu je dat prijedlog arhitekture web aplikacija koja može opslužiti veliki broj korisnika i raditi nesmetano usljed otkaza nekog od servera posmatranog sistema. Naglasak je na skalabilnosti i dostupnosti web aplikacije. Posebna pažnja posvećena je klasterizaciji servera i keširanju podataka. Opisan je korišteni stek tehnologija za predloženu arhitekturu web aplikacija, kao i web aplikacija koja je razvijena za svrhe testiranja i koja se izvršava na implementiranoj arhitekturi. Dat je kratak pregled razvoja arhitekture, uključujući prednosti i nedostatke, počevši od jednoslojne arhitekture, zatim dvoslojne, troslojne i višeslojne arhitekture. Praktični dio rada počinje sa arhitekturom koja se sastojala od jednog web servera i jednog servera baza podataka. Postepeno su dodavane nove komponente arhitekture, testirane performanse čitanja podataka i praćen je napredak u odnosu na prethodno implementiranu arhitekturu. Izvršena je analiza dobijenih rezultata i date su preporuke za realizaciju arhitekture web aplikacija sa visokim nivoom saobraćaja, koja je skalabilna i otporna na otkaze.
Ključne riječi:	arhitektura web aplikacija, klasterizacija servera, Varnish server, MariaDB, Galera klaster, NoSQL, Redis, Redis klaster
Naučna oblast:	Prirodne nauke
Naučno polje:	Računarske i informacione nauke
Klasifikaciona oznaka:	-
Tip odabrane licence Kreativne zajednice:	CC BY-NC

Mentor:	prof. dr Slavko Marić, full professor, University of Banja Luka, Faculty of Electrical Engineering
Master thesis title:	Web application architecture with high level of traffic
Abstract:	In this master thesis is given a proposal of web application architecture, which can serve a big amount of users and work unhindered due to the failure of one of the servers of the observed system. The emphasys is on scalability and availability of web application. Special attention is dedicated to server clustering and data retrieval. Stack technology is described for the proposed application architecture, as well as web application which was developed for testing purposes and which executes on the implemented architecture. A brief overview of the development of architecture is given, including advantages and disadvantages, starting from single-tier architecture, then two-tier, three-tier and the concept of multitier architecture. The practical part of the paper begins with an architecture that consists of one web server and one database server. New components of architecture were gradually added, reading performance was tested and progress in relation to the previously implemented architecture was monitored. An analysis of the obtained data was made and recommendations were given for the implementation of the web application architecture with a high level of traffic that is scalable and resistant to failures.
Key words:	web application architecture, server clustering, Varnish server, MariaDB, Galera cluster, NoSQL, Redis, Redis cluster
Scientific area:	Natural sciences
Scientific field:	Computer and Information Science
Classification label:	-
Type of selected Creative Commons license:	

SADRŽAJ

1. UVOD

Naglim razvojem informaciono-komunikacionih tehnologija dolazi do pojave velike količine podataka koja se obrađuje i pohranjuje za buduću upotrebu. Sve više ljudi koristi Internet da bi pristupili tim podacima. Podaci se pohranjuju na udaljenim lokacijama širom svijeta. Krajnjim korisnicima je potrebno obezbijediti nesmetanu komunikaciju i pristup podacima. Pored dobro osmišljene fizičke arhitekture, potrebno je posvetiti pažnju samom softveru i hardveru, kako bi se iz njih mogao izvući maksimum i mogla obezbijediti pouzdana i brza komunikacija, bez otkaza sistema.

Osnovni cilj bilo koje web aplikacije jeste da smanji vrijeme odziva na korisničke akcije. Korisnici najčešće nemaju strpljenja da sjede i čekaju spori odgovor od neke web aplikacije. Ovakve situacije najčešće navode korisnike da napuste web sajt u korist konkurenata, tako da kašnjenje odgovora može predstavljati veliki problem. Međutim, sam otkaz sistema predstavlja najgoru moguću situaciju, pri čemu krajnji korisnik uopšte nema pristup podacima. Do otkaza sistema najčešće dolazi usljed preopterećenosti ili fizičkog otkaza nekog od servera. Jedan od primjera otkaza sistema je pad servera novinskih portala, kada se objavi neka udarna vijest. U trenutku objavljivanja takve vijesti, sistemu pristupa ogroman broj korisnika. Ka serveru se upućuje veliki broj zahtjeva i kada dođe do zasićenja mreže ili servera, pri čemu sistem nije dovoljno dobro isprojektovan, nastupa otkaz sistema. Novi zahtjevi pristižu, ali ih nije moguće obraditi. Sve dok se sistem ne oporavi nije moguće dobiti odgovor ni na jedan novi zahtjev korisnika.

U ovom radu posebna pažnja je posvećena mehanizmu za keširanje podataka i klasterizaciji servera, kako bi se implementirala arhitektura web aplikacija koja može da opsluži veliki broj korisnika, a istovremeno je otporna i na otkaze.

Vrijeme odziva web aplikacije zavisi od:

- propusne moći aplikativnog servera – koliko zahtjeva aplikativni server može da obradi u jedinici vremena,
- propusne moći servera baza podataka – koliko transakcija server baze podataka može da obradi u jedinici vremena i
- efikasnosti klijentskog nivoa – koliko brzo pretraživač može da prikaže rezultat korisničke akcije.

Takođe, postoje i drugi faktori koji utiču na vrijeme odziva web aplikacije, počevši od mrežnih performansi, zatim hardverskih performansi klijenta i servera, te projektovanja same arhitekture web aplikacija.

Tradicionalne arhitekture web orijentisanih sistema zadovoljavaju potrebe pri radu sa manjim brojem korisnika. Potrebno je obezbijediti zadovoljavajući mrežni protok i dovoljno dobru konfiguraciju servera za konkretnu namjenu. Cilj ovog rada i istraživanja je bio analiza i projektovanje arhitekture web aplikacija koja bi riješila nedostatke tradicionalne arhitekture pri radu sa velikim brojem korisnika. Suštinski, kako unaprijediti tradicionalnu arhitekturu sistema, izvršiti njenu analizu i postepeno dodavati nove komponente arhitekture, kako bi se povećala dostupnost i skalabilnost sistema. Mrežne performanse i hardver neće biti razmatrani zato što ne utiču direktno na prednosti i mane arhitekture web aplikacija, koja će biti analizirana i prikazana u ovom radu.

Performansa predstavlja mjeru koliko sistem efikasno može da obavi zadatak za koji je namijenjen. Sistem sa dobrim performansama odlikuje kapacitet za brzu obradu zahtjeva, skalabilnost prilikom povećanja opterećenja sistema i dostupnost čak i u nepredviđenim situacijama.

Ovaj rad sadrži šest poglavlja, uključujući i ovo uvodno poglavlje.

U drugom poglavlju opisano je funkcionisanje arhitekture web aplikacija i dat je pregled tipova arhitektura sa prednostima i nedostacima, počevši od jednoslojne arhitekture, zatim dvoslojne, troslojne i višeslojne arhitekture. Definisani su osnovni tipovi web aplikacija, kao i unapređenja u radu web aplikacija, od slanja statičkih fajlova do pojave servisno orijentisanih jednostraničnih web aplikacija. Opisani su pojmovi skalabilnosti i dostupnosti i istaknuta je njihova važnost prilikom projektovanja arhitekture i funkcionisanja aplikacija.

U trećem poglavlju opisan je korišteni stek tehnologija za implementaciju predložene arhitekture web aplikacija. Posebna pažnja posvećena je klasterizaciji servera sa ciljem povećanja dostupnosti, kao i keširanju korisnički nezavisnih i zavisnih podataka, kako bi se ubrzao rad web aplikacije. Dat je pregled tipova proxy servera, te je opisan rad i klasterizacija Varnish servera. Klaster Varnish servera namijenjen je za keširanje korisnički nezavisnih podataka. Naglasak je bio i na klasterizaciji MariaDB baze podataka, sa ciljem da se ubrza čitanje podataka iz baze i poveća dostupnost u slučaju otkaza nekog

od servera baza podataka. Korišten je Galera klaster. Dat je i kratak opis MariaDB baze podataka, kao i NoSQL baza podataka i predstavnika ključ-vrijednost NoSQL sistema za upravljanje bazama podataka - Redis servera. Opisan je rad Redis klastera, koji je korišten za svrhe keširanja korisnički zavisnih podataka.

U četvrtom poglavlju izložen je koncept skalabilne arhitekture web aplikacija. Dat je vizuelni prikaz i opisan je koncept funkcionisanja predložene arhitekture web aplikacija. Pored toga, navedena su ograničenja neophodna za ispravno funkcionisanje cijelog sistema.

U petom poglavlju opisani su projektni zahtjevi na osnovu kojih je izvršena implementacija predložene arhitekture web aplikacija. Dodatno je razvijena web aplikacija *Online news,* koja se izvršava na predloženoj arhitekturi. Testirane su performanse navedene aplikacije prilikom prikupljanja vijesti sa naslovne stranice i odabranih vijesti za korisnika. Postepeno su dodavani novi serveri i nove tehnologije sa ciljem povećanja dostupnosti i brzine odziva web aplikacije. Nakon svakog unapređenja arhitekture web aplikacije testirane su performanse i vršeno je poređenje u odnosu na prethodnu arhitekturu web aplikacija. Konačna arhitektura web aplikacija može opslužiti veliki broj korisnika i raditi nesmetano u slučaju otkaza nekog od servera u okviru posmatranog sistema. Izložena web arhitektura namijenjena je za web aplikacije koje se većim dijelom koriste za čitanje, a ne za upis podataka.

U šestom poglavlju data je kratka rekapitulacija istraživanja i izložena su zaključna razmatranja.

2. ARHITEKTURA WEB APLIKACIJA

Arhitektura web aplikacija definiše interakciju između klijentske aplikacije, srednjeg sloja i baze podataka, kako bi osigurala da više klijentskih aplikacija radi istovremeno [1].

Kada korisnik u svom web pretraživaču (eng. *web browser*) otkuca neki URL (Uniform Resource Locator) i potvrdi unos, na Internetu se traži pristupna tačka koja odgovara unesenom URL-u. Serveri koji se nalaze na unesenoj adresi generišu fajlove koji odgovaraju traženom upitu i šalju ih nazad klijentu, u ovom slučaju web pretraživaču. Kada fajlovi stignu na stranu web pretraživača, on ih procesira i prikazuje korisniku web stranicu (eng. *webpage*) koju je zahtijevao. Korisnik tada može da vrši interakciju sa web stranicom. U okviru web stranice nalazi se kôd na osnovu kojeg web pretraživač zna kako će postupiti kada korisnik izvrši neku aktivnost. Moguće je promijeniti nešto na stranici na osnovu podataka koje web pretraživač već posjeduje ili uputiti novi zahtjev ka istom ili nekom drugom web serveru, kako bi se prikupili potrebni podaci za realizaciju zahtijevane aktivnosti.

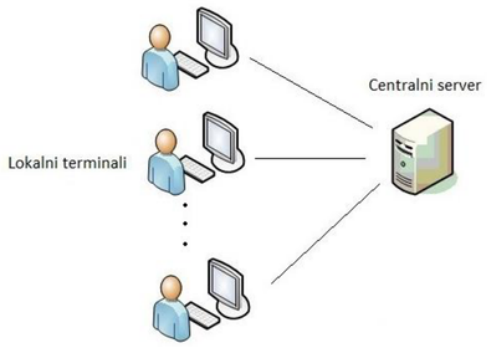
Zahtijevanje web stranice, njeno procesiranje i prikazivanje korisniku potrebno je da traje što je moguće kraće, kako korisnik ne bi izgubio interesovanje i napustio web sajt u korist konkurenata. Očekivanu brzinu odziva moguće je postići pravilnim projektovanjem i razvojem web aplikacije, kao i raznim mehanizmima keširanja podataka. Pored brzine odziva, neophodno je postići i skalabilnost, robusnost i sigurnost cijelog sistema.

2.1 Arhitekture aplikativnih sistema

Potrebno je izvršiti pažljivu analizu korisničkih zahtjeva kako bi se mogla donijeti ispravna odluka o arhitekturi aplikativnih sistema. Neophodno je uzeti u obzir trenutne potrebe korisnika, ali i obezbijediti arhitekturu sistema koja će podržati proširenje i budući razvoj aplikacija, koje će se izvršavati na prethodno pomenutoj arhitekturi. Odabir arhitekture zavisi od: broja klijenata, korištenih programskih alata, razvojnog okruženja, sigurnosti sistema, potrebne brzine odziva aplikacije, dostupnosti u slučaju otkaza nekog dijela sistema i drugih faktora.

Kod jednoslojne arhitekture postoji jedan centralni server (eng. *mainframe*) i lokalni terminali (Slika 2.1). Lokalni terminali služe samo za unos i prikaz podataka, dok centralni server izvršava sve ostale operacije. Kod ovakve arhitekture server je preopterećen jer

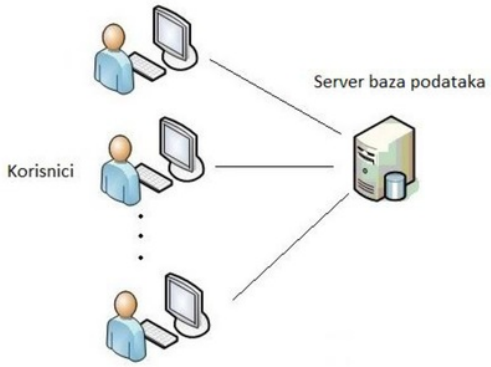
izvršava svu potrebnu logiku i skladišti podatke. Sa porastom broja korisnika potrebno je povećavati i performanse centralnog servera. Konstantnim zahtjevima za povećavanjem performansi doći će do zasićenja servera, pri čemu bi dalji rast broja korisnika otežao obradu korisničkih zahtjeva ili doveo do otkaza sistema.



Slika 2.1 – Jednoslojna arhitektura

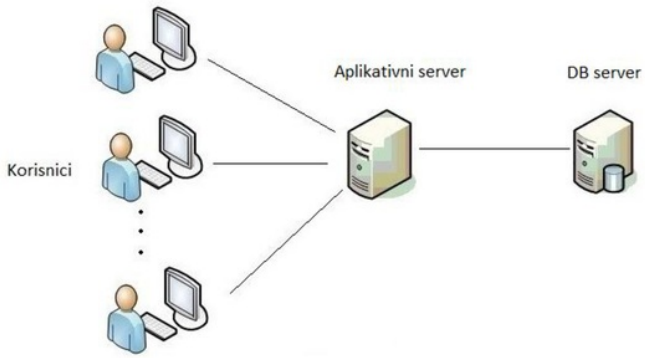
Klijent-server arhitektura je zasnovana na razmjeni podataka između klijenta i servera. Klijentske aplikacije su optimizovane za prikaz interfejsa i interakciju sa korisnikom. Server je optimizovan za skladištenje i obradu podataka. Postoji mogućnost horizontalnog i vertikalnog skaliranja. Horizontalnim skaliranjem postižu se bolje performanse dodavanjem novih servera koji rade u paraleli. Vertikalnim skaliranjem povećavaju se performanse postojećih servera. Podaci se čuvaju na serveru, umjesto na svakom klijentu, čime se smanjuje redundantnost podataka.

Dvoslojnu klijent-server arhitekturu čine: klijentski računari i server baza podataka (Slika 2.2). Klijentski računar pristupa bazi podataka, izvršava svu potrebnu logiku i obezbjeđuje prikaz korisničkog interfejsa. Na serveru se nalazi baza podataka, pri čemu server služi samo za prihvatanje zahtjeva i vraćanje podataka klijentu iz baze. Brzina rada aplikacije zavisi prevashodno od performansi klijentskog i serverskog računara, kao i mrežnih performansi.



Slika 2.2 – Dvoslojna klijent-server arhitektura

Kod troslojne klijent-server arhitekture, pored klijentskih računara i servera baza podataka nalazi se i aplikativni server (Slika 2.3). Namjena aplikativnog servera je da vodi računa o autentikaciji i autorizaciji korisnika, prihvata klijentske zahtjeve, pristupa serveru baze podataka, prikuplja podatke iz baze, izvršava poslovnu logiku i vraća odgovor klijentu. Na klijentskom računaru se vrši prikaz korisničkog interfejsa, registrovanje korisničkih akcija i događaja, slanje zahtjeva aplikativnom serveru, prikupljanje podataka od aplikativnog servera, pri čemu se može izvršavati i dio logike aplikacije. Web aplikacije su često bazirane na troslojnoj klijent-server arhitekturi. Putem računarske mreže ostvaruje se razmjena podataka između aplikativnog servera i klijenta.



Slika 2.3 – Troslojna klijent-server arhitektura

Daljim proširivanjem koncepta troslojne arhitekture dolazi se do pojma višeslojne arhitekture, kod koje se vrši dalja podjela na komponente na serverskoj strani, sa ciljem povećanja performansi sistema.

Višeslojna (eng. *multitier* ili *N-tier*) arhitektura je klijent-server arhitektura kod koje se prezentacione, aplikativne i funkcije nad bazom podataka fizički i logički odvajaju.

Troslojna arhitektura (eng. *3-tier architecture*) je najzastupljenija višeslojna arhitektura kod koje broj slojeva iznosi 3.

Prednosti višeslojne arhitekture:

- veća sigurnost – sprovodi se na svakom sloju,
- svakim slojem se može zasebno upravljati, bez uticaja na ostale,
- bolja skalabilnost – moguće je dodavati nove resurse na svakom od slojeva, bez promjene ostalih i
- bolja fleksibilnost – u zavisnosti od potreba moguće je proširiti svaki sloj [2].

Višeslojna arhitektura omogućava da se dodaju nove komponente i uvode nove tehnologije, bez potrebe da se vrši redizajn ili mijenja cijela aplikacija. Jedan od primjera kako bi se mogao ubrzati odziv web aplikacije je keširanje podataka. Naravno, potrebno je voditi računa o podacima koji se keširaju, da se ne bi postigao kontraefekat. Ukoliko bi se dodala neka od ključ-vrijednost NoSQL baze podataka, koja bi se koristila za keširanje podataka, bilo bi potrebno izvršiti izmjenu samo serverskog dijela aplikacije, pri čemu se klijentska aplikacija i baza podataka ne bi mijenjali.

Posmatrajući sa aspekta razvoja aplikacija, višeslojna arhitektura nudi brži i efikasniji razvoj. Moguće je oformiti timove, gdje bi svaki tim bio zadužen za svoj nivo. Eksperti za dizajn i prezentaciju aplikacije radili bi na prezentacionom nivou, eksperti za poslovnu logiku pisali bi serverski dio koda i radili na aplikativnom nivou, dok bi eksperti za baze podataka radili na nivou baza podataka. Pošto je aplikacija prilagođena za rad na višeslojnoj arhitekturi, odnosno podijeljena je na dijelove, moguća je jednostavna ponovna upotreba već postojećeg koda za potrebe drugih projekata. Npr. moguće je iskoristiti istu bazu podataka za realizaciju drugog projekta, ili iskoristiti isti aplikativni nivo da bi se kreirao drugačiji prezentacioni sloj za krajnjeg korisnika.

2.2 Funkcionisanje arhitekture web aplikacija i tipovi web aplikacija

Web aplikacije su klijent-server aplikacije, čiji se rad bazira na izvršavanju koda koji se nalazi na serverskoj strani (eng. *back-end*) i koda koji se nalazi na klijentskoj strani (eng. *front-end*).

U suštini, to su dva programa koja se izvršavaju konkurentno:

- program koji se nalazi na strani klijenta i koji odgovara na korisničke akcije i
- program koji se nalazi na strani web servera i odgovara na HTTP zahtjeve [1].

HTTP (*Hypertext Transfer Protocol*) je dizajniran tako da omogući komunikaciju između klijenta i servera. Radi na principu zahtjev-odgovor. Klijent (npr. web pretraživač) šalje HTTP zahtjev na server, server generiše odgovor i šalje ga klijentu. Odgovor sadrži informaciju o statusu odgovora i generisani sadržaj koji se prosljeđuje klijentu [3].

Za pisanje programa na klijentskoj strani najčešće se koristi JavaScript programski jezik, uključujući HTML i CSS. JavaScript se često i definiše kao programski jezik za pisanje koda na klijentskoj strani. Kôd se izvršava na klijentskom računaru nakon učitavanja web stranice. Hypertext Markup Language (HTML) je standardizovani jezik za kreiranje web stranica. HTML elementi čine gradivne blokove HTML stranica i predstavljaju se putem tagova. Pretraživači ne prikazuju HTML tagove, ali ih koriste za prikaz sadržaja stranice. CSS (Cascading Style Sheets) je jezik koji opisuje kako će se HTML elementi prikazati u okviru web stranice.

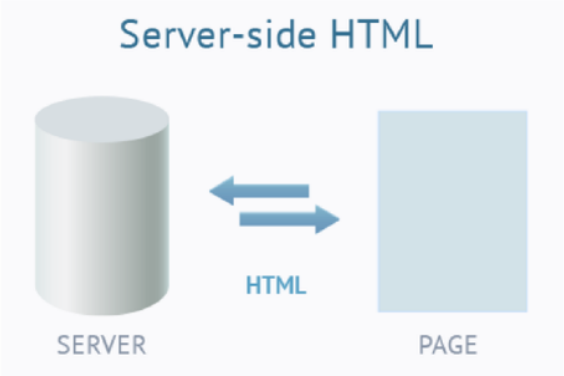
Postoji niz programskih jezika koji se koriste za pisanje koda na serverskoj strani. Neki od najviše korištenih su: JAVA, PHP i .NET(C#, VB). Programi koji se izvršavaju na serverskoj strani prihvataju zahtjeve sa klijentske strane, generišu odgovor i šalju ga klijentu.

Tipovi web aplikacija su:

- serverski orijentisane HTML web aplikacije,
- web aplikacije bazirane na funkcionalno nezavisnim jedinicama (eng. *widgets*), koje koriste AJAX (Asynchronous JavaScript And XML) i
- servisno orijentisane jednostranične web aplikacije [4].

Kod serverski orijentisanih HTML web aplikacija web serveri generišu HTML stranice koje se prosljeđuju klijentu prilikom svakog zahtjeva (Slika 2.4). To su statičke web stranice kod kojih svaki poziv sa klijentske strane zahtijeva da se izgeneriše nova HTML stranica i pošalje klijentu. Pri ovakvom funkcionisanju aplikacija šalje se velika količina podataka između klijenta i servera, što rezultuje lošim odzivom. Klijent mora sačekati da se učita cijela web stranica, čak iako je došlo do promjene samo dijela web stranice. Usljed slanja velike količine podataka i ograničenja brzine mobilnog interneta, ovakav način rada

je teško prihvatljiv kod mobilnih telefona. Implementacija ovakvih aplikacija je prilično jednostavna i predstavlja najstariji tip web aplikacija.



Slika 2.4 – Serverski orijentisane HTML web aplikacije [4]

Zbog loših performansi kod prvog tipa aplikacija, gdje se usljed promjene dijela web stranice zahtijeva ponovno generisanje i slanje cijele stranice, došlo se na ideju da se izmjena stranice vrši putem asinhronih zahtjeva.

AJAX omogućava:

- izmjenu dijela web stranice bez ponovnog učitavanja cijele stranice,
- zahtijevanje podataka od servera nakon što je stranica učitana,
- prihvatanje podataka od servera nakon što je stranica učitana i
- slanje podataka serveru bez blokiranja klijentskog interfejsa – slanje podataka se izvršava u pozadini (eng. *background*).

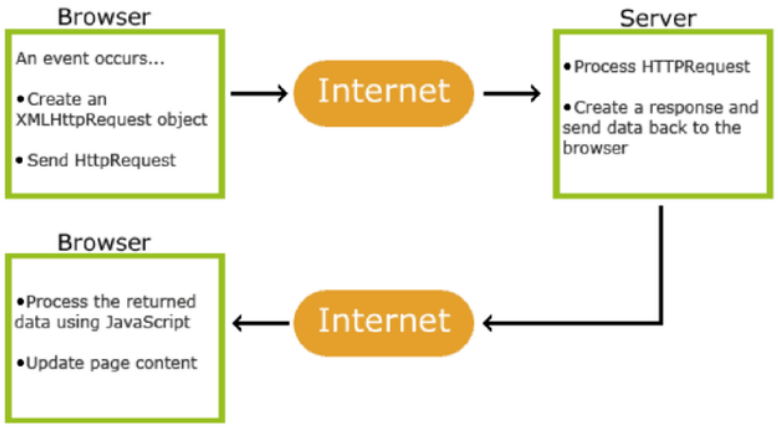
AJAX nije programski jezik. Predstavlja kombinaciju:

- ugrađenog XMLHttpRequest objekta u pretraživaču (koristi se za slanje HTTP zahtjeva web serveru) i
- JavaScript-a i HTML DOM-a (za prikaz ili korištenje podataka).

Na sljedećoj slici (Slika 2.5) prikazan je tok aktivnosti prilikom korištenja AJAX-a:

1. na web stranici se generiše neki događaj (npr. klik na neko dugme),
2. JavaScript kodom se kreira XMLHttpRequest objekat,
3. putem XMLHttpRequest objekta šalje se HTTP zahtjev web serveru,
4. web server obrađuje zahtjev,
5. server šalje odgovor nazad na web stranicu,
6. JavaScript kodom se obrađuje odgovor,

7. izvršava se neka akcija korištenjem JavaScript-a (npr. izmjena sadržaja web stranice).



Slika 2.5 – Kako radi AJAX [5]

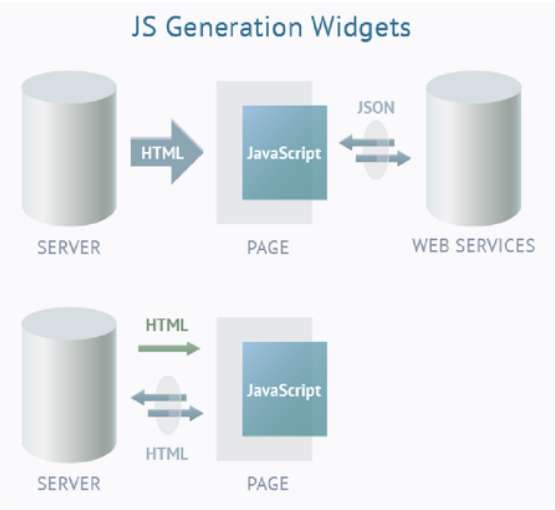
JavaScript aplikacije bazirane na AJAX-u razvile su se iz prvog tipa web aplikacija. Razlika u odnosu na prvi tip je u tome što se web stranica koja se prikazuje sastoji od funkcionalno nezavisnih jedinica. Putem AJAX zahtjeva funkcionalno nezavisne jedinice dobijaju podatke od servera. Podaci koji se dobijaju su u vidu:

- dijelova HTML-a, koji se mogu direktno ugraditi u web stranicu putem nekih jednostavnijih alata (npr. jQuery), bez korištenja JavaScript-a i
- JSON podataka, koji se putem JavaScript-a transformišu u sadržaj web stranice.

JSON (JavaScript Object Notation) je sintaksa koja se koristi za snimanje i razmjenu podataka. JavaScript objekti se mogu konvertovati u JSON. JSON predstavlja tekst koji se u ovom formatu može proslijediti serveru. Slična situacija je i prilikom primanja podataka sa servera. Moguće je konvertovati JSON u JavaScript objekat. Na ovaj način se sa JSON sintaksom olakšava slanje, parsiranje i korištenje podataka.

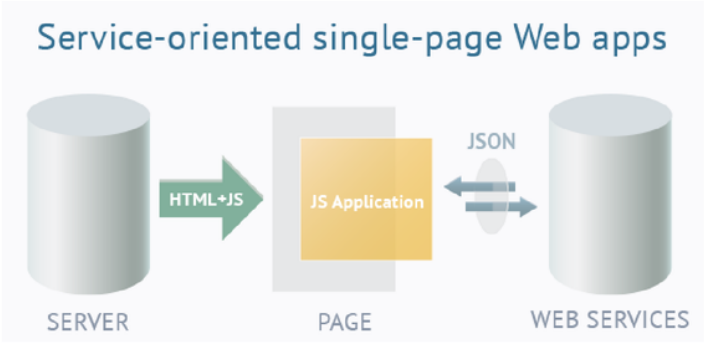
Osnovna prednost ovog tipa aplikacija je što izmjena funkcionalno nezavisnih dijelova web stranice (widget-a) ne utiče na cijelu web stranicu. Smanjuje se količina podataka koja se razmjenjuje između servera i klijenta, pri čemu je i odziv aplikacija bolji. Prvo učitavanje stranice traje nešto duže, zbog toga što se mora učitati i JavaScript kôd kojim se vrši asinhrona komunikacija. Ovo je cijena koja se mora „platiti“ da bi dobili bolji odziv web aplikacije nakon učitavanja stranice, u odnosu na prvi tip aplikacija.

Na sljedećoj slici (Slika 2.6) prikazano je funkcionisanje web aplikacija baziranih na funkcionalno nezavisnim jedinicama. Server na zahtjev klijenta šalje HTML stranicu koja sadrži i JavaScript kôd. Nakon učitavanja web stranice vrši se razmjena podataka korištenjem AJAX-a između klijenta i servera putem web servisa.



Slika 2.6 – Web aplikacije bazirane na funkcionalno nezavisnim jedinicama koje koriste AJAX [4]

Treći tip aplikacija predstavljaju servisno orijentisane jednostranične web aplikacije. Rad ovih aplikacija bazira se na jednoj HTML stranici koja se na zahtjev klijenta povlači sa servera. HTML stranica sadrži JavaScript kôd koji predstavlja JS aplikaciju (Slika 2.7). JavaScript aplikacija generiše HTML sadržaj i vrši prikaz web stranice, mijenja sadržaj web stranice, komunicira sa serverom putem web servisa, pri čemu se šalju samo podaci koji se prikazuju korisniku. Komunikacija je asinhrona i koristi se AJAX. Dio funkcionalnosti i logike je prebačen na klijentsku stranu, čime se jednim dijelom rasterećuje rad web servera. Pošto se cijela JavaScript aplikacija nalazi na strani klijenta, ovo predstavlja potencijalni problem sa aspekta sigurnosti, jer se aplikacija može relativno lako modifikovati od strane malicioznih korisnika. Potrebno je posvetiti veću pažnju samoj sigurnosti rada aplikacije na serverskoj strani (adekvatna autentikacija i autorizacija korisnika i siguran transfer podataka). Između klijenta i servera se razmjenjuju samo podaci koji se prikazuju korisniku, čime je količina saobraćaja svedena na minimum, a samim tim se povećava i odziv aplikacije na najviši nivo. Ovaj tip aplikacija pogodan je i za korištenje na mobilnim telefonima i tablet uređajima.



Slika 2.7 – Servisno orijentisane jednostranične web aplikacije [4]

2.3 Skalabilnost i dostupnost

Internet se sve više koristi kao globalna računarska mreža. Korisnici putem Interneta pristupaju web stranicama kako bi došli do željenih informacija. Kod web aplikacija najzastupljenija je troslojna klijent-server arhitektura. Povećanje broja konkurentnih korisnika i obima podataka dovodi do toga da web aplikacije i njihove baze podataka moraju biti skalabilne. Ukoliko je potrebno zadovoljiti visok nivo saobraćaja, gdje je odziv aplikacije od presudnog značaja, dolazi do potrebe za skaliranjem servera i dodavanjem mehanizama koji omogućavaju keširanje podataka. Dodavanjem novih komponenti na aplikativnom nivou prelazi se sa troslojne na višeslojnu arhitekturu. Pored brzog odziva, koji je za krajnjeg korisnika najbitniji, neophodno je povećati i dostupnost web aplikacije, kako bi ona radila i u slučaju otkaza neke komponente sistema.

Postoje dva tipa skaliranja: horizontalno (eng. *scale out*) i vertikalno (eng. *scale up*). Vertikalno skaliranje podrazumijeva centralizovani pristup koji se oslanja na sve moćnije servere sa više procesorske snage i većom radnom memorijom, dok horizontalno skaliranje podrazumijeva distribuirani pristup, odnosno pristup sa više fizičkih ili virtuelnih servera.

Posmatrajući troslojnu arhitekturu (Slika 2.3) sa jednim aplikativnim serverom i jednim serverom baza podataka, u slučaju porasta broja konkurentnih korisnika, u početku se bolje performanse postižu vertikalnim skaliranjem servera. Međutim, u nekom trenutku kapacitet i „najmoćnijeg" servera dostiže limit, cijena raste, a performanse polako opadaju [6]. U ovakvim slučajevima neophodno je vršiti horizontalno skaliranje, koje podrazumijeva dodavanje ne tako „moćnih" servera, na koje će se rasporediti posao i koji će raditi u paraleli. Suštinski, neophodno je kombinovati horizontalno i vertikalno skaliranje kako bi se postigli zadovoljavajući rezultati.

Dostupnost sistema se povećava dodavanjem redundantnih servera, pri čemu sistem nastavlja da funkcioniše i u slučaju otkaza nekog od servera.

Serveri mogu da rade u dva moda:

- aktiv/pasiv mod – u slučaju otkaza *master* servera, *slave* server preuzima njegovu ulogu i
- rad u paraleli (eng. *load balancing*) – vrši se raspodjela opterećenja između servera sa istom ulogom, pri čemu ako dođe do otkaza nekog od servera ostali nastavljaju sa radom, pod uslovom da je zadovoljen definisani kvorum (broj servera koji mora biti dostupan da bi klaster nastavio sa radom).

Zahtjevi sa klijentske strane mogu se podijeliti u dvije kategorije:

- zahtjevi kod kojih server ne treba da pamti informacije o stanju između zahtjeva (eng. *stateless*) – korisnički nezavisni podaci i
- zahtjevi kod kojih server mora da pristupa informacijama koje su generisane tokom prethodnih zahtjeva (eng. *stateful*) – korisnički zavisni podaci.

Performanse sistema se mogu znatno povećati keširanjem podataka, ukoliko se keširanje podataka vrši na adekvatan način. Keširanje korisnički nezavisnih podataka vrši se ispred web servera i to su podaci koji su isti za sve korisnike. Keširanim korisnički zavisnim podacima pristupa web server i oni su specifični za svakog korisnika.

3. SERVERSKE TEHNOLOGIJE I TEHNOLOGIJE BAZA PODATAKA

U ovom poglavlju dat je pregled serverskih tehnologija i tehnologija baza podataka, koje su korištene u praktičnom dijelu rada za implementaciju predložene arhitekture web aplikacija sa visokim nivoom saobraćaja, sa ciljem povećanja skalabilnosti i dostupnosti cijelog sistema.

3.1 Proxy serveri

Proxy serveri su posrednici između klijenata koji šalju zahtjeve za nekim resursima i servera koji vraćaju resurse nazad klijentima.

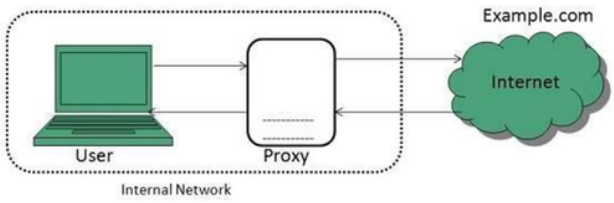
Uloga proxy servera je:

- kontrolisanje i filtriranje saobraćaja,
- poboljšanje performansi,
- prevođenje (eng. *translation*) – mijenjanje izvornog sadržaja,
- anoniman pristup uslugama i
- postizanje većeg nivoa sigurnosti.

Tipovi proxy servera:

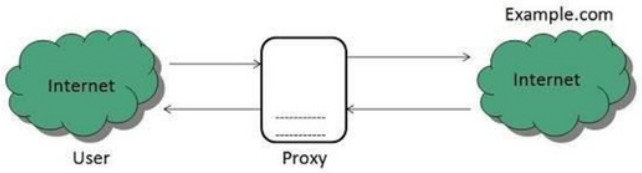
- forward proxy,
- open proxy i
- reverse proxy [7].

Forward proxy se koristi za pristup Internetu, odnosno klijenti putem proxy servera šalju zahtjeve i preuzimaju resurse sa Interneta.



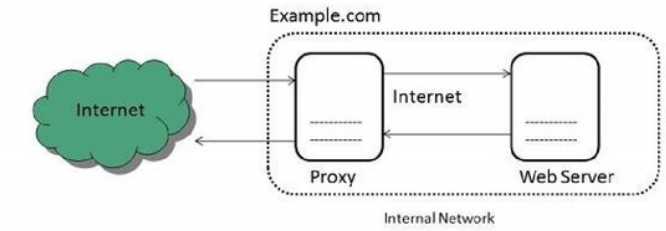
Slika 3.1 – Forward proxy [7]

Open proxy omogućava korisnicima da sakriju svoju IP adresu prilikom pretraživanja web-a.



Slika 3.2 – Open proxy [7]

Reverse proxy vrši kontrolu pristupa serverima u privatnoj mreži.



Slika 3.3 – Reverse proxy [7]

3.1.1 Kratak pregled karakteristika i primjena Varnish servera

Varnish server je *reverse proxy*. Predstavlja dio softvera koji se stavlja ispred web servera da bi se smanjilo vrijeme odziva web sajta ili aplikacije raspodjelom opterećenja i/ili keširanjem odgovora od web servera. U pitanju je *open source* alat pisan u programskom jeziku C [8].

Dobre performanse su veoma bitne za krajnjeg korisnika. Web stranica treba da radi približno dobro i sa milion, kao i sa deset korisnika. Potrebno je uočiti razliku između performanse i skalabilnosti. Performansa je brzina odziva web sajta, odnosno predstavlja vrijeme koje je potrebno da bi se učitala neka web stranica, dok skalabilnost predstavlja održavanje performansi pri povećanju opterećenja. Keširanje je ključni mehanizam za povećanje i održavanje performansi. Bez keširanja podataka rast broja korisnika dovodi do potrebe za mijenjanjem koda aplikacije ili optimizacije infrastrukture.

Keširanje korisnički nezavisnih podataka znatno ubrzava rad aplikacije bez potrebe da se mijenja kôd aplikacije ili infrastruktura. Loše performanse znatno utiču na *google* rangiranje web sajtova.

Varnish serverom se smanjuje vrijeme odziva bez mnogo truda. Naravno, sam Varnish nije dovoljan za kvalitetnu realizaciju arhitekture web aplikacija. On predstavlja samo dio lanca koji utiče na pouzdanost i brzinu odziva uzevši u obzir: mrežu, ostale servere, operativni sistem i samo izvršavanje aplikacije na instaliranom operativnom sistemu.

Varnish se može instalirati na istoj mašini gdje se nalazi i web server ili na zasebnim mašinama. Može komunicirati sa jednim ili više web servera. Unaprijed zauzima dio virtuelne memorije koju će koristiti za čuvanje objekata. Svaki objekat sadrži zaglavlje (eng. *header*) HTTP odgovora, kao i sadržaj koji se dobije od web servera. Objekti u kešu se identifikuju putem heš (eng. *hash*) vrijednosti, koja se podrazumijevano dobija na osnovu URL-a zahtjeva (ako *hostname* nije specifikovan koristi se IP adresa servera). Objekti se iz memorije vraćaju na zahtjev krajnjih korisnika. Pravilnim konfigurisanjem Varnish može ubrzati rad web aplikacije i do 1000 puta [8].

Varnish koristi VCL (Varnish Configuration Language) jezik za kontrolisanje keširanja podataka. VCL konfiguracioni fajl sadrži podrutine (eng. *subroutines*) i VCL kôd. Prilikom pokretanja Varnish servera vrši se čitanje VCL fajla, koji se prevodi u C, kompajlira i dinamički učitava. VCL fajl se može učitati i u toku rada Varnish servera. Moguće je pisati proizvoljne Varnish module u C programskom jeziku, kojim se obogaćuje VCL sintaksa. Upravo sâm VCL predstavlja razlog zašto ljudi daju prednost Varnish serveru u odnosu na neke druge mehanizme keširanja. Sa par linija koda moguće je spriječiti i DDoS (Distributed Denial of Service) napade.

Osnovni cilj većine web aplikacija jeste prikaz podataka koji dolaze iz baze ili nekog drugog spoljašnjeg izvora. Mnogo vremena se gubi na prikupljanje, obradu i prikaz podataka. Bez keširanja podataka ovaj proces se ponavlja za svaki korisnički zahtjev ka istom resursu. Bespotrebno se troši dodatno vrijeme za prikupljanje i obradu podataka iako se podaci nisu mijenjali. Keširanje predstavlja balansiranje između prikaza „svježih" podataka i prihvatljivog odziva aplikacije. Keširanje podataka nije način da se nadoknade loše performanse sistema ili aplikacije, već je odluka kojom se povećava efikasnost i smanjuju troškovi infrastrukture, pod uslovom da se sprovede na adekvatan način.

Veličina keš memorije zavisi od veličine i broja objekata koji se snimaju. Idealno bi bilo kada bi se svi podaci čuvali u kešu. U tom slučaju aplikacija bi radila veoma brzo. Međutim, prostor za skladištenje podataka RAM (Random Access Memory) memorije je ograničen. U slučaju da Varnish iskoristi svu predviđenu memoriju za keširanje podataka,

koristi se LRU (Least Recently Used) algoritam za izbacivanje objekata iz keša. Ukoliko se ne specifikuje veličina raspoložive RAM memorije, Varnish bi potencijalno mogao da iskoristi svu RAM memoriju servera. Ako bi se upotrijebila sva RAM memorija servera koristio bi se *swap* prostor za prebacivanje dijela podataka iz RAM memorije na disk. Ovo bi generalno moglo da uspori rad cijelog sistema ukoliko bi se disk preopteretio. Varnish pamti koliko se puta koristio neki objekat iz keša. Kada dođe do nedostatka memorije izbacuju se najmanje korišteni objekti, dok se ne oslobodi dovoljno prostora za smještanje novog objekta.

Ukoliko se u zaglavlju ne navede vrijeme keširanja objekta (*time-to-live*) ili nije specifikovano u VCL konfiguracionom fajlu, podrazumijevano je 120 sekundi. Varnish koristi memoriju za keširanje objekata, ali i za logovanja i statistike. Ne uključuje podršku za TLS (Transport Layer Security), jer nije prvobitno osmišljen za te svrhe. Ideja je da se sigurna komunikacija ukloni prije nego što zahtjev dođe do Varnish servera. Potrebno je ubaciti još jedan sloj ispred Varnish servera koji bi uklonio TLS konekciju i komunicirao putem HTTP protokola sa Varnish serverom. Za razliku od ostalih proxy alata, Varnish je HTTP akcelerator (eng. *accelerator*), što znači da Varnish isključivo koristi HTTP protokol. Postoji i Varnish Plus, naprednija verzija Varnish servera, koja podržava SSL/TLS komunikaciju bez korištenja drugih alata.

Varnish kešira jedino odgovore GET i HEAD zahtjeva, dok ostale šalje direktno na pozadinski server (eng. *backend*) i ne vrši keširanje njihovog odgovora. Postoje dva načina da se sačuva sesija korisnički zavisnih podataka: putem autorizacionog zaglavlja ili putem kolačića (eng. *cookie*). Kada Varnish registruje bilo koji od prethodno pomenuta dva načina čuvanja sesije, vrši se prosljedivanje zahtjeva ka pozadinskom serveru i ne kešira se njihov odgovor. Ovo je rezultat toga što su zahtjevi koji uključuju autorizaciono zaglavlje ili kolačić jedinstveni za svakog korisnika. Ukoliko bi se ipak odlučili za keširanje odgovora ovakvih zahtjeva, prvi korisnik bi dobio tražene podatke, ali i svi ostali korisnici koji bi pristupali istom odredištu (eng. *endpoint*) dobili bi isti odgovor kao i prvi korisnik. U odgovoru korisnički zavisnih podataka mogu se nalaziti osjetljivi ili neodgovarajući podaci. Iz ovih razloga nije dobra praksa keširati korisnički zavisne podatke korištenjem Varnish servera.

Potrebno je podesiti vrijeme koje će označavati koliko objekat „živi" u keš memoriji (*time-to-live*). HTTP posjeduje dva načina da se ovo postigne putem zaglavlja koji se nalazi u odgovoru web servera:

- Expires – vrijeme isteka (npr. Expires: Sat, 16 Sep 2017 15:30:00 GMT) i
- Cache-control – vrijeme u sekundama koje označava koliko će se objekat nalaziti u kešu prije nego što zastari (npr. Cache-control: public, max-age=3600, s-maxage=86400). Cache-control posjeduje sljedeća polja:
    - public – označava da i web pretraživač (eng. *browser*) i dijeljeni keš mogu keširati sadržaj,
    - max-age – vrijeme keširanja u sekundama koje se mora poštovati od strane pretraživača i
    - s-maxage – vrijeme keširanja u sekundama koje se mora poštovati od strane proxy servera.

Varnish na osnovu *time-to-live* vrijednosti odlučuje koliko će se dugo objekat keširati. TTL vrijednost se dobija na osnovu sljedeće prioritetne liste:

1. vrijednost beresp.ttl polja iz VCL konfiguracionog fajla pod uslovom da je postavljena,
2. na osnovu Cache-control zaglavlja i postavljene vrijednosti za s-maxage,
3. na osnovu Cache-control zaglavlja i postavljene vrijednosti za max-age,
4. vrijednosti Expires zaglavlja,
5. ukoliko ništa nije navedeno objekat se podrazumijevano kešira 120 sekundi.

Varnish posjeduje mehanizam za osvježavanje keša. U kešu se nalaze zaglavlja, kao i sadržaj odgovora web servera. Ukoliko se sadržaj u kešu nije promijenio, a vrijeme čuvanja objekta u kešu je isteklo, nema potrebe da se podaci ponovo povlače sa web servera i da se dodatno troše resursi i mrežni protok. HTTP rješava ovaj problem korištenjem uslovnih zahtjeva. Prvi način je na osnovu Etag zaglavlja u odgovoru. Etag HTTP zaglavlje se postavlja od strane web servera, odnosno aplikacije prilikom odgovora i sadrži jedinstvenu heš vrijednost koja odgovara stanju resursa. Heš vrijednost se dobija putem SHA ili MD5 algoritama na osnovu URL-a i datuma modifikovanja resursa. Može se dobiti i pozivom heš funkcije nad bilo kojim podacima koji su jedinstveni za traženi resurs. Klijent pamti vrijednost Etag zaglavlja iz odgovora. Prilikom sljedećeg poziva istog URL-a šalje se i vrijednost Etag zaglavlja, ali ovaj put kao vrijednost If-None-Match zaglavlja u zahtjevu. Server prihvata zahtjev i poredi da li se vrijednost razlikuje od Etag

vrijednosti koju bi ponovo trebalo proslijediti. Ukoliko je vrijednost Etag zaglavlja ista kao i vrijednost If-None-Match zaglavlja, aplikacija bi trebala da vrati u odgovoru *304 Not Modified* status, koji ukazuje na to da se vrijednost resursa nije promijenila. Kada se proslijedi *304* statusni kôd u odgovoru se ne šalje sadržaj. Kada klijent/pretraživač dobije u odgovoru status *304* zadržava prethodno povučene podatke i prikazuje ih korisniku. Ako vrijednost If-None-Match zaglavlja nije ista kao vrijednost Etag zaglavlja aplikacija će vratiti novi sadržaj resursa (eng. *payload*) i status *200 OK*, kao i novu vrijednost Etag zaglavlja. Ovo je dobar način da se izbjegne slanje suvišnih podataka ukoliko se koristan sadržaj nije mijenjao. Na ovaj način se značajno smanjuje količina saobraćaja koja se prosljeđuje nazad klijentu, ali i korištenje memorije i CPU-a na klijentskoj strani. Na sličnom principu radi i drugi način, koji se implementira putem Last-Modified zaglavlja. Web server ili aplikacija postavlja Last-Modified zaglavlje u odgovoru. Vrijednost Last-Modified zaglavlja sadrži datum i vrijeme modifikovanja resursa. Pretraživač pamti vrijednost Last-Modified zaglavlja iz odgovora. Prilikom sljedećeg poziva istog URL-a, pretraživač šalje ovu vrijednost u okviru If-Modified-Since zaglavlja u zahtjevu. Web server vrši poređenje vremena. Ukoliko se traženi podaci nisu mijenjali u odgovoru se vraća status *304*, pri čemu vrijednost Last-Modified zaglavlja ostaje ista. U slučaju da se razlikuju vremena prilikom poređenja, zahtijevani podaci su se promijenili. U odgovoru web servera nalazi se statusni kôd *200*, uključujući novi sadržaj i novu vrijednost Last-Modified zaglavlja. Rezultat korištenja je isti kao i kod prvog pristupa.

Varnish ima podršku za rad sa uslovnim zahtjevima. Ukoliko se u zahtjevu koji prihvati nalazi If-Modified-Since ili If-None-Match zaglavlje, Varnish prati vrijednosti Last-Modified ili Etag zaglavlja. Bez obzira da li Varnish posjeduje objekat u kešu ili ne, vraća status *304* ukoliko dođe do poklapanja vrijednosti If-Modified-Since/If-None-Match sa Last-Modified/Etag vrijednosti zaglavlja. Na ovaj način Varnish smanjuje količinu saobraćaja koja se prenosi ka klijentu, pod uslovom da nije došlo do promjene traženih podataka. Ukoliko je neki od objekata koji se nalazi u kešu zastario, a pri tom se u prethodnom odgovoru od web servera nalazilo Last-Modified ili Etag zaglavlje, Varnish šalje If-Modified-Since i If-None-Match zaglavlje ka web serveru. Kada web server vrati u odgovoru status *304 Not Modified*, Varnish ne dobija tijelo (eng. *body*) u odgovoru, što znači da se podaci nisu mijenjali. Na osnovu statusa 304 Varnish osvježava keš za traženi objekat, pri čemu objekat više nije zastario i vrijednost Age zaglavlja se postavlja na 0. Age zaglavlje označava koliko dugo se objekat nalazi u kešu i izraženo je u sekundama.

Postavljanjem polja beresp.keep u VCL fajlu utiče se na ponašanje Varnish servera pri radu sa uslovnim zahtjevima. Vrijednost beresp.keep polja se dodaje na *ttl*. Ovim se obezbjeđuje da Varnish izvršava uslovne zahtjeve asinhrono, tako da krajnji korisnik ne osjeti kašnjenje.

HTTP protokol je javno dostupan i isti je za sve korisnike. Ako su podaci korisnički zavisni ne bi se trebali keširati. Ipak, HTTP protokol nudi zaglavlje koje se koristi upravo za svrhe keširanja takvih podataka. Vary zaglavlje se šalje sa strane web servera i sadrži naziv drugog zaglavlja na osnovu kojeg se vrši keširanje korisnički zavisnih podataka. Na primjer, ukoliko pretpostavimo da se jezik nije naveo u URL-u ili putem kolačića za neku traženu stranicu, jedini preostali način je da se traženi jezik proslijedi kroz Accept-Language zaglavlje. Ukoliko prvi korisnik zahtijeva podatke na engleskom jeziku, nakon prvog pristupa svi ostali korisnici će dobiti takođe podatke koji su na engleskom jeziku, sve dok se objekat nalazi u kešu. Da bi izbjegli ovu situaciju kao vrijednost Vary zaglavlja navodi se Accept-Language zaglavlje. Nakon postavljenog Vary zaglavlja isti podaci će se keširati za različite vrijednosti Accept-Language zaglavlja. Na ovaj način omogućava se keširanje različitih varijanti, iako je isti URL. Treba voditi računa o tome šta se navodi u Vary zaglavlju. Ukoliko se navede User-Agent ili Cookie zaglavlje kao vrijednost Vary zaglavlja, može se javiti veliki broj različitih varijanti, upravo zato što kod različitih korisnika i uređaja vrijednost prethodna dva zaglavlja znatno varira. Ne bi trebalo keširati ovakve tipove zahtjeva. Cilj je izvršiti keširanje objekata koji će se često zahtijevati, odnosno onih koji će imati dobar *hit rate*. Varnish identifikuje objekte koji se nalaze u kešu koristeći heš vrijednosti ključa, koja se kreira na osnovu zahtijevanog URL-a. Na osnovu kreirane heš vrijednosti ključa Varnish vraća objekte iz keša.

Keširaju se zahtjevi sa GET ili HEAD metodama, koji ne sadrže kolačić i autorizaciono zaglavlje. Na osnovu odgovora od web servera Varnish odlučuje da li će se objekat sačuvati u kešu ili će se samo proslijediti klijentu.

Odgovor se kešira ako:

- *time-to-live* je veći od nule,
- odgovor ne sadrži Set-Cookie zaglavlje,
- Cache-control zaglavlje kao vrijednost ne sadrži: no-cache, no-store ili private,
- Vary zaglavlje ne sadrži *, što znači da bi se pravile varijante na osnovu svih zaglavlja.

Ako Varnish odluči da ne kešira neki objekat, stavlja ga na crnu listu (eng. *black list*) i ovaj keš se naziva *hit-for-pass* keš. U narednih 120 sekundi svi zahtjevi koji upućuju na isti objekat biće direktno prosljeđeni pozadinskom serveru, bez odlučivanja da li će se objekat keširati ili neće. Nakon 120 sekundi kada dođe novi zahtjev za istim objektom, ponovo se odlučuje o keširanju i prolazi se isti postupak. Vrijednost od 120 sekundi za hit-for-pass keš moguće je promijeniti putem VCL fajla.

VCL je domenski specifičan jezik koji se koristi za kontrolisanje ponašanja Varnish servera. Po sintaksi najsličniji je C, C++ i Perl jeziku. VCL se prevodi u C programski jezik i dinamički učitava u radno okruženje Varnish servera, prilikom učitavanja VCL fajla. Sa VCL jezikom se može uticati na tok izvršavanja aktivnosti (eng. *execution flow*). U VCL fajlu se piše kôd u predefinisanim podrutinama kojima se proširuje ponašanje Varnish servera. Svaka podrutina može da vrati vrijednost iz tačno definisanog skupa vrijednosti kojim se mijenja tok izvršavanja aktivnosti. Ukoliko se ne navede povratna vrijednost, Varnish će da izvrši podrazumijevani VCL kôd, pri čemu se neće izvršiti kôd koji je pisan za tu podrutinu.

VMOD moduli su Varnish moduli koji su pisani u programskom jeziku C i koji proširuju ponašanje Varnish servera i obogaćuju VCL sintaksu.

Ugrađene Varnish podrutine se mogu podijeliti u sljedeće grupe:

- klijentske podrutine,
- pozadinske podrutine i
- inicijalizacione i završne podrutine.

Klijentske podrutine su:

- vcl_recv – poziva se na početku svakog zahtjeva,
- vcl_pipe – prosljeđuje zahtjev direktno na pozadinski server i ubuduće svi podaci od klijenta i od pozadinskog servera se prenose neizmijenjeni sve dok se ne zatvori konekcija,
- vcl_pass – prosljeđuje zahtjev direktno na pozadinski server, odgovor se ne čuva u kešu,
- vcl_hit – poziva se kada se pronađe objekat u kešu,
- vcl_miss – poziva se kada objekat nije pronađen u kešu,

- vcl_hash – poziva se poslije *vcl_recv* za kreiranje heš vrijednosti koja se koristi kao ključ prilikom traženja objekta u kešu,
- vcl_purge – poziva se kada se objekat uspješno ukloni iz keša,
- vcl_deliver – poziva se na kraju zahtjeva kada je odgovor poslat klijentu i
- vcl_synth – vraća klijentu sintetički objekat, odnosno objekat koji nije poslat od strane pozadinskog servera, nego je iskreiran u VCL fajlu.

Kada se konekcija nalazi u *pipe* modu ne poziva se ni jedna druga podrutina. Varnish se ponaša kao TCP proxy.

Pozadinske podrutine su:

- vcl_backend_fetch – poziva se prije slanja zahtjeva ka pozadinskom serveru,
- vcl_backend_response – poziva se nakon uspješno primljenog odgovora od pozadinskog servera i
- vcl_backend_error – poziva se kada odgovor od pozadinskog servera nije uspješan.

Inicijalizacione i završne podrutine su:

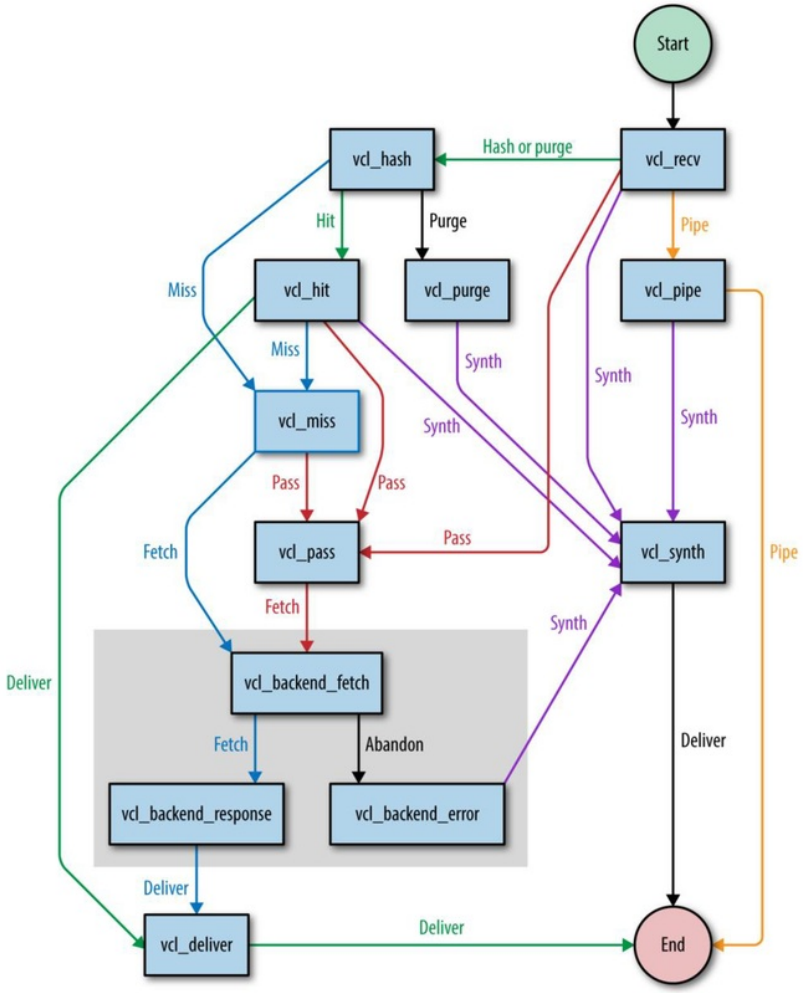
- vcl_init – poziva se kada je VCL učitan; koristi se i za inicijalizaciju Varnish modula i
- vcl_fini – poziva se kada je VCL izvršen; koristi se i za čišćenje Varnish modula.

Pored ugrađenih, korisnik može definisati svoje vlastite podrutine koje je moguće pozivati iz VCL koda. VCL podrutine predstavljaju različita stanja u kojim se Varnish može naći. Povratnim vrijednostima iz svake podrutine prelazi se u drugo stanje.

Povratne vrijednosti:

- hash – provjerava da li se objekat nalazi u kešu,
- pass – prosljeđuje se zahtjev pozadinskom serveru, pri čemu se rezultat ne kešira,
- pipe – prosljeđuje se zahtjev pozadinskom serveru i zaobilazi se bilo kakva logika vezana za keširanje,
- synth – zaustavlja se izvršavanje i vraća se sintetički odgovor, koji uključuje HTTP statusni kôd i poruku,
- purge – izbacuje objekat i njegove varijante iz keša,
- fetch – prosljeđuje se zahtjev ka pozadinskom serveru i kešira se odgovor,
- restart – ponovo se pokreće transakcija i uvećava se req.restarts brojač sve dok se ne dostigne *max_restarts*,

- deliver – vraća odgovor klijentu,
- miss – sinhrono se šalje zahtjev pozadinskom serveru i osvježava objekat, bez obzira da li se nalazi u kešu,
- lookup – koristi se heš vrijednost kako bi se provjerilo da li se objekat nalazi u kešu i
- abandon – odbacuje se odgovor od pozadinskog servera i prosljeđuje se HTTP greška *503 Service Unavailable*.



Slika 3.4 – Tok izvršavanja aktivnosti kod Varnish servera [8]

Moguće je podijeliti tok u dvije cjeline:

- rad sa pozadinskim serverom (sivi pravougaonik) i
- upravljanje sa zahtjevima i odgovorima (ostatak dijagrama).

Dijagram se dijeli na dva dijela, iz razloga što se rad sa pozadinskim serverom izvršava asinhrono. Zastarjeli objekat iz keša se vraća klijentu sve dok se ne osvježi sa novim objektom koji se dobija od pozadinskog servera. Ovo znatno ubrzava rad Varnish servera i ne pravi se red čekanja ukoliko je odgovor od pozadinskog servera spor.

Tok izvršavanja aktivnosti je sljedeći:

1. svaka sesija započinje sa *vcl_recv*,
2. traži se objekat u kešu – *vcl_hash*,
3. zahtjevi čiji se odgovor ne kešira direktno se prosljeđuju pozadinskom serveru u *vcl_pass* podrutini,
4. objektima koji su pronađeni u kešu se upravlja u *vcl_hit* podrutini,
5. ukoliko se objekat ne pronađe u kešu izvršava se *vcl_miss* podrutina,
6. putem podrutine *vcl_backend_fetch* obrađuju se zahtjevi kod kojih objekat nije pronađen u kešu ili ukoliko se ne vrši keširanje,
7. odgovor od pozadinskog servera se obrađuje u *vcl_backend_response* podrutini,
8. ukoliko odgovor od pozadinskog servera nije uspješan izvršava se *vcl_backend_error* podrutina,
9. uspješni odgovori, bez obzira da li je objekat pronađen u kešu ili je zahtjev direktno prosljeđen pozadinskom serveru, šalju se klijentu putem *vcl_deliver* podrutine.

U VCL fajlu navode se svi pozadinski serveri sa kojim će Varnish komunicirati. Prilikom definisanja pozadinskog servera moguće je navesti sljedeće atribute:

- host – *hostname* ili IP adresa pozadinskog servera,
- port – port na pozadinskom serveru koji će se koristiti, podrazumijevani port je 80,
- connect_timeout – vrijeme koje Varnish čeka da se uspostavi konekcija sa pozadinskim serverom, podrazumijevano 3.5 sekundi,
- first_byte_timeout – vrijeme koje Varnish čeka da se vrati prvi bajt od pozadinskog servera nakon inicijalne konekcije, podrazumijevano 60 sekundi,
- between_bytes_timeout – vrijeme koje Varnish čeka između slanja svaka dva bajta kako bi se osigurao tok podataka, podrazumijevano 60 sekundi,
- max_connections – maksimalan broj konekcija koje se uspostavljaju sa pozadinskim serverom,
- probe – koristi se za ispitivanje „zdravlja" (eng. *health*) pozadinskog servera, odnosno da li je pozadinski server u funkciji.

Ukoliko je ograničenje *max_connections* dostignuto, svaka sljedeća konekcija se odbacuje. Potrebno je provjeriti maksimalan broj konekcija koje pozadinski server može opslužiti istovremeno.

Ako bilo koji od definisanih kriterijuma nije ispunjen, generiše se greška pozadinskog servera i izvršavanje se prebacuje na *vcl_backend_error* podrutinu sa statusnim kodom *503 Service Unavailable*. Ukoliko se ne koristi ispitivanje „zdravlja" pozadinskog servera, u slučaju da dođe do prestanka rada nekog od pozadinskih servera, Varnish će i dalje slati zahtjeve ka neispravnom pozadinskom serveru. Za svaki pozadinski server *probe* atribut se navodi pojedinačno.

Lista podržanih atributa za *probe* je sljedeća:

- url – URL koji se poziva prilikom provjere ispravnosti pozadinskog servera, podrazumijevano „/",
- request – proizvoljan HTTP zahtjev koji se može slati na pozadinski server,
- expected_response – očekivani HTTP status kôd u odgovoru od pozadinskog servera, podrazumijevani kôd je *200 OK*,
- timeout – vrijeme koje se čeka na odgovor od pozadinskog servera, podrazumijevano 2 sekunde,
- interval – vremenski period za slanje zahtjeva pozadinskom serveru,
- window – broj zahtjeva u jednom opsegu koji se šalje pozadinskom serveru za utvrđivanje ispravnosti servera, podrazumijevano je 8,
- threshold – broj zahtjeva iz prethodno definisanog opsega (window) koji moraju biti uspješni kako bi se pozadinski server proglasio ispravnim/zdravim, podrazumijevano 3 i
- initial – inicijalni broj uspješnih zahtjeva za pozadinski server kada Varnish započne sa radom, prije nego što je pozadinski server proglašen kao ispravan, podrazumijevano -1.

Putem liste za kontrolu pristupa ACL (Access Control List) najčešće se ograničava pristup sadržajima na pozadinskom serveru. U ACL listama se navodi IP adresa, opseg IP adresa ili *hostname* sa kojeg se može pristupiti određenom sadržaju.

Postoje sljedeće VCL varijable:

- bereq – struktura podataka za zahtjev ka pozadinskom serveru,

- beresp – struktura podataka u odgovoru od pozadinskog servera,
- client – varijabla koja sadrži informacije o klijentskoj konekciji,
- local – informacije o lokalnoj TCP konekciji,
- now – informacija o trenutnom vremenu,
- obj – varijabla koja sadrži informacije o objektu koji se čuva u kešu,
- remote – informacije o udaljenoj TCP konekciji, u pitanju je klijent ili proxy koji se nalazi ispred Varnish servera,
- req – objekat zahtjeva,
- req_top – informacije o zahtjevu najvišeg nivoa u ESI (Edge Side Includes) stablu zahtjeva,
- resp – objekat odgovora,
- server – informacije o Varnish serveru i
- storage – informacije o mehanizmu za skladištenje.

Osnovni cilj keširanja podataka je zadovoljstvo krajnjih korisnika prilikom korištenja web sajta. Problem koji se može javiti je narušavanje integriteta podataka. Jedina gora situacija od keširanja objekata čiji *hit rate* nije dobar je keširanje objekata koji se često mijenjaju na duži vremenski period. Potrebno je voditi računa o tome šta se kešira i na koliki vremenski period. Ukoliko se podaci ne osvježavaju u skladu sa namjenom sajta, krajnji korisnici će izgubiti interesovanje i otići na konkurentski web sajt. Odličan primjer su novinski portali u trenutku objavljivanja udarne vijesti. Ako se u tom trenutku ne osvježi keš memorija, dobija se ista situacija kao da udarna vijest nije ni objavljena. Kako bi se izbjegle ovakve situacije vrši se oslobađanje keš memorije. U većini slučajeva dešava se da podaci u kešu zastare prije nego što su istekli. Varnish posjeduje mehanizme za oslobađanje keša i moguće ih je pozivati direktno iz koda aplikacije. Čišćenje (eng. *purging*) je jedan od najjednostavnijih mehanizama za oslobađanje keša. Neophodno je izmijeniti kôd u *vcl_recv* podrutini VCL fajla. Poželjno je koristiti ACL liste, kako bi se spriječilo čišćenje keš memorije sa bilo koje IP adrese. Potrebno je obraditi zahtjev metodom PURGE u VCL fajlu.

Čišćenje je prilično jednostavan metod za implementaciju zasnovanu na heš vrijednosti objekta. Drugi malo složeniji mehanizam za oslobađanje keš memorije je zabrana (eng. *banning*). Kod ovog mehanizma koriste se regularni izrazi kojim se označavaju objekti koji treba da se uklone iz keša. Zabranom se objekti stavljaju na listu zabrana (eng. *ban list*),

odnosno ne vrši se direktno oslobađanje memorije. Zabrana se vrši u trenutku kada se pronađe objekat u kešu, pod uslovom da se nalazi na listi zabrana. Oslobađanje memorije za objekte koji ne sadrže informacije o zahtjevu (npr. *obj* objekat) obavlja nit (eng. *thread*) koja se izvršava u pozadini (*ban lurker* thread). Svi ostali objekti iz keša se uklanjaju u trenutku zahtijevanja objekta. Ukoliko se pojavi veliki broj objekata na listi zabrana zasnovanih na *req* objektu, a kojima se ne pristupa tako često, mogao bi se pojaviti problem sa CPU performansama. Što je lista veća, procesoru je potrebno sve više vremena da je prođe prilikom svakog zahtijevanja objekta koji se nalazi u kešu. Iz ovih razloga preporučuje se korištenje *Lurker-Friendly Bans* mehanizma. *Ban lurker* nit se izvršava periodično i oslobađa keš memoriju Varnish servera na osnovu liste zabrana. Rad *ban lurker* niti je baziran na *obj* objektu, što je ograničavajući faktor prilikom oslobađanja keš memorije. *Lurker-Friendly Bans* mehanizam bazira se na istom principu kao i *ban lurker* mehanizam, s tim da se kopiraju informacije o zahtjevu iz *req* objekta. Ideja je da se host i URL iz zahtjeva dodaju u zaglavlju odgovora prilikom čuvanja objekata u kešu. Na ovaj način rješava se problem nedostatka informacija o zahtjevu, koji je imao prethodni mehanizam i oslobađanje keša se vrši asinhrono putem *ban lurker* niti.

Promjenom vrijednosti sljedeća tri parametra utiče se na funkcionisanje *ban lurker* niti:

- ban_lurker_age – najkraći vremenski period koji je potreban da objekat provede na listi zabrana kako bi bio uklonjen iz keša, podrazumijevana vrijednost je 60 sekundi,
- ban_lurker_batch – broj objekata koji se uklanjaju iz keša prilikom jednog izvršavanja *ban lurker* niti, podrazumijevana vrijednost je 1000 i
- ban_lurker_sleep – period izvršavanja *ban lurker* niti, podrazumijevana vrijednost je 0.010 sekundi.

Kao unapređenje prethodnog mehanizma moguće je navesti i regularni izraz koji bi se koristio za postavljanje objekata na listu zabrana. Ukoliko regularni izraz nije postavljen, rad se bazira na URL adresi. Sa komandom *varnishadm ban.list* moguće je provjeriti listu zabrana na Varnish serveru. Zabranu je moguće izvršiti i iz komandne linije, a ne samo putem koda u VCL fajlu.

Čišćenje i zabrana se koriste za uklanjanje objekata iz keš memorije, ali ne i za prikupljanje nove vrijednosti od pozadinskog servera. Postavljanjem varijable *req.hash_always_miss* na *true* u podrutini *vcl_recv* VCL fajla uvijek se povlači najnovija

vrijednost objekta, bez obzira da li je objekat u kešu istekao ili nije. Stari objekat se više ne koristi, ali će se nalaziti u kešu dok god ne istekne *ttl*. Postavljanjem *req.hash_always_miss* varijable na *true* ne vrši se oslobađanje memorije. Naprotiv, čuva se još jedan objekat u kešu. U kombinaciji ove varijable sa ACL listom dobija se mehanizam koji je potrebno koristiti za korisnike koji mijenjaju sadržaj web sajta. Na ovaj način se nakon izmjene sadržaja web sajta prikazuju najnoviji, a ne prethodno keširani podaci.

Postoje dva načina da se pozadinski serveri dodaju u rad Varnish servera:

- dodavanje pozadinskog servera u VCL fajlu i
- dodavanje pozadinskog servera prilikom pokretanja Varnish servera korištenjem *b* flega.

Ukoliko se pri radu Varnish servera koristi više pozadinskih servera, postoji potreba da se saobraćaj ka pozadinskim serverima kontroliše, odnosno da se definiše koji zahtjev se usmjerava na koji pozadinski server. U VCL fajlu, tačnije u podrutini *vcl_recv,* vrši se usmjeravanje saobraćaja na osnovu URL adrese iz zahtjeva. Saobraćaj se usmjerava na željeni pozadinski server dodjeljivanjem vrijednosti varijabli *req.backend_hint*.

Iako je osnovna namjena Varnish servera da ubrza rad web aplikacija (HTTP accelerator), istovremeno se može koristiti i za raspoređivanje opterećenja ka više pozadinskih servera. Menadžer/direktor (eng. *director*) je Varnish modul koji omogućava da se više pozadinskih servera posmatra kao jedan. Postoji više različitih strategija za određivanje koji od navedenih pozadinskih servera će obraditi klijentski zahtjev. Neke od strategija su opisane dalje u tekstu.

Korištenjem *director* Varnish modula postiže se raspodjela opterećenja, odnosno horizontalna skalabilnost. Takođe, ovaj modul vodi računa da se u raspodjelu saobraćaja uključuju samo „zdravi" serveri. Prva strategija za obradu klijentskih zahtjeva od strane pozadinskih servera je *Round-Robin*. Kod ove strategije svi serveri se smjenjuju u krug i raspodjela saobraćaja je ravnomjerna. Ovo može predstavljati problem ukoliko pozadinski serveri imaju različite performanse, što dovodi do situacije da se server sa najlošijim performansama opterećuje isto kao i server sa najboljim performansama. Ovaj problem se može riješiti upotrebom strategije *The Random Director*. Kod ove strategije pozadinskim serverima se dodjeljuje težina na osnovu koje se vrši raspodjela saobraćaja. Podrazumijevana težina kod svih servera je ista, što znači da se saobraćaj ravnomjerno raspoređuje ka svim pozadinskim serverima. Ukoliko se ne mijenja težina servera, u

principu dobijamo sličnu situaciju kao i sa *Round-Robin* strategijom. Nekim pozadinskim serverima je moguće dodijeliti veću težinu u odnosu na ostale, što znači da se saobraćaj usmjerava više na njih nego na druge pozadinske servere. Sljedeća strategija je *The Hash Director,* kod koje je izbor pozadinskog servera baziran na SHA256 heš algoritmu. Uzima se otisak proslijeđene vrijednosti u *backed()* metodi *director* objekta. Inače se prosljeđuje klijentska IP adresa ili sesijski kolačić (eng. *session cookie*). Ideja je da se saobraćaj za istog korisnika usmjerava na isti pozadinski server, ukoliko se podaci o sesiji čuvaju lokalno na pozadinskom serveru. Moguće je usmjeravati saobraćaj na osnovu URL adrese iz zahtjeva, pri čemu bi se svi zahtjevi ka istom URL-u usmjeravali na isti pozadinski server. Problem se može javiti ako se za heš vrijednost uzima IP adresa klijenta. Neki od pozadinskih servera može biti preopterećen ukoliko je IP adresa klijenta u stvari IP adresa proxy servera koju koristi više klijenata istovremeno. Slična situacija se može javiti i sa usmjeravanjem saobraćaja na osnovu URL adrese ukoliko je neka adresa više posjećena u odnosu na drugu, što bi opet rezultovalo preopterećenjem nekog od pozadinskih servera. Kao i kod prethodne strategije, i kod ove je moguće dodijeliti težinu serverima kako bi se upravljalo opterećenjem pozadinskog servera. Ukoliko promijenimo težinu nekog servera i dalje će se za istog korisnika, URL ili sesiju zahtjevi usmjeravati na isti pozadinski server, samo sa različitim opterećenjem. *The Fallback Director* strategija radi tako što saobraćaj usmjerava na „zdrav" pozadinski server. Potrebno je prilikom definisanja pozadinskog servera u VCL fajlu definisati i *probe* varijablu, u protivnom ova strategija neće raditi. Uzima se prvi navedeni server i ukoliko je „zdrav" saobraćaj se usmjerava na njega. U trenutku kada se pozadinski server proglasi kao „bolestan" uzima se sljedeći definisani pozadinski server i saobraćaj se usmjerava na njega. Moguće je kombinovati različite strategije. Kao primjer mogu se uzeti dva *Round-Robin* menadžera sa odgovarajućim serverima, koji se dodaju kao članovi *The Fallback Director* menadžera [8].

3.1.2 Klasterizacija Varnish servera

Varnish server predstavlja jedinu pristupnu tačku za klijentske zahtjeve. U slučaju da dođe do otkaza Varnish servera dolazi do otkaza i web aplikacije. Kako bi se spriječio otkaz postoji niz alata pomoću kojih se može izvršiti klasterizacija Varnish servera.

Putem Pacemaker alata obezbjeđuje se visoka dostupnost. Pacemaker alat upravlja resursima na Linux operativnom sistemu. Koristi se za detekciju i oporavak u slučaju da dođe do otkaza servisa ili instance u klasteru. Da bi se obezbijedilo upravljanje članstvom i

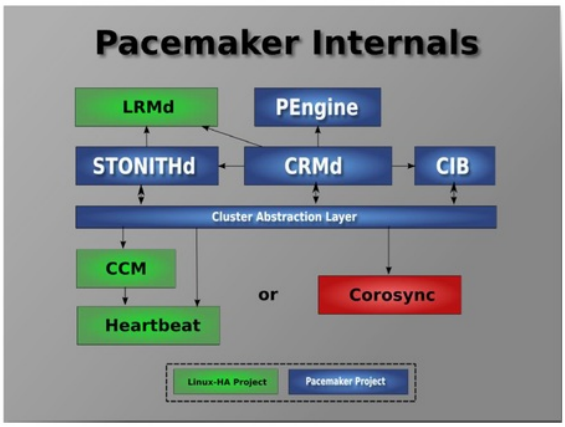
slanjem poruka u klasteru koriste se alati koji obezbjeđuju infrastrukturu klastera (npr. Corosync ili Heartbeat) [9].

Posmatrano sa višeg nivoa, *Pacemaker corosync* klaster se sastoji iz tri dijela:

- komponenata koje ne ovise o klasteru – resursi, skripte za pokretanje, zaustavljanje i monitoring klastera,
- dijela za upravljanje resursima – obrađuje događaje i izvršava akcije nad klasterom i
- nižeg nivoa infrastrukture – Corosync omogućava razmjenu poruka, vodi računa o članstvu u klasteru i kvorumu.

Pacemaker corosync klaster posjeduje sljedeće funkcionalnosti:

- grupnu komunikaciju, kojom se garantuje kreiranje replikacije sa istim stanjem mašine,
- ponovno pokretanje servisa ukoliko dođe do njegovog otkaza,
- bazu podataka koja se nalazi u memoriji i skladišti konfiguracione i statističke podatke i
- kvorum sistem, kojim se šalju notifikacije aplikaciji kada se postigne ili izgubi kvorum koji je neophodan za rad klastera [10].



Slika 3.5 – Veza između komponenti Pacemaker alata [9]

Pacemaker se sastoji od pet komponenti:

- Cluster Information Base (CIB),
- Cluster Resource Management daemon (CRMd),
- Local Resource Management daemon (LRMd),

- Policy Engine (PEngine ili PE) i
- Fencing daemon (STONITHd – Shoot-The-Other-Node-In-The-Head).

CIB koristi XML za prikaz konfiguracije i trenutnog stanja svih resursa u klasteru. Ovi podaci se sinhronizuju na cijelom klasteru i koriste od strane PEngine komponente za izračunavanje i postizanje najboljeg stanja klastera. Lista instrukcija se prosljeđuje Designated Controller-u (DC). Sve aktivnosti u klasteru se sprovode od strane jednog CRMd procesa na *master* instanci (odabranom kontroleru – DC). U slučaju otkaza CRMd procesa ili instance, u kratkom periodu se bira nova *master* instanca. DC izvršava instrukcije PEngine komponente u odgovarajućem redoslijedu prosljeđujući ih LRMd komponenti ili CRMd komponenti drugih instanci, putem infrastrukture za poruke. Proslijeđene instrukcije se obrađuju od strane LRMd komponente instance koja ih je prihvatila. Sve instance izvršavaju instrukcije i vraćaju odgovor DC komponenti. Na osnovu očekivanih rezultata izvršiće se potrebne aktivnosti ili će se prekinuti izvršavanje i ponovo uputiti zahtjev PEngine komponenti, da izračuna najbolje stanje klastera na osnovu neočekivanih rezultata.

U nekim slučajevima može doći do potrebe da se isključi neka instanca kako bi se zaštitili podaci ili dok se ne završi oporavak resursa. Za prethodno navedenu aktivnost je zadužena STONITHd komponenta. STONITHd komponenta je modelovana kao resurs i konfigurisana u CIB-u. Ona sprečava neispravne instance da mijenjaju podatke i vodi računa o istovremenom pristupu podacima. Ako neka od instanci ne komunicira sa ostatkom klastera, to ne znači da ona ne mijenja podatke drugih instanci. Jedini način da se osiguraju podaci jedne instance je korištenje STONITHd komponente, kojom se osigurava da je druga instanca isključena, sve dok se ne omogući pristup podacima željene instance. Još jedna uloga STONITHd komponente je da isključi instancu iz klastera u slučaju kada nije moguće zaustaviti servis na toj instanci, nakon čega bi se taj servis pokrenuo na nekoj drugoj instanci.

Postoje sljedeći tipovi klastera:

- aktiv-aktiv – aktivne su sve instance u klasteru; u slučaju da dođe do otkaza neke instance saobraćaj se usmjerava na druge instance u klasteru; moguća raspodjela saobraćaja,
- aktiv-pasiv – postoji rezervna pasivna instanca koja preuzima ulogu primarne instance u slučaju da primarna instanca otkaže,

- N+1 – kod ove konfiguracije jedna instanca je odgovorna za nekoliko aktivnih instanci, pri čemu preuzima ulogu instance koja otkaže,
- N+M – u slučajevima kada jedan klaster upravlja sa više servisa postojanje jedne rezervne instance nije dovoljno; koristi se više (M) instanci koje preuzimaju ulogu aktivnih instanci u slučaju otkaza,
- N-to-1 – rezervna instanca preuzima ulogu primarne instance u slučaju otkaza primarne instance, sve dok se primarna instanca ne oporavi i ne vrati u funkciju i
- N-to-N – kombinacija aktiv-aktiv i N+M klastera; sve instance su aktivne; u slučaju da dođe do otkaza neke od instanci saobraćaj se preusmjerava na druge instance u klasteru [11].

Pacemaker podržava rad sa svim tipovima klastera.

Jedna od mogućih varijanti klasterizacije Varnish servisa je upotreba aktiv-pasiv *Pacemaker corosync* klastera. Varnish servis bi se izvršavao na primarnoj instanci. Ukoliko bi došlo do otkaza primarne instance, rad Varnish servisa bi se prebacio na rezervnu instancu. Aktiv-pasiv klaster podržava rad Varnish servisa bez prekida. Na ovaj način povećava se dostupnost web sajta. Varnish servis predstavlja resurs u *Pacemaker corosync* klasteru.

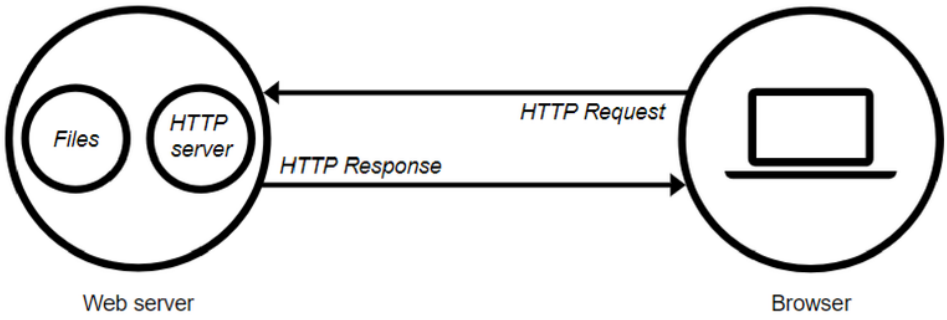
3.2 Web serveri

Web server je istovremeno i hardverska i softverska komponenta. Sa hardverske strane, web server predstavlja računar na kojem se nalazi softverska komponenta web servera, kao i komponente web stranice (npr. HTML, CSS, JavaScript fajlovi i slike). Web server je povezan na Internet, čime je omogućena komunikacija sa drugim uređajima na globalnoj mreži. Sa softverske strane, web server predstavlja softverske komponente koje omogućavaju krajnjim korisnicima da pristupe web stranici. HTTP server je dio softvera koji omogućava pristup web stranicama putem URL adresa, koristeći HTTP protokol. HTTP server prihvata zahtjev i vraća odgovor krajnjem korisniku [12].

Pojednostavljena komunikacija između klijenta (web pretraživača) i web servera se može objasniti na sljedeći način:

1. klijent zahtijeva fajl koji se nalazi na strani web servera putem HTTP protokola,
2. zahtjev stiže na odgovarajući web server (hardver),

3. HTTP server (softver) prihvata zahtjev, traži zahtijevani fajl i šalje odgovor klijentu putem HTTP protokola (ukoliko HTTP server ne pronađe zahtijevani fajl, klijentu se šalje greška *404 Not Found*).



Slika 3.6 – Pojednostavljena komunikacija između web servera i klijenta [12]

U zavisnosti od sadržaja koji se šalje klijentu, postoje sljedeća dva tipa web servera:

- statički web server i
- dinamički web server.

Statički web server klijentu šalje statičke fajlove koji odgovaraju zahtijevanom URL-u. Dinamički web server posjeduje istu funkcionalnost kao i statički web server, uključujući još dvije softverske komponente: aplikativni server i bazu podataka. Naziva se dinamički zbog toga što aplikativni server mijenja fajl sa podacima dobijenim iz baze podataka, prije nego što HTTP server pošalje fajl klijentu.

Web server treba imati: zadovoljavajuće performanse (performanse koje su neophodne za rad sa web stranicama koje se nalaze na tom serveru), statičku IP adresu i potrebno je da sve vrijeme bude dostupan.

Najpoznatija softverska rješenja web servera su: Apache HTTP server, Apache Tomcat, Internet Information Services (IIS), Nginx, lighttpd i drugi.

3.3 Relacione baze podataka

Relacione baze podataka imaju jednostavnu logičku organizaciju u formi kolekcije tabela. Redovi u tabelama odgovaraju entitetima realnog svijeta, pri čemu kolone predstavljaju svojstva entiteta. Zbog svoje jednostavne logičke organizacije tabele su intuitivne za korištenje. Ideja relacionog koncepta je odvajanje logičke od fizičke reprezentacije. Cilj je

da sistem može na različite načine da predstavi podatke, uz uslov da je obezbijeđeno korektno mapiranje na logički nivo. Drugi bitan motiv primjene relacionog koncepta bio je upotreba postojećeg matematičkog aparata za rad sa relacijama. Na osnovu ovog koncepta, razvijen je jezik visokog nivoa za manipulaciju podacima u relacionim bazama podataka. Bitna karakteristika ovog koncepta je da su operandi operatora koji se koriste u relacionom modelu cijele tabele, a ne pojedinačni redovi, pri čemu su rezultati izvršavanja operacija takođe tabele. Moguće je vršiti dodavanje novih redova, ažurirati ili brisati postojeće redove tabela, definisati upite na osnovu kojih se vrši izdvajanje potrebnih informacija iz tabela, spajati podatke iz različitih tabela i prezentovati ih korisniku. Relacione baze podataka su široko rasprostranjene [13].

SQL (Structured Query Language) je jezik visokog nivoa, namijenjen za rad sa relacionim bazama podataka. RDBMS (Relational Database Management System) je osnova za primjenu SQL jezika. Neke od najpoznatijih baza podataka baziranih na relacionom modelu su: MS SQL Server, IBM DB2, Oracle, MySQL, MariaDB itd.

3.3.1 CAP teorema

CAP (Consistency, Availability, Partition tolerance):

- konzistentnost – podrazumijeva da u istom trenutku svi čvorovi imaju iste podatke,
- raspoloživost – garancija da će se na svaki zahtjev dobiti odgovor,
- tolerancija dijeljenja mreže – sistem nastavlja da radi i u slučaju mrežnih prekida.

CAP teorema tvrdi da distribuirani sistemi mogu obezbijediti bilo koja dva, ali ne i sva tri prethodno navedena svojstva.

Pri radu sa bazama podataka mogu se javiti write-write i read-write konflikti. Write-write konflikti se javljaju kada dva korisnika pokušavaju da promijene iste podatke u istom trenutku. Do read-write konflikta dolazi kada korisnik čita nekonzistentne podatke dok ih drugi upisuje.

Konflikte je moguće riješiti na pesimističan ili optimističan način. Pesimističan način sprečava da dođe do konflikta, dok optimističan detektuje konflikte i rješava ih.

Kod distribuiranih sistema, do read-write konflikta dolazi kada samo dio instanci prihvati promjene. Konačna konzistentnost se postiže kada se upis podataka izvrši na svim

instancama. Postizanjem konačne konzistentnosti znatno se povećava vrijeme odziva sistema.

Na osnovu CAP teoreme, ukoliko dođe do dijeljenja mreže potrebno je izvršiti analizu i napraviti kompromis između konzistentnosti i dostupnosti podataka. Povećavanje dostupnosti postiže se replikacijom podataka na druge instance, ali se time povećava i vrijeme odziva sistema. Kako bi se smanjilo vrijeme odziva sistema, podaci se ne moraju kopirati na sve instance, ali je neophodno zadovoljiti kvorum o nivou usaglašenosti instanci. Ukoliko se replikacija podataka vrši na tri instance, zadovoljavajući nivo konzistentnosti je postignut ako se na dvije instance nalaze isti podaci [14].

Konzistentnost upisa podataka se postiže ako je zadovoljena sljedeća nejednakost: $W > N/2$, poznata kao kvorum pisanja. Broj instanci koje učestvuju pri upisu podataka (W) mora biti veći od polovine instanci uključenih u replikaciju (N). Broj replikacija se naziva faktor replikacije.

Slično kvorumu pisanja, postoji i kvorum čitanja. Kvorum čitanja se postiže ako je zadovoljena sljedeća nejednakost: $R + W > N$, odnosno zbir broja instanci koji je potrebno kontaktirati prilikom čitanja podataka i broja instanci koje potvrđuju pisanje mora biti veći od faktora replikacije (N).

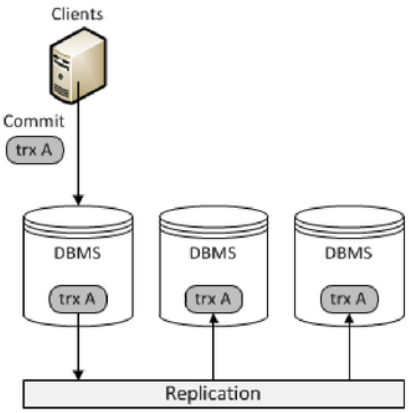
3.3.2 Klasterizacija servera baza podataka

Replikacija baze podataka se bazira na čestom kopiranju podataka sa jednog servera baza podataka na drugi. Klaster baza podataka predstavlja distribuiranu bazu podataka na više servera baza podataka, koji dijele informacije istog nivoa. Klaster nije vidljiv krajnjim korisnicima, ali im omogućava bolji odziv aplikacije u slučaju raspodjele opterećenja između više servera i dostupnost u slučaju da dođe do otkaza nekog od servera.

Postoje dva tipa replikacije baza podataka u zavisnosti od načina upisa podataka u bazu:

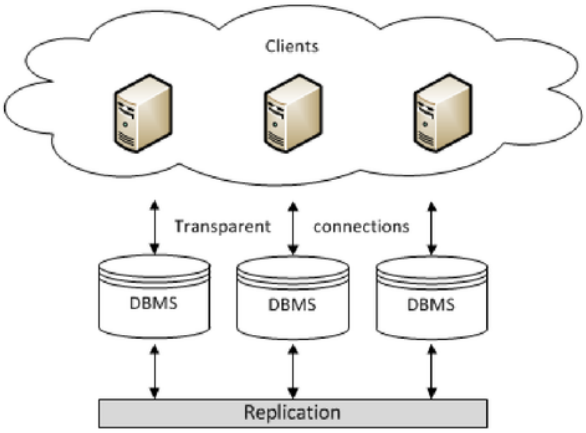
- master-slave replikacija i
- multi-master replikacija.

Kod master-slave replikacije, *master* server baza podataka prihvata izmjene na bazi podataka, nakon čega putem mreže prosljeđuje te izmjene ostalim *slave* serverima. *Slave* serveri baza podataka prihvataju izmjene i primjenjuju ih na svojoj bazi podataka.



Slika 3.7 – Master-slave replikacija [15]

Kod multi-master replikacije, izmjene na bazi podataka se mogu proslijediti bilo kojem serveru u klasteru. Server koji prihvati izmjene primjenjuje ih nad svojom bazom podataka i dalje ih prosljeđuje putem mreže ka ostalim serverima.



Slika 3.8 – Multi-master replikacija [15]

U zavisnosti od toga kako se informacije prosljeđuju između servera, u klasteru postoji:

- sinhrona replikacija i
- asinhrona replikacija.

Kod sinhrone replikacije, svi serveri baza podataka ažuriraju svoju bazu prilikom izmjene u jednoj transakciji. Kada se transakcija završi svi serveri imaju iste podatke. Kod asinhrone replikacije, izmjene na bazama svih servera u klasteru izvršavaju se postepeno, odnosno *master* server prosljeđuje izmjene drugim serverima. Kada proslijedi izmjenu jednom serveru, završava se jedna transakcija. Transakcije se završavaju u kratkom

vremenskom periodu. Za razliku od sinhrone replikacije, može doći do situacije da se na svim serverima u klasteru ne nalaze isti podaci.

Prednosti sinhrone replikacije:

- ne gube se podaci prilikom otkaza nekog od servera u klasteru, pri čemu se zadržava konzistentnost i postiže visoka dostupnost,
- sinhrona replikacija dozvoljava da se operacije izvršavaju na cijelom klasteru u paraleli, čime se postižu bolje performanse i
- rezultati upita su isti bez obzira na kojem serveru se izvršavaju.

Nedostatak sinhrone replikacije je što se povećanjem broja servera u klasteru eksponencijalno povećava:

- vrijeme izvršavanja transakcije,
- mogućnost da dođe do konflikta u pristupu podacima i
- vrijeme zaključavanja dok se ne izvrši transakcija, što rezultuje većim brojem potpunih zastoja (eng. *deadlock*).

Replikacija koja se bazira na sertifikaciji koristi tehnike grupne komunikacije i raspoređivanja transakcija, kako bi se postigla sinhrona replikacija. Ovakav vid replikacije se ne može implementirati na svim sistemima baza podataka.

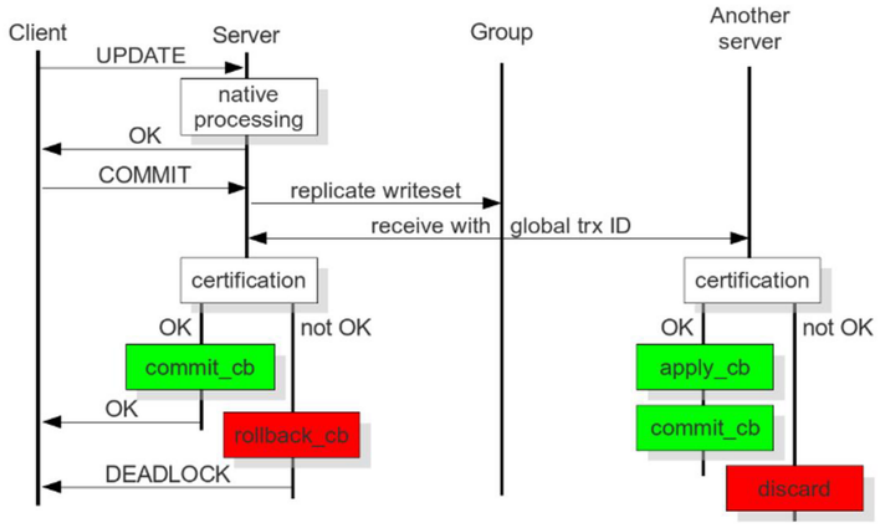
Da bi se mogla implementirati replikacija na bazi sertifikacije, system za upravljanje bazom podataka mora da ima sljedeće karakteristike:

- transakciono svojstvo – system za upravljanje bazom podataka ima sposobnost da vrati izmjene koje nisu potvrđene COMMIT naredbom,
- atomične promjene – promjene koje se izvršavaju prilikom replikacije atomično mijenjaju bazu podataka, odnosno niz operacija nad bazom podataka će se izvršiti ili se neće ništa dogoditi i
- globalno upravljanje – podrazumijeva upravljanje replikacionim događajima na globalnom nivou, odnosno da se događaji izvršavaju na svim instancama istim redoslijedom.

Replikacija bazirana na sertifikaciji radi na sljedećem principu:

1. klijent šalje izmjene ka bazi podataka,
2. izmjene se primjenjuju na bazi,
3. klijent šalje komandu COMMIT, kojom započinje potvrda izmjena,

4. prije nego što se potvrdi izmjena od strane servera ka klijentu, ostalim serverima u klasteru se šalju izmjene i primarni ključevi redova koji se mijenjaju (*replicate write-set*),
5. na svakom serveru *write-set* prolazi kroz sertifikacioni test kako bi se utvrdilo da li će se izmjene izvršiti ili neće,
6. ako sertifikacioni test ne prođe na nekom od servera, server odbacuje *write-set* i izmjene izvršene putem orginalne transakcije (od koje je sve započelo),
7. ako sertifikacioni test prođe uspješno, transakcija se potvrđuje i *write-set* se primjenjuje nad ostatkom klastera.



Slika 3.9 – Replikacija bazirana na sertifikaciji [15]

3.3.3 MariaDB baza podataka

MariaDB je *open source* relaciona baza podataka i trenutno je jedna od najpoznatijih i najzastupljenijih baza podataka. Razvijena je od strane istog tima koji je razvijao i MySQL. Neki od poznatih korisnika MariaDB baze podataka su: Wikipedia, WordPress.com i Google. MariaDB je brza, skalabilna i otporna na otkaze. Podržava rad sa GIS (Geographic Information System) i JSON (JavaScript Object Notation) podacima. Predstavlja unapređenje i koristi se kao zamjena za MySQL [16]. Sve što postoji u MySQL postoji i u MariaDB (komande, interfejsi, biblioteke, API). Ukoliko se prelazi sa MySQL na MariaDB DBMS, nema potrebe za bilo kakvim konvertovanjem baza podataka, pošto koriste istu arhitekturu i strukturu koda. Unapređenja koja MariaDB DBMS nudi u odnosu

na MySQL su: više mehanizama za skladištenje podataka, poboljšanje brzine, nove funkcionalnosti, bolje testiranje, manje grešaka i u potpunosti je *open source* [17].

3.3.4 Galera klaster

Galera klaster je sinhroni multi-master klaster baza podataka, baziran na sinhronoj replikaciji i Oraklovoj MySQL/InnoDB. Putem Galera klastera moguće je vršiti upis i čitanje podataka na bilo kojem serveru u klasteru. U slučaju otkaza nekog od servera, klaster nastavlja da funkcioniše bez bilo kakvog zastoja u radu. Galera klaster se koristi za klasterizaciju MySQL i MariaDB baza podataka.

Galera klaster ima sljedeće osobine:

- multi-master – upis i čitanje podataka je moguće na bilo kojem serveru u bilo kojem trenutku,
- sinhrona replikacija – ne gube se podaci prilikom otkaza nekog od servera,
- čvrsto povezani serveri – svi serveri imaju isto stanje i podatke,
- korištenje više niti prilikom upisa podataka na ostale servere – radom u paraleli postižu se bolje performanse,
- nema master-slave operacija koje se izvršavaju prilikom otkaza ili korištenja VIP (Virtual IP) adrese,
- nema zastoja prilikom otkaza nekog od servera,
- nema potrebe za ručnim kopiranjem baze i prebacivanjem na drugi server,
- podržava InnoDB mehanizam za rad sa podacima, koji nudi visoke performanse i pouzdanost,
- zahtijeva veoma male ili čak nikakve izmjene na aplikaciji i
- nema potrebe za razdvajanjem čitanja i upisa podataka.

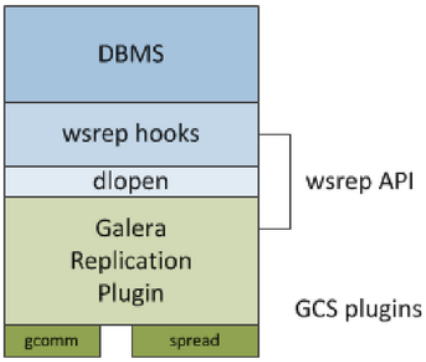
Korištenjem Galera klastera postižu se visoke performanse i dostupnost, bez značajnog zastoja prilikom otkaza nekog od servera. Dodatno se postiže i očuvanje integriteta podataka.

Kod Galera klastera, replikacija bazirana na sertifikaciji se zasniva na globalnom raspoređivanju transakcija. Prilikom replikacije, svaka transakcija dobija svoj broj na osnovu redoslijeda izvršavanja (eng. *sequence number*). Kada dođe do potvrde izmjena, putem sertifikacionog testa i *write-set* podataka, klaster upoređuje trenutnu transakciju sa posljednjom uspješnom. Ukoliko se u tom periodu, između dvije globalno raspoređene

transakcije pojavi bilo koja druga transakcija, dolazi do pada testa. Pad testa je uzrokovan konfliktom između primarnih ključeva. Postupak je deterministički i svi serveri u klasteru prihvataju transakcije istim redoslijedom. Na osnovu toga, svi serveri imaju istu odluku o ishodu transakcije. Server koji je započeo transakciju obavještava klijentsku aplikaciju o tome da li je uspješno izvršena ili nije.

Interna arhitektura Galera klastera posjeduje sljedeće komponente:

- DBMS (Database Management System),
- wsrep API (Write Set Replication API),
- Galera Replication Plugin i
- Group Communication Plugins.



Slika 3.10 – Replikacioni API [15]

DBMS predstavlja server baza podataka koji se izvršava na svakoj instanci u klasteru. Putem *wsrep API-ja*, Galera klaster obezbjeđuje replikaciju baziranu na sertifikaciji. *Wsrep API* posmatra šemu baze podataka i podatke kao njeno stanje. Klijent izvršavanjem transakcija koje predstavljaju niz atomičnih operacija mijenja njeno stanje. Da bi se održalo konzistentno stanje na svim instancama u klasteru, *wsrep API* koristi Global Transaction ID (GTID) za svaki *write-set*. GTID se koristi za identifikovanje i poređenje trenutnog stanja prilikom transakcije i prethodnog stanja baza podataka. Potrebno je da sve instance u klasteru imaju isto stanje. GTID se sastoji iz dva dijela:

- UUID – kojim se identifikuje stanje i
- Ordinal Sequence Number – 64-bitni označeni cijeli broj koji predstavlja redni broj promjene.

Primjer GTID-a: 45eec521-2f34-11e0-0800-2a36050b826b:94530586304.

Galera Replication Plugin implementira *wsrep API*, pri čemu zajedno čine wsrep provajdera (*wsrep Provider*). *Wsrep hooks* se koriste za integraciju sa serverom baza podataka kako bi se omogućila replikacija. Putem funkcije *dlopen()* omogućena je veza između *wsrep hooks* i wsrep provajdera. Replikacija se bazira na GTID identifikatoru i *write-set* skupu podataka.

Galera Replication Plugin se sastoji iz tri komponente:

- Certification Layer – sertifikacioni sloj koji priprema *write-set* i izvršava sertifikacioni test,
- Replication Layer – replikacioni sloj koji upravlja procesom replikacije i globalnog raspoređivanja transakcija i
- Group Communication Framework – pruža dodatnu arhitekturu za povezivanje sa različitim sistemima za grupnu komunikaciju.

Group Communication Plugins su grupni sistemi za komunikaciju koji su dostupni Galera klasteru (npr. gcomm i Spread).

Promjena stanja baze podataka postiže se izvršavanjem sljedećih koraka:

1. na neku od instanci u klasteru dolazi zahtjev od klijenta, odnosno dolazi do izmjene na bazi podataka kojom započinje promjena stanja,
2. wsrep provajder koristeći *wsrep hooks* kreira *write-set* skup podataka na osnovu izmjena,
3. putem *dlopen()* funkcije ostvaruje se veza između wsrep provajdera i *wsrep hooks,*
4. *Galera Replication Plugin* izvršava *write-set* sertifikaciju i replikaciju podataka na svim instancama u klasteru.

U sistemima baza podataka konkurentne transakcije se izvršavaju „izolovane“ jedne od drugih. Nivo izolacije zavisi od njihovog međusobnog uticaja.

Postoje sljedeći nivoi izolacije:

- READ-UNCOMMITTED,
- READ-COMMITTED,
- REPEATABLE-READ i
- SERIALIZABLE.

Kod READ-UNCOMMITED nivoa izolacije dolazi do „prljavog“ čitanja (eng. *dirty read*). Moguće je da transakcije vide izmjene nad bazom podataka od strane drugih transakcija, koje još uvijek nisu potvrđene (eng. *uncommited changes*). Transakcije koje prave izmjene nad bazom podataka uvijek mogu da vrate prethodne podatke, a da ne izvrše potvrdu.

Kod READ-COMMITED nivoa izolacije ne može doći do „prljavog“ čitanja. Izmjene nad bazom podataka nisu vidljive drugim transakcijama, sve dok ih transakcija koja ih je izvršila ne potvrdi. Ukoliko se odabere ovaj nivo izolacije može doći do *non-repeatable* čitanja. Kada se izvršava SELECT upit dobijaju se podaci koji su potvrđeni prije pokretanja SELECT upita. Ako se u okviru iste transakcije više puta uzastopno izvršavaju SELECT upiti za pristup istim podacima, mogu se vratiti različiti podaci ukoliko je između SELECT upita došlo do promjene istog skupa podataka od neke druge transakcije koja je u međuvremenu potvrdila svoje izmjene. *Non-repeatable* čitanje se dešava kada transakcija koja se izvršava čita potvrđene izmjene izvršene od strane druge transakcije. Isti redovi mogu imati različite i potvrđene vrijednosti od trenutka kada je transakcija započela pa do trenutka završetka transakcije.

REPEATABLE-READ nivo izolacije ne dozvoljava da dođe do *non-repeatable* čitanja. Tokom izvršavanja transakcije koristi se isti set podataka koji se dobije prilikom prvog pokretanja SELECT upita. Kod ovog nivoa izolacije ne uzimaju se u obzir izmjene podataka koje su izvršile druge transakcije, bez obzira da li su potvrđene ili nisu.

Kod SERIALIZABLE nivoa izolacije, svi redovi koji se koriste tokom izvršavanja neke transakcije su zaključani. Onemogućeno je i dodavanje novih ili brisanje postojećih redova u tabeli, sve dok se transakcija ne završi. Ovim nivoom serijalizacije sprečava se *phantom* čitanje. Do ovakvog čitanja dolazi ukoliko se u okviru transakcije izvrše dva identična upita, pri čemu se podaci koje vrati drugi upit razlikuju od rezultata koje vraća prvi upit. *Phantom* čitanje je slično *non-repeatable* čitanju, s tim da transakcija čita podatke koje je neka druga transakcija dodala ili obrisala i pri tome potvrdila izmjene. Suštinski, postoje novi redovi ili su neki obrisani od trenutka kada je transakcija započela.

Multi-master Galera klaster radi samo sa REPEATABLE-READ nivoom izolacije.

Postoje dvije metode u Galera klasteru putem kojih je moguće izvršiti replikaciju podataka:

- State Snapshot Transfers (SST) – replikacija stanja jedne instance i

- Incremental State Transfers (IST) – replikacija samo onih transakcija koje nedostaju.

SST metodom vrši se kopiranje svih podataka sa jedne instance na drugu. Kada se pojavi nova instanca u klasteru, ona inicira SST metodu kojom započinje sinhronizaciju podataka sa instancom koja se već nalazi u klasteru.

Putem IST metode, umjesto da se šalje cijelo stanje, klaster identifikuje samo one transakcije koje nedostaju i samo se one prosljeđuju instanci koja se pridružila klasteru.

IST metoda se može izvršiti pod sljedećim uslovima:

- UUID stanja instance koja se pridružuje je isti kao kod ostalih instanci u klasteru i
- sve transakcije koje nedostaju se nalaze u *write-set* kešu instance sa kojom se vrši sinhronizacija.

Ako su ispunjeni prethodno navedeni uslovi, može započeti sinhronizacija nove, ili instance koja se ponovo pridružila klasteru, sa instancom koja je već dio klastera. Ukoliko se klasteru ponovo pridružuje instanca sa GTID-om: 5a76ef62-30ec-11e1-0800-dba504cf2aab:197222, pri čemu je stanje klastera: 5a76ef62-30ec-11e1-0800-dba504cf2aab:201913, instanca sa kojom se vrši sinhronizacija prihvata zahtjev od instance koja se pridružuje klasteru i provjerava da li se redni broj promjene 197222 nalazi u njenom *write-set* kešu. Ukoliko se redni broj promjene nalazi u kešu, instanca koja je prihvatila zahtjev šalje sve promjene od 197222 do 201913 instanci koja je inicirala zahtjev, u protivnom se inicira početak SST metode.

Prednost IST metode je što se znatno ubrzava vraćanje instance u klaster, pri čemu taj proces ne blokira rad instance sa kojom se vrši sinhronizacija.

Galera klaster čuva *write-set* u posebnoj keš memoriji koja se naziva *Write-set Cache* ili *GCache*. Putem parametra *gcache.size* moguće je mijenjati veličinu prostora koji se alocira u sistemskoj memoriji za skladištenje *write-set* skupa podataka. Potrebno je voditi računa da veličina alociranog prostora ne bude veća od same baze podataka, čime bi IST metoda postala manje efikasna od SST metode.

GCache posjeduje sljedeća tri tipa memorije za skladištenje *write-set* skupa podataka:

- Permanent In-Memory Store – alocira se RAM memorija,

- Permanent Ring-Buffer File – alocira se memorija na disku prilikom inicijalizacije keša, podrazumijevana veličina je 128MB i
- On-Demand Page Store – ukoliko bude potrebno, alocira se memorija na disku u toku izvršavanja, pri čemu je veličina memorije za skladištenje ograničena slobodnim prostorom na disku.

Algoritam za alokaciju memorije pokušava sačuvati *write-set* u memoriji prethodno navedenim redoslijedom. *Permanent In-Memory Store* tip memorije je podrazumijevano onemogućen. Za skladištenje *write-set* skupa podataka podrazumijevano se koristi *Permanent Ring-Buffer File* tip memorije.

Kod Galera klastera procesom replikacije upravlja mehanizam koji se naziva *Flow Control*. Putem ovog mehanizma, instanca u klasteru može da zaustavi i ponovo nastavi proces replikacije u zavisnosti od svojih potreba. Na ovaj način se sprečava da instanca koja je inicirala sinhronizaciju kasni prilikom izvršavanja nadolazećih transakcija, u odnosu na ostale instance u klasteru.

Instanca prihvata *write-set* skupove podataka i organizuje ih na osnovu globalnog redoslijeda izvršavanja. Na ovaj način se osigurava da će sve instance izvršiti transakcije istim redoslijedom i imati isto stanje. Transakcije koje prihvati instanca u klasteru, a koje nisu započele za izvršavanjem i nisu potvrđene, smještaju se u red za izvršavanje. Kada red dostigne određenu veličinu aktivira se *Flow Control* mehanizam. Instanca zaustavlja replikaciju i započinje sa izvršavanjem transakcija iz reda. Kada se veličina reda smanji na veličinu koja je adekvatna za dalje upravljanje, instanca nastavlja sa procesom replikacije podataka.

Instanca se može naći u sljedećim stanjima: OPEN, PRIMARY, JOINER, JOINED, SYNCED i DONOR.

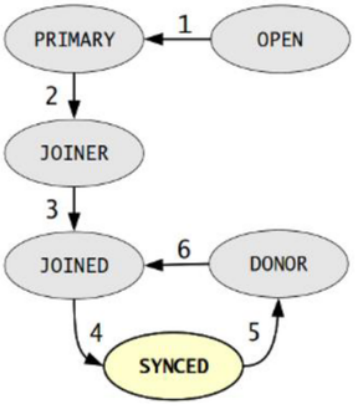
U zavisnosti od stanja instance, postoje četiri osnovne vrste *Flow Control* mehanizma:

- No Flow Control – instance nisu dio klastera i nalaze se u OPEN ili PRIMARY stanju. Zbog toga im nije dozvoljeno da vrše replikaciju, primjenjuju ili keširaju *write-set* skup podataka.
- Write-set Caching – instance su dio klastera i nalaze se u JOINER ili DONOR stanju. Ne mogu primjenjivati *write-set* skupove podataka, keširaju ih za naknadnu upotrebu.

- Catching Up – instance su u JOINED stanju i mogu primijeniti *write-set* skup podataka na svojoj bazi podataka. Na ovaj način se oslobađa *GCache* memorija. Kako je primjena *write-set* skupa podataka nad bazom podataka inače nekoliko puta brža od izvršavanja transakcije, instance u ovom stanju jedva da utiču na performanse klastera.
- Cluster Sync – instance se nalaze u SYNCED stanju.

Promjene stanja instance u Galera klasteru se odvijaju u sljedećim koracima (Slika 3.11):

1. instanca se pokreće i uspostavlja komunikaciju sa primarnom komponentom klastera (eng. *Primary Component*),
2. nakon što uspješno prođe zahtjev za transfer stanja, instanca započinje keširanje *write-set* skupova podataka,
3. instanca je završila sa SST metodom, posjeduje sve podatke iz klastera i započinje sa primjenjivanjem keširanih *write-set* podataka nad svojom bazom podataka,
4. instanca je pridružena klasteru i može započeti sa izvršavanjem transakcija,
5. instanca prihvata zahtjev za SST metodom od druge instance, pri čemu kešira sve *write-set* skupove podataka koje ne može primijeniti nad svojom bazom,
6. instanca završava sa SST metodom, pri čemu druga instanca koja je zahtijevala SST posjeduje sve podatke iz klastera.



Slika 3.11 – Promjena stanja instance u Galera klasteru [15]

U slučaju da dođe do otkaza nekog dijela servera, greške u sistemu, gubitka mrežne konekcije ili greške prilikom promjene stanja, instanca više nije dio klastera.

Instanca nije dio klastera ukoliko izgubi vezu sa primarnom komponentom. Iz perspektive klastera, kada instanca iz primarne komponente ne vidi neku instancu, ta instanca više nije dio klastera.

Klaster utvrđuje vezu sa nekom instancom na osnovu posljednjeg mrežnog paketa koji je primio od te instance. Moguće je konfigurisati period provjere od strane klastera putem parametra *evs.inactive_check_period*. Ukoliko prilikom provjere klaster utvrdi da je period od posljednjeg primljenog mrežnog paketa od strane neke istance veći od vrijednosti parametra *evs.keepalive_period*, klaster započinje sa slanjem *heartbeat* paketa. Ukoliko instanca kojoj se šalju paketi ne vrati odgovor u periodu definisanom putem *evs.suspect_timeout* parametra, ta instanca se proglašava suspendovanom. Kada sve instance iz primarne komponente proglase neku instancu suspendovanom, ta instanca se proglašava neaktivnom. Ukoliko se u periodu definisanom putem parametra *evs.inactive_timeout* ne dobije odgovor od instance, ta instanca se proglašava za neaktivnu bez obzira da li su se sve instance usaglasile ili nisu.

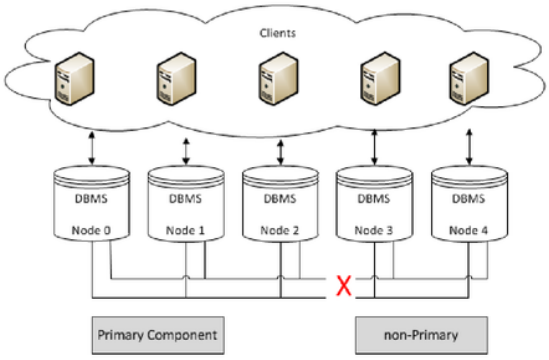
Galera klaster garantuje sigurnost i konzistentnost podataka. Na osnovu CAP teoreme dolazi do kompromisa između raspoloživosti i tolerancije dijeljenja mreže. Do problema može doći ukoliko se definišu niske vrijednosti za parametre *evs.suspect_timeout* i *evs.inactive_timeout*, pri čemu se instanca može proglasiti neaktivnom, iako je još uvijek aktivna. Takođe, problem mogu predstavljati i veće vrijednosti ovih parametara ukoliko dođe do otkaza neke od instanci. U ovoj situaciji se produžuje vrijeme potrebno za detektovanje greške.

Klaster nastavlja sa radom ukoliko dođe do otkaza neke od instanci. Kada se instanca ponovo vrati u klaster, ona sinhronizuje svoje podatke sa ostalim instancama iz klastera i tek nakon toga postaje ponovo dio klastera.

Ukoliko dođe do dijeljenja mreže, dolazi i do dijeljenja klastera u više komponenti. Komponenta predstavlja grupu instanci koje međusobno komuniciraju. U ovakvim situacijama samo jedna komponenta može mijenjati stanje baze podataka. Ta komponenta se naziva primarna komponenta i ona u suštini predstavlja klaster. Kada dođe do dijeljenja mreže, Galera klaster pokreće algoritam kojim se utvrđuje primarna komponenta. Ovim algoritmom se garantuje da će u klasteru uvijek biti samo jedna primarna komponenta.

Veličina klastera se mijenja dodavanjem ili uklanjanjem instanci iz klastera. Na osnovu veličine klastera definiše se broj glasova potrebnih za postizanje kvoruma prilikom odabira primarne komponente.

Kada dođe do dijeljenja mreže nastaju dvije komponente, pri čemu samo jedna postiže kvorum i postaje primarna komponenta, dok druga pokušava da uspostavi konekciju ka primarnoj komponenti.



Slika 3.12 – Podjela klastera na dvije komponente, od kojih je jedna primarna [15]

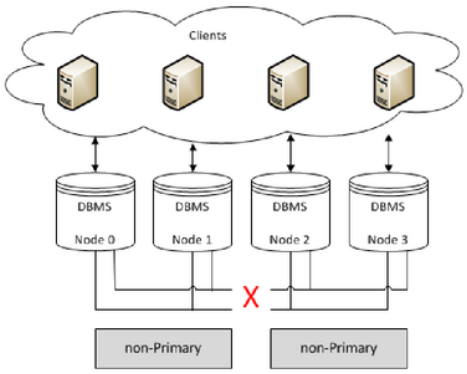
U slučaju da dođe do podjele na dvije komponente koje imaju isti broj instanci, ni jedna od njih neće biti izabrana za primarnu komponentu. U ovakvim situacijama Galera klaster omogućava da se izvrši konfiguracija klastera na taj način da se uvijek izabere primarna komponenta. Galera klaster može instancama dodijeliti težinu koja se koristi pri računanju kvoruma putem sljedeće formule:

$$\frac{\sum_{p_i \times w_i} - \sum_{l_i \times w_i}}{2} < \sum_{m_i \times w_i},$$

gdje je:

- p_i – broj instanci posljednje primarne komponente,
- l_i – broj instanci koje su napustile klaster,
- m_i – broj trenutnih instanci i
- w_i – težina instanci.

Kvorum je postignut ako je suma težina instanci nove komponente veća od polovine težine instanci prethodne primarne komponente. Težina instance se može mijenjati putem parametra *pc.weight*. Podrazumijevana težina je 1.



Slika 3.13 - Podjela klastera na dvije komponente koje nisu primarne [15]

3.4 NoSQL baze podataka

Za razliku od tradicionalnih baza podataka, NoSQL (*Not only SQL*) se može definisati kao kolekcija različitih koncepata za skladištenje i manipulaciju podacima. Potreba za novim konceptom se javila usljed naglog porasta količine podataka i potrebe za skladištenjem nestrukturisanih podataka. Velike količine podataka generišu se putem društvenih mreža, geoinformacionih sistema, sistema sa senzorima, aplikacija za praćenje logova i niza drugih sistema. NoSQL baze podataka odlikuju dobre performanse, skalabilnost i fleksibilnost.

Pošto su NoSQL baze podataka bez šeme (eng. *schemaless*), pružaju slobodu i fleksibilnost prilikom skladištenja podataka. Čuvaju se samo podaci od interesa. Za razliku od relacionih baza podataka zapisi mogu imati različita polja. Na ovaj način rješava se problem tabela čije kolone uobičajeno imaju *null* vrijednosti. Neophodno je voditi računa o načinu na koji se podaci snimaju, da ne bi došlo do problema prilikom pristupa podacima iz iste baze podataka od strane više različitih korisnika [14].

Kod relacionog modela podataka vizuelna predstava podataka je u vidu tabela. NoSQL baze podataka se udaljavaju od relacionog modela.

Mogu se grupisati u četiri različite kategorije:

- ključ-vrijednost baze podataka (eng. *key-value store*) – dozvoljavaju čuvanje bilo kakvih podataka kojima se pristupa putem ključa,
- baze podataka orijentisane ka dokumentima (eng. *document store*) – rade na sličnom principu kao i ključ-vrijednost baze podataka, s tim da uvode određena ograničenja na strukturu dokumenta koji se čuva u bazi,

- kolonski orijentisane baze podataka (eng. *wide column store*) – dozvoljavaju čuvanje bilo kakvih podataka u okviru bilo koje kolone i
- baze podataka orijentisane ka grafovima (eng. *graph store*) – dozvoljavaju da se slobodno dodaju nove veze između čvorova, kao i da se dodaju svojstva samim čvorovima.

3.4.1 Redis baza podataka

Redis je jedna od najzastupljenijih ključ-vrijednost baza podataka. Ključ-vrijednost baze podataka predstavljaju jednostavne heš tabele. Konkretnoj vrijednosti nekog zapisa pristupa se putem ključa. Vrijednost koja se snima u bazu može biti obični tekst, JSON, XML ili bilo šta drugo.

Kod Redis baze podataka vrijednost koja se čuva nije striktno definisana kao string, već može biti i neka složenija struktura podataka. Ključevi su binarno sigurni, što znači da se za ključ može koristiti bilo koja binarna sekvenca, od stringa do JPEG datoteke. Potrebno je voditi računa o dužini ključa. Prazan string je takođe validan ključ. Predugi ključevi nisu dobri prilikom poređenja jer se troši mnogo vremena, a pri tome zauzimaju puno memorije. Potrebno je naći balans između dužine ključa i njegove primjene. Prilikom kreiranja zapisa poželjno je pridržavati se sljedećeg pravila za definisanje ključa: *tip-objekta:id* (npr. korisnik:1). Crtica ili dvotačka se često koriste za razdvajanje riječi. Maksimalna dužina ključa je 512MB [18].

Strukture podataka koje podržava Redis su:

- binary-safe stringovi,
- liste – najčešće su to povezane liste i predstavljaju kolekciju string elemenata poredanih po redoslijedu dodavanja elemenata u listu,
- skupovi – kolekcija jedinstvenih, nesortiranih elemenata tipa string,
- sortirani skupovi – svakom elementu je pridružen broj (sa decimalnom tačkom), koji se naziva skor; elementi se sortiraju po skoru (moguće je dobiti određeni niz elemenata, npr. prvih 10 ili posljednjih 10 elemenata iz skupa),
- heševi (eng. *hashes*) – predstavljaju mape koje se sastoje od polja kojima su pridružene vrijednosti (polja i vrijednosti su stringovi),
- nizovi bitova (eng. *bitmaps*) – kod ove strukture podataka koriste se specijalne komande za upravljanje sa string vrijednostima kao nizom bitova (moguće je:

postaviti ili obrisati pojedine bitove, prebrojati sve bitove postavljene na 1, pronaći prvi bit postavljen na 1 ili na 0 i drugo) i

- HyperLogLogs – struktura podataka koja se koristi za određivanje kardinalnosti skupa [18].

3.4.2 Redis klaster

Redis klaster predstavlja distribuiranu implementaciju Redisa koja posjeduje visoke performanse, linearnu skalabilnost do 1000 instanci, prihvatljiv stepen sigurnosti upisa podataka i osigurava dostupnost [19]. Distribuirani Redis podržava sve operacije sa jednim ključem, kao i nedistribuirani Redis. Kompleksnije operacije sa više ključeva, kao što su unija ili presjek skupova, podržane su i kod distribuiranog Redisa, ali pod uslovom da se svi ključevi nalaze na jednoj instanci. Redis klaster radi na bazi heš tagova tako da se određeni ključevi mogu sačuvati na istoj instanci. Može se desiti da prilikom ručnog unosa rekorda u bazu dođe do situacije kada operacije koje rade sa više ključeva nisu dostupne, za razliku od operacija sa jednim ključem.

Kod nedistribuiranog Redisa, komandom SELECT se mijenjaju baze podataka sa kojima klijent radi. Redis klaster ne podržava rad sa više baza podataka. Postoji samo jedna baza (database 0) i nije moguće koristiti komandu SELECT.

Redis klaster vodi računa o pronalasku novih instanci, detekciji neispravnih, kao i proglašavanju *slave* (rezervnih) instanci za *master* instance.

Sve instance u Redis klasteru su međusobno povezane putem *Redis Cluster Bus* protokola, baziranom na *TCP bus* i *binary* protokolu. Klijent može uputiti zahtjev na bilo koju instancu u klasteru. Po potrebi će se preusmjeriti na odgovarajući čvor. Performanse se mogu poboljšati ako je klijent u mogućnosti da vodi evidenciju o parovima ključ-instanca.

Redis klaster koristi asinhronu replikaciju između čvorova. Sa posljednje izabrane *master* instance podaci se asinhrono kopiraju na *slave* instance. Podaci iz Redis klastera mogu se izgubiti na dva načina. Kod prvog načina dolazi do upisa podataka u *master* instancu, ali usljed otkaza ne dolazi do asinhronog upisa podataka u *slave* instance. *Slave* instanca postaje izabrana za novu *master* instancu i podaci koji su prethodno bili upisani u staru *master* instancu zauvijek se gube. Kod drugog načina, usljed otkaza *master* instance, *slave* instanca preuzima njenu ulogu. Nakon nekog vremena *master* instanca koja je prethodno

otkazala postaje ponovo aktivna. Usljed zastarjele tabele rutiranja na strani klijenta, može se desiti da dođe do upisa podataka u staru *master* instancu, prije nego što je proglašena *slave* instancom, pri čemu se gube podaci.

Ukoliko posmatramo Redis klaster sa N *master* instanci, gdje svaka *master* instanca ima po jednu *slave* instancu, klaster će biti dostupan u slučaju da otkaže jedna instanca. Klaster ostaje dostupan sa vjerovatnoćom od 1-(1/(N*2-1)), gdje je 1/(N*2-1) vjerovatnoća otkaza *master* instance koja je ostala bez svoje *slave* instance.

Redis klaster posjeduje svojstvo koje se naziva *migracija replika* (eng. *replicas migration*). Ukoliko se u klasteru nalazi dovoljan broj instanci, ovo svojstvo omogućava da pri svakom otkazu neke od instanci u klasteru dođe do rekonfiguracije klastera, tako da *master* instanca koja je ostala bez svojih *slave* instanci dobije novu *slave* instancu. Dovoljan broj instanci podrazumijeva da svaka *master* instanca ima bar jednu *slave* instancu, pri čemu bar jedna od *master* instanci mora imati više od jedne *slave* instance.

Redis klaster instancama dodjeljuje opseg ključeva za koji su odgovorne. Po potrebi, instance mogu da vrše redirekciju na drugu instancu na osnovu proslijeđenog ključa. Takođe, prilikom izvršavanja komandi klijent na bazi ključa bira odgovarajuću instancu. Prilikom asinhronog upisa podataka sa *master* instance na *slave* instance, ne čeka se odgovor od *slave* instanci, ukoliko se to eksplicitno ne navede korištenjem WAIT komande. WAIT komanda se navodi nakon komande za upis i čeka se na potvrdu od *slave* instanci. Prilikom navođenja WAIT komande, specifikuje se minimalan broj *slave* instanci koje moraju vratiti potvrdu i *timeout* u milisekundama. *Timeout-om* se prekida izvršavanje komande, čak iako nije odgovorio traženi broj *slave* instanci. Ukoliko se navede *timeout* 0 čeka se neograničen vremenski period. Kao odgovor WAIT komande dobija se broj *slave* instanci koje su potvrdile upis.

Skaliranje kod Redis klastera je linearno, što znači da su performanse Redis klastera sa N *master* instanci iste kao i performanse N zasebnih Redis instanci. Osnovni cilj Redis klastera je postizanje skalabilnosti i veoma visokih performansi, uz zadovoljavajući nivo sigurnosti i dostupnosti.

Opseg ključeva podijeljen je na 16384 heš slota, što znači da bi teoretski Redis klaster podržao rad sa 16384 *master* instance. U praksi se preporučuje rad sa maksimalno 1000 *master* instanci. Svaka *master* instanca u Redis klasteru je odgovorna za određeni opseg

heš slotova. Za klaster se može reći da je stabilan ako nije došlo do procesa rekonfiguracije. Prilikom rekonfiguracije opseg slotova se pomjera sa jedne instance na drugu. U trenutku snimanja vrijednosti u Redis klaster, na osnovu ključa koji odgovara toj vrijednosti, računa se slot u koji će se smjestiti zapis ključ-vrijednost. Pravilo za računanje vrijednosti slota je: *HASH_SLOT = CRC16(key) mod 16384*. CRC16 predstavlja *Cyclic Redundancy Check 16-bitni algoritam*. Pošto svaka *master* instanca odgovara za određeni opseg slotova, nakon izračunate vrijednosti HASH_SLOT, tačno je definisano na koju *master* instancu se smješta zapis. Postoji izuzetak pri računanju HASH_SLOT vrijednosti koji se javlja pri radu sa operacijama koje koriste više ključeva. Potrebno je da se svi ključevi čuvaju na istom heš slotu. Uvedeno je sljedeće pravilo: ako ključ sadrži otvorenu i zatvorenu vitičastu zagradu „{...}", sve što se nalazi između prvog pojavljivanja otvorene vitičaste zagrade i zatvorene, koja se nalazi nakon otvorene, koristi se za računanje heš vrijednosti na osnovu koje se dobija slot u koji će se smjestiti zapis. Primjer: ključevi „{skup}.prvi" i „{skup}.drugi" će imati istu HASH_SLOT vrijednost zato što se za računanje heš slota koristi podstring „skup", a ne cijela vrijednost ključa. Ukoliko se navede ključ „{}skup.prvi", cijeli string se koristi za računanje HASH_SLOT vrijednosti.

Svaka instanca u Redis klasteru ima jedinstveno ime, odnosno identifikator (ID), koji se dodjeljuje prilikom prvog pokretanja instance. ID predstavlja proizvoljno generisani string od 160 bita koji se čuva u konfiguracionom fajlu. Ponovo se generiše jedino u situacijama kada sistem administrator obriše konfiguracioni fajl ili kada dođe do poziva naredbe za reset klastera CLUSTER RESET. ID instanca je jedinstvena i globalna. Koristi se za identifikaciju instanci na cijelom klasteru. Za ID instance vežu se određene informacije. Neke od tih informacija odnose se na detalje konfiguracije Redis klastera za specifičnu instancu (npr. kada je posljednji put prošao uspješno *ping* ka specifičnoj instanci).

Svaka instanca čuva sljedeće informacije o drugim instancama:

- ID instance,
- IP adresu i port,
- da li je instanca *master* ili *slave*,
- posljednje vrijeme kada je ka instanci upućena PING naredba,
- posljednje vrijeme kada je dobijen odgovor (PONG) od naredbe PING,
- trenutni *configuration epoch* instance,
- stanje linka i

- opseg heš slot vrijednosti.

Ove informacije se mogu dobiti na svakoj od instanci koje se nalaze u klasteru izvršavanjem komande CLUSTER NODES.

Svaka instanca u Redis klasteru posjeduje jedan dodatni TCP port (klaster bus port) koji se koristi za komunikaciju sa ostalim instancama. Vrijednost Redis klaster porta se dobija tako što se na TCP port koji se koristi za komunikaciju sa klijentima doda *offset* od 10000. Primjer: ukoliko instanca osluškuje klijentske konekcije na portu 7000, klaster bus port će biti 17000 i podrazumijevano je otvoren.

Sve instance u klasteru su međusobno povezane putem TCP konekcije (*full mesh* topologija). U Redis klasteru od N instanci, svaka instanca ima N-1 odlazećih i N-1 dolazećih konekcija od drugih instanci. Konekcije između instanci su aktivne sve vrijeme dok je aktivan klaster. Za komunikaciju koriste *gossip* protokol.

Instanca prihvata drugu instancu kao dio klastera jedino u sljedeća dva slučaja:

1. korištenjem MET poruke ili
2. putem *gossip* poruka.

MET poruka je slična PING poruci, s tim da instanca koja primi MET poruku prihvata instancu koja je poslala poruku u klaster. Instance šalju MET poruku drugim instancama jedino u slučaju kada sistem administrator to zahtijeva izvršavanjem komande CLUSTER MET ip port.

Primjer dodavanja instance u klaster putem *gossip* poruka:
ukoliko instanca A poznaje instancu B, a pri tom instanca B poznaje instancu C, instanca B šalje *gossip* poruku instanci A o instanci C. Kada se pošalje *gossip* poruka, instanca A registruje instancu C kao dio mreže i pokušava uspostaviti komunikaciju s njom.

Ukoliko je komunikacija između instanci sigurna, sistem administrator može omogućiti drugu opciju. Na ovaj način se postiže automatsko međusobno povezivanje instanci.

Postoje dva tipa redirekcije:

1. MOVED i
2. ASK.

Klijent može slati upite na bilo koju instancu. Instanca koja prihvati zahtjev provjerava da li može odgovoriti na upit, odnosno da li ključ ili ključevi odgovaraju njenim heš slotovima. Ukoliko odgovaraju, instanca vraća odgovor za izvršeni upit. U protivnom, provjerava internu evidenciju o ostalim instancama u klasteru i njihovom opsegu slotova. Kada pronađe instancu koja može odgovoriti na traženi upit vraća *MOVED error* odgovor klijentu.

Primjer:

```
127.0.0.1:7001> get kljuc
(error) MOVED 5120 192.168.15.244:7003
```

U odgovoru se nalazi: heš slot koji odgovara ključu i ip_adresa:port instance koja može odgovoriti na upit. Nakon odgovora, klijent treba da uputi zahtjev na instancu koja se nalazila u *MOVED error* odgovoru. Ukoliko klijent ne uputi zahtjev odmah nakon odgovora, može se desiti da u međuvremenu dođe do rekonfiguracije klastera, što znači da bi instanca koja prihvati novi zahtjev mogla vratiti isti odgovor *MOVED error*. Slična situacija se može desiti ukoliko instanca nema najnovije informacije o ostalim instancama u klasteru. Nakon što se prihvati *MOVED error,* poželjno je osvježiti informacije o ostalim instancama u klasteru komandama CLUSTER NODES ili CLUSTER SLOTS.

Prilikom rada Redis klastera moguće je dodavati nove ili uklanjati postojeće instance. Dodavanje ili uklanjanje instanci bazirano je na premještanju heš slotova sa jedne instance na drugu. Prilikom dodavanja nove instance, u klaster se dodaje „prazna" instanca, pri čemu se dio heš slotova sa postojeće instance prebacuje na novu. Usljed uklanjanja instance iz klastera, heš slotovi instance koja se uklanja prebacuju se na neku drugu instancu u klasteru. Moguće je vršiti i prebacivanje heš slotova između dvije postojeće instance. Sve operacije su bazirane na premještanju heš slotova između instanci.

Komande koje se koriste za manipulaciju klasterom su:

- CLUSTER ADDSLOTS slot1 [slot2] ... [slotN],
- CLUSTER DELSLOTS slot1 [slot2] ... [slotN],
- CLUSTER SETSLOT slot NODE node,
- CLUSTER SETSLOT slot MIGRATING node,
- CLUSTER SETSLOT slot IMPORTING node.

Sa komandama ADDSLOTS i DELSLOTS moguće je dodati ili ukloniti slotove neke instance. Nakon što se neki slotovi dodaju instanci, informacija se putem *gossip* protokola širi na cijelom klasteru. Komanda ADDSLOTS se obično koristi pri samom kreiranju klastera, kako bi se svakoj *master* instanci dodijelio neki opseg od ukupno dostupnih 16384 heš slotova. U praksi komanda DELSLOTS je slabo zastupljena i koristi se prilikom ručne modifikacije klastera ili prilikom otklanjanja grešaka.

SETSLOT komandom je moguće dodijeliti slot instanci. Pri radu sa SETSLOT komandom postoje dva stanja: MIGRATING i IMPORTING, koja se koriste za migraciju heš slotova sa jedne instance na drugu. Kada se slot označi sa MIGRATING, instanca će prihvatiti sve upite za navedeni heš slot ukoliko se ključ još uvijek nalazi u njenom slotu ili će proslijediti zahtjev korištenjem ASK redirekcije ka instanci koja je navedena kao odredište prilikom migracije. Ako se slot označi sa IMPORTING, instanca će prihvatiti sve zahtjeve pod uslovom da je zahtjevima prethodila ASKING komanda. Ukoliko klijent nije poslao ASKING komandu, upit se prosljeđuje ka instanci koja posjeduje heš slot, putem MOVED redirekcije.

Primjer:
ako posmatramo dvije *master* instance A i B i želimo migrirati slot 2 sa instance A na instancu B, potrebno je izvršiti sljedeće komande:

- na instanci B: CLUSTER SETSLOT 2 IMPORTING A,
- na instanci A: CLUSTER SETSLOT 2 MIGRATING B.

Sve ostale instance će usmjeravati klijenta ka instanci A svaki put kada se šalje zahtjev sa ključem koji odgovara slotu 2.

U tom slučaju postoje dvije opcije:

1. ukoliko instanca A posjeduje ključ izvršava sve upite,
2. sve upite izvršava instanca B, zato što je instanca A izvršila redirekciju ka instanci B.

Nakon izvršenih gore navedenih komandi, novi ključevi se kreiraju na instanci B. U međuvremenu, putem programa *redis-trib*, koji se koristi za rekonfiguraciju i konfiguraciju Redis klastera, vrši se migriranje postojećih ključeva u heš slotu 2 sa instance A na instancu B. Komandom *CLUSTER GETKEYSINSLOT slot n* vraća se maksimalno n ključeva iz navedenog slota. Korištenjem prethodno navedene komande, *redis-trib*

program dobija spisak svih ključeva koji odgovaraju zahtijevanom slotu i komandom MIGRATE vrši migraciju svakog ključa i vrijednosti koja odgovara ključu sa instance A na instancu B. Operacija migracije ključeva je atomična, što znači da su obje instance zaključane prilikom migracije ključeva i da ne može doći do nekonzistentnosti podataka. Naredba za migraciju: MIGRATE target_host target_port key target_database id timeout.

Naredba MIGRATE nije striktno vezana za Redis klaster, već se može koristiti i za druge potrebe. Prethodno je navedeno da Redis klaster ne može raditi sa više baza, tako da nema potrebe postavljati target_database na neku drugu vrijednost različitu od 0. Kako je neophodno da sve instance u klasteru budu informisane o novoj konfiguraciji klastera, nakon što se završi proces migracije komandom *SETSLOT slot NODE node-id* šalje se informacija o novoj lokaciji slota svim instancama u klasteru, uključujući i instance koje su učestvovale u migraciji.

MOVED redirekcijom klijent dobija informaciju da se traženi ključ nalazi na drugoj instanci i da bi svi naredni zahtjevi trebalo da budu upućeni ka toj instanci. ASK redirekcijom klijentu se sugeriše da samo sljedeći zahtjev treba uputiti ka navedenoj instanci. Ovo je potrebno zbog situacije kada klijent u procesu migracije zahtijeva ključ koji se još uvijek nalazi na instanci A. Potrebno je osigurati da klijent prvo uputi zahtjev ka instanci A koja može odgovoriti na traženi upit, ukoliko ključ nije izmigriran na instancu B. Ukoliko je došlo do migracije ključa na instancu B za slot koji je na instanci B označen sa IMPORTING, pod uslovom da je zahtjevu prethodila ASKING komanda, instanca B će odgovoriti na traženi upit.

Sa stanovišta klijenta, ASK komanda radi na sljedeći način:

1. ukoliko je klijent dobio ASK redirekciju kao odgovor, samo sljedeći upit se šalje na navedenu instancu, a svi ostali upiti se šalju na staru instancu,
2. redirekcioni upit mora započeti sa ASKING komandom,
3. još uvijek se ne vrši ažuriranje lokalne klijentske tabele,
4. nakon što je završena migracija cijelog slota 2 sa instance A na instancu B, sljedeći zahtjev koji dođe od strane klijenta na instancu A se preusmjerava putem MOVED redirekcije,
5. nakon MOVED redirekcije klijent dobija informaciju da sve naredne upite vezane za taj slot treba uputiti ka instanci B i vrši ažuriranje svoje lokalne klijentske tabele, tako da se slot 2 nalazi na novoj instanci (nova IP i port).

Potrebno je da klijent popuni svoju lokalnu tabelu na početku, a ne da nasumično šalje zahtjeve ka instancama i postepeno popunjava tabelu, što bi znatno usporilo njegov rad i učinilo ga neefikasnim.

Klijenti obično imaju potrebu da sačuvaju mapiranja slotova i adresa instanci u sljedeća dva slučaja:

- na početku sa inicijalnom konfiguracijom (aktuelnom konfiguracijom u tom trenutku) i
- kada u odgovoru dobiju MOVED redirekciju.

Problem se može javiti usljed premještanja više slotova odjednom (npr. ukoliko se *slave* instanca proglasi *master* instancom). U takvim slučajevima jednostavnije je pokupiti ponovo sva mapiranja umjesto da se tabela popunjava u skladu sa MOVED redirekcijom. Pored komande CLUSTER NODES postoji još jedna komanda CLUSTER SLOTS, koja omogućava da se popuni klijentska tabela bez potrebe da se vrši parsiranje. Prethodno navedenom komandom dobijaju se samo informacije koje su neophodne za klijenta. Kao rezultat CLUSTER SLOTS komande dobija se spisak svih opsega slotova, pri čemu se za svaki opseg dobije informacija o *master* instanci, kao i o svim *slave* instancama koje posjeduju taj opseg.

Primjer:

```
127.0.0.1:7003> CLUSTER SLOTS
1) 1) (integer) 5461
   2) (integer) 10922
   3) 1) "192.168.15.242"
      2) (integer) 7001
      3) "ea3886a9939ce3d1cdd53d59907033c4b33357da"
   4) 1) "192.168.15.245"
      2) (integer) 7004
      3) "be0ac0347f70b07f8285a940762d1ef89fc811f0"
2) 1) (integer) 0
   2) (integer) 5460
   3) 1) "192.168.15.244"
      2) (integer) 7003
      3) "f008530b1593674f73d23e779bf26ca85d47bf17"
```

```
   4) 1) "192.168.15.241"
      2) (integer) 7000
      3) "3f3d93f4a0e31e9478e7b99895523cfe2545b745"
3) 1) (integer) 10923
   2) (integer) 16383
   3) 1) "192.168.15.243"
      2) (integer) 7002
      3) "2153b99cc4f4b4b17dbc5965c1aa1d6dbbd0cf49"
   4) 1) "192.168.15.246"
      2) (integer) 7005
      3) "cbce3628580127e6340c3a74acc448ec2c23c564"
```

U prethodnom primjeru, svaki element predstavlja jedan opseg slotova i posjeduje podelemente kod kojih prva dva označavaju početak i kraj opsega, treći daje informaciju o *master* instanci, dok ostali podelementi nose informacije o *slave* instancama. Informacije koje se prikazuju za *master* i *slave* instance su: IP adresa, port i ID instance.

Ukoliko klaster nije dobro konfigurisan može doći do situacije kada nije pokriven cijeli opseg od ukupno 16384 slota. Neophodno je da klijenti nakon poziva komande CLUSTER SLOTS popune nedostupne slotove sa *null* vrijednostima, kako bi mogli vratiti odgovor krajnjem korisniku, ukoliko dođe do izvršavanja upita sa ključem koji nije pridružen niti jednom slotu. Prije nego što se vrati odgovor o grešci, neophodno je da klijent još jednom pokupi konfiguraciju cijelog klastera, kako bi bio siguran da u međuvremenu nije došlo do rekonfiguracije klastera i otklanjanja problema.

Ukoliko prilikom migracije ključeva dođe do upita koji podržava rad sa više ključeva, može se desiti da se ključevi u tom trenutku ne nalaze na istoj instanci, odnosno da su podijeljeni između dvije instance. Kao odgovor na prethodnu situaciju dobija se *TRYAGAIN error*, pri čemu klijent može ponovo uputiti zahtjev nakon nekog vremena ili vratiti grešku krajnjem korisniku. Nakon što se završi migracija za zahtijevani slot, biće moguće ponovo izvršavati operacije koje podržavaju rad sa više ključeva.

Ako se od *slave* instance zahtijeva izvršavanje nekog upita doći će do redirekcije na njenu *master* instancu. Ukoliko klijent želi da izvrši skaliranje pri čitanju podataka tako što će uputiti zahtjev ka *slave* instanci, potrebno je da prije željenog upita izvrši komandu

READONLY. READONLY komandom *slave* instanci se govori da klijent nije zainteresovan da izvrši upis podataka, kao i da može doći do čitanja „zastarjelih" podataka.

Kada se koristi READONLY mod, klaster može odgovoriti klijentu sa redirekcijom pod sljedećim uslovima:

- klijent je poslao komandu *slave* instanci vezanu za slot koji nikad nije bio u opsegu njene *master* instance,
- klaster je rekonfigurisan i *slave* instanca više nije u mogućnosti da odgovori na operaciju za traženi slot.

U slučaju da dođe do redirekcije, neophodno je da klijent ponovo osvježi svoju lokalnu tabelu. Moguće je prekinuti READONLY mod izvršavanjem komande READWRITE.

Instance u Redis klasteru razmjenjuju *ping* i *pong* pakete. Ovi paketi su slične konstrukcije i nose informacije o konfiguraciji instanci u klasteru. Jedina razlika su im tipovi polja. *Heartbeat* paketi predstavljaju sumu *ping* i *pong* paketa. Instance šalju *ping* pakete i očekuju odgovor od druge instance putem *pong* paketa. Moguća je i opcija gdje instance šalju samo *pong* pakete, koji nose informacije o njihovoj konfiguraciji, bez da očekuju odgovor. Ovakve poruke se inače šalju u vidu *broadcast* poruka u slučaju da instanca ima novu konfiguraciju. *Ping* i *pong* paketi sadrže zaglavlje koje je zajedničko za sve tipove paketa, kao i *gossip* sekciju koja je specifična samo za njih.

Zajedničko zaglavlje sadrži sljedeća polja:

- ID instance – pseudoslučajni 160-bitni string koji se generiše prilikom kreiranja instance i isti je sve dok klaster postoji, ukoliko ne dođe do promjene od strane sistem administratora,
- currentEpoch i configEpoch – ako je instanca *slave,* njen *configEpoch* je isti kao posljednji *configEpoch master* instance,
- flegove – označavaju da li je instanca *slave* ili *master*, s tim da nose i neke druge jednobitne informacije o instanci,
- bitmapu heš slotova instance koja šalje pakete – ukoliko je instanca *slave,* šalje se bitmapa heš slotova njene *master* instance,
- TCP port instance koja šalje pakete – port koji se koristi za prihvatanje klijentskih komandi,
- stanje klastera iz perspektive pošiljaoca – da li je klaster aktivan ili nije i

- ID *master* instance – ukoliko je pošiljalac *slave* instanca.

Gossip sekcija nosi informacije o nekoliko proizvoljnih instanci koje su poznate pošiljaocu. Za svaku instancu koja se dodaje u prethodno navedenu sekciju šalju se sljedeća polja:

- ID instance,
- IP adresa i port instance i
- flegovi.

Gossip sekcija je korisna za uočavanje grešaka ili otkrivanje novih instanci u klasteru. Uočavanje grešaka u Redis klasteru svodi se na detekciju *master* ili *slave* instance koja nije dostupna većini instanci u klasteru. U slučaju da dođe do pada *master* instance, potrebno je *slave* instancu proglasiti za novu *master* instancu. Ukoliko nije moguće proglasiti novu *master* instancu, klaster prelazi u stanje greške. Kada se klaster nalazi u stanju greške ne mogu se prihvatati zahtjevi od strane klijenata. Svaka instanca posjeduje listu flegova za svaku od njoj poznatih instanci.

Sljedeća dva flega su bitna za uočavanje greške pri radu klastera:

- PFAIL (possible failure) – označava moguću grešku i
- FAIL – označava da je došlo do greške i da je to stanje potvrdila većina *master* instanci u određenom vremenu.

Master ili *slave* instanca postavlja fleg PFAIL za drugu instancu ukoliko nije dostupna više od definisanog NODE_TIMEOUT vremenskog perioda. Da bi ovaj mehanizam pravilno funkcionisao, NODE_TIMEOUT period mora biti prihvatljiv u odnosu na broj instanci i vrijeme koje je potrebno paketu prilikom putovanja kroz mrežu.

Da bi se *slave* instanca proglasila za *master* instancu, potrebno je da bude postavljen FAIL fleg.

PFAIL prelazi u FAIL ukoliko se zadovolje sljedeći uslovi:

- instanca A označi instancu B sa PFAIL,
- instanca A putem *gossip* protokola dobija informacije o instanci B od većine ostalih *master* instanci u klasteru i

- većina *master* instanci označi instancu sa PFAIL ili FAIL u okviru NODE_TIMEOUT * FAIL_REPORT_VALIDITY_MULT vremena (faktor validnosti FAIL_REPORT_VALIDITY_MULT je podrazumijevano 2).

Ukoliko su svi prethodno navedeni uslovi ispunjeni, instanca A će označiti FAIL fleg instance B i proslijediti FAIL poruku svim dostupnim instancama. Sve instance koje dobiju FAIL poruku označiće navedenu instancu sa FAIL u svojoj lokalnoj evidenciji, bez obzira da li je za tu instancu prethodno bio postavljen PFAIL fleg.

PFAIL fleg može preći u FAIL fleg, dok se FAIL fleg može obrisati samo pod sljedećim uslovima:

- instanca je *slave* i dostupna je,
- instanca je *master* i nema dodijeljen opseg heš slotova ili
- instanca je *master* i ponovo je dostupna, a pri tom je prošao duži vremenski period u kojem se *slave* instanca nije proglasila za *master* instancu (u tom slučaju bolje je pustiti dostupnu *master* instancu u rad, kako bi klaster nastavio sa radom).

U slučaju da više instanci sadrži različite informacije, instanca koja prihvata informacije može na osnovu polja *currentEpoch* utvrditi koje su najnovije i koje će prihvatiti. Polje *currentEpoch* predstavlja 64-bitni neoznačeni broj. Prilikom pokretanja instanci u Redis klasteru i *master* i *slave* instance postavljaju polje *currentEpoch* na 0. Kada instanca prihvati paket od neke druge instance, provjerava da li je *currentEpoch* veći od njenog. Ukoliko jeste, usvaja vrijednost polja *currentEpoch* pošiljaoca, a isto tako i vrijednosti ostalih informacija.

Izbor i proglašavanje nove *master* instance vrše *slave* instance, u trenutku kada je *master* instanca u stanju FAIL iz aspekta barem jedne od njenih *slave* instanci. *Slave* instance započinju izbor kada je *master* instanca u stanju FAIL. Svaka od *slave* instanci promoviše sebe kao kandidata za izbor nove *master* instance. Za novu *master* instancu uzima se *slave* instanca koja je pobijedila prilikom izbora.

Slave instanca može započeti sa izborom za *master* instancu pod sljedećim uslovima:

- *master* instanca je u stanju FAIL,
- zadužena je za iste slotove kao i *master* instanca (broj slotova mora biti veći od 0) i
- nema komunikaciju sa *master* instancom određeni vremenski period (period je konfigurabilan).

Prvi korak *slave* instance je da uveća *currentEpoch* brojač i zahtijeva glasove od ostalih *master* instanci. Zahtjevi se šalju putem FAILOVER_AUTH_REQUEST paketa svakoj *master* instanci u klasteru. Nakon toga se čeka maksimalno 2 * NODE_TIMEOUT na odgovor od *master* instanci. Kada *master* glasa za neku *slave* instancu odgovara joj sa FAILOVER_AUTH_ACK i ne može glasati za druge *slave* instance od iste *master* instance u periodu od 2 * NODE_TIMEOUT. Na ovaj način se sprečava da u istom trenutku bude izabrano više *slave* instanci za novu *master* instancu. *Slave* instance odbacuju FAILOVER_AUTH_ACK odgovore ako je *currentEpoch* manji od njihovog, čime se obezbjeđuje da se ne broje glasovi od prethodnog izbora. Instanca koja je primila većinu ACK odgovora pobjeđuje prilikom izbora. Ukoliko većina nije postignuta u periodu od 2 * NODE_TIMEOUT, novi izbori se pokreću ponovo nakon 4 * NODE_TIMEOUT.

Svaka *master* instanca oglašava svoj *configEpoch* prilikom slanja *ping* i *pong* paketa zajedno sa opsegom heš slotova. Kod *master* instanci polje *configEpoch* se postavlja na 0 kada je kreirana nova instanca. Kada *slave* instanca pobijedi prilikom izbora, postaje nova *master* instanca i dobija novu jedinstvenu i uvećanu *configEpoch* vrijednost, koja je veća od *configEpoch* vrijednosti svih ostalih *master* instanci u klasteru. Nakon toga, počinje da se oglašava kao *master* instanca putem *ping* i *pong* paketa, šaljući svoje heš slotove sa *configEpoch* poljem, čija je vrijednost veća u odnosu na postojeće. *Pong* paketi se šalju putem *broadcast* poruka svim instancama u klasteru. Instance koje trenutno nisu dostupne mogu naknadno da izvrše rekonfiguraciju kada prihvate *ping* ili *pong* pakete od drugih instanci ili putem UPDATE paketa od drugih instanci, ako su informacije koje se oglašavaju putem *heartbeat* paketa zastarjele. Druge instance će detektovati da nova *master* instanca opslužuje iste heš slotove kao prethodna *master* instanca, ali sa većom *configEpoch* vrijednosti, nakon čega će ažurirati svoju konfiguraciju. Ostale *slave* instance stare *master* instance promijeniće svoju konfiguraciju i postati *slave* instance nove *master* instance. *Master* instanca koja ima najveću *configEpoch* vrijednost postaje vlasnik opsega heš slotova, za razliku od ostalih instanci koje ga oglašavaju. Ovaj mehanizam u Redis klasteru se naziva *last failover wins*.

Slična situacija je i kod migracije heš slotova između instanci. Kada odredišna instanca završi sa *import* operacijom, vrijednost njenog *configEpoch* polja se uvećava, kako bi se izmjena usvojila u cijelom klasteru.

Ukoliko se neka od instanci ponovo pridruži klasteru, njena *configEpoch* vrijednost je manja u odnosu na ostale instance. Nakon što se priključi klasteru, šalje *heartbeat* pakete sa svojim opsegom heš slotova i *configEpoch* vrijednošću. Instance koje prihvataju *heartbeat* pakete uočavaju da druga instanca u klasteru sa većom *configEpoch* vrijednošću ima isti opseg heš slotova kao instanca koja se ponovo pridružila klasteru. Na osnovu ovoga zaključuju da je konfiguracija instance pošiljaoca zastarila i šalju joj UPDATE poruku sa novom konfiguracijom slotova. Instanca koja se ponovo pridružila klasteru prihvata UPDATE poruku i ažurira svoju konfiguraciju.

Kada se instanca ponovo pridružuje klasteru koristi se sljedeće pravilo:
master instanca mijenja svoju konfiguraciju u *slave* instancu one instance koja je preuzela njen posljednji heš slot nakon otkazivanja.

Postoje dva slučaja:

- Kada posmatramo *master* instancu A i njenu jedinu *slave* instancu B, u slučaju da *master* instanca A prestane sa radom, njenu poziciju preuzima njena *slave* instanca B. Kada se instanca A vrati u klaster, ona postaje *slave* instanca instance B.
- U slučaju da *master* instanca A koja je posjedovala heš slotove 1 i 2 prestane sa radom, pri čemu je u međuvremenu *master* instanca B preuzela heš slot 1, a instanca C preuzela heš slot 2, nakon vraćanja u klaster instanca A postaje *slave* instanca instance C.

Ako posmatramo Redis klaster kod kojeg svaka *master* instanca ima po jednu *slave* instancu i pri tome dođe do otkaza bilo koje *master* instance, klaster će nastaviti sa radom. Njegovu ulogu će preuzeti njegova *slave* instanca. Međutim, ako otkaže i ta *slave* instanca, koja je proglašena za *master* instancu, klaster će prestati sa radom. Redis klaster nudi mehanizam koji se naziva *migracija replika*. Putem ovog mehanizma moguće je izbjeći prethodno navedenu situaciju, ako u klaster dodamo još *slave* instanci.

Ako posmatramo klaster sa *master* instancama A i B koje imaju po jednu *slave* instancu A1 i B1 i *master* instancu C sa dvije *slave* instance C1 i C2, u trenutku otkaza instance A njenu ulogu preuzima njena *slave* instanca A1. Putem mehanizma za migraciju replika *slave* instanca C2 postaje *slave* instanca instance A1, koja nije imala niti jednu *slave* instancu do tog trenutka. U slučaju da dođe do otkaza *master* instance A1 njenu ulogu će preuzeti instanca C2. Na ovaj način dobija se Redis klaster koji je otporniji na otkaze, ali uz nešto veće troškove, zbog toga što nova Redis instanca zahtijeva više memorije, više

procesorske snage, te možda čak i novi server, u zavisnosti od predviđene arhitekture. Ukoliko se u klasteru nalazi više *master* instanci koje imaju više od jedne *slave* instance, prilikom migracije replika dolazi do migracije one *slave* instance koja ima najmanji ID. Algoritam kod migracije replika se može kontrolisati od strane korisnika mijenjanjem *cluster-migration-barrier* parametra. Ovaj parametar označava minimalan broj *slave* instanci koje moraju ostati vezane za neku *master* instancu prilikom izvršavanja algoritma. Ako je *cluster-migration-barrier* postavljen na 2, *slave* instanca može migrirati pod uslovom da njena *master* instanca ostane sa najmanje dvije *slave* instance.

Ukoliko se *master* instanca ukloni iz klastera, njeni podaci se prebacuju na druge *master* instance. Kako bi se spriječilo da druge instance i dalje pristupaju uklonjenoj instanci, pošto pamte njen ID i adresu, koristi se komanda *CLUSTER FORGET node-id*.

Izvršavanjem prethodne komande postiže se sljedeće:

- uklanja se instanca sa specifikovanim ID-om iz tabele drugih instanci i
- postavlja se period zabrane (eng. *ban period*) od 60 sekundi kojim se sprečava da se doda nova instanca u klaster sa istim ID-om.

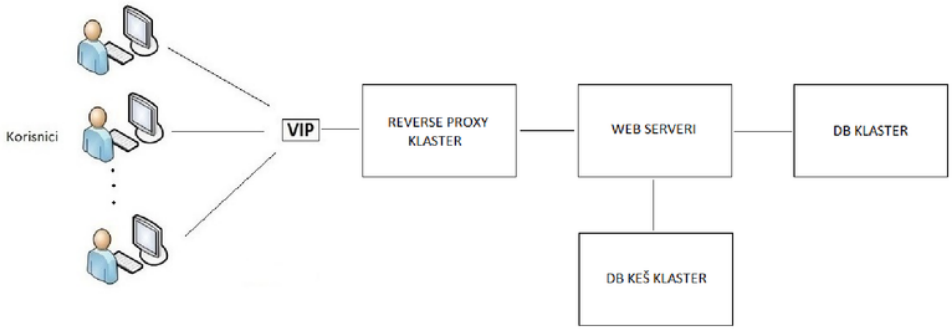
Postavljanje perioda zabrane je veoma bitno, zbog toga što Redis klaster koristi *gossip* protokol kako bi otkrio nove instance u klasteru. Ukoliko ne bi bilo ove operacije, kada se izbaci instanca X iz tabele instance A, može se desiti da instanca B putem *gossip* protokola ponovo proslijedi instanci A informacije o instanci X, pri čemu bi se instanca X ponovo nalazila u tabeli instance A. Postavljanjem perioda zabrane na 60 sekundi, alati za administraciju Redis klastera za to vrijeme moraju da uklone instancu iz tabele svih ostalih instanci, čime se sprečava ponovno dodavanje instance u tabelu prilikom automatskog pronalaženja novih instanci.

4. PRIJEDLOG SKALABILNE ARHITEKTURE WEB APLIKACIJA

U ovom poglavlju izložen je koncept višeslojne arhitekture web aplikacija, koja je skalabilna i otporna na otkaze. Namijenjena je za web aplikacije sa velikim brojem korisnika, kod kojih je naglasak na čitanju, a ne na upisu podataka.

4.1 Koncept skalabilne arhitekture web aplikacija

Na sljedećoj slici (Slika 4.1) prikazan je koncept arhitekture web aplikacija koja je skalabilna, otporna na otkaze i koja znatno smanjuje vrijeme odziva web aplikacije pri radu sa velikim brojem korisnika.



Slika 4.1 – Koncept skalabilne arhitekture web aplikacija

VIP adresa predstavlja jedinstvenu tačku pristupa za Reverse proxy klaster. Osnovna namjena Reverse proxy klastera je keširanje korisnički nezavisnih podataka i raspodjela opterećenja ka web serverima. Klasterizacijom Reverse proxy servera povećava se i dostupnost web aplikacije, odnosno u slučaju da dođe do otkaza nekog Reverse proxy servera web aplikacija će nastaviti sa radom. Keširanjem korisnički nezavisnih podataka znatno se ubrzava rad aplikacije. Prvi zahtjev za nekim resursom se prosljeđuje web serveru. Web server prikuplja podatke iz baze ili iz nekog drugog izvora (npr. web servisa), po potrebi izvršava odgovarajuću logiku i vraća odgovor Reverse proxy serveru. Ukoliko je dozvoljeno keširanje podataka, Reverse proxy server kešira odgovor od web servera. Za svaki sljedeći zahtjev za istim resursom, Reverse proxy server vraća traženi resurs iz svoje keš memorije, sve dok period čuvanja objekta u kešu ne istekne ili dok se objekat ne ukloni iz keš memorije. Raspodjelom opterećenja ka više web servera postižu se bolje performanse i obezbjeđuje veća dostupnost. Ukoliko dođe do otkaza nekog web servera, web aplikacija će nastaviti sa radom. Web serveri rade u paraleli i pristupaju DB

klasteru i DB keš klasteru. DB klaster predstavlja klaster relacionih baza podataka, koji je namijenjen za skladištenje svih podataka. Klasterizacijom DB servera povećava se dostupnost u slučaju da dođe do otkaza nekog DB servera. DB keš klaster se koristi za keširanje korisnički zavisnih podataka, čime se postižu znatno bolje performanse pri radu sa korisnički zavisnim podacima. Keširanje podataka dolazi do izražaja prilikom upotrebe relacionih baza podataka, kada se zahtijevaju podaci iz više različitih tabela, koje posjeduju veliku količinu podataka. Keširanjem takvih podataka znatno se ubrzava rad web aplikacije i rasterećuje upotreba web servera i servera baza podataka. Upotreba DB keš klastera i Reverse proxy klastera sve više dobija na značaju kako se povećava količina podataka u relacionoj bazi podataka.

Sigurnost rada web aplikacije se ne uzima u razmatranje, kao ni propusni opseg mreže. Akcenat je stavljen na skalabilnost i dostupnost cijelog sistema. Sigurnost je veoma bitna, ali kod izloženog koncepta arhitekture web aplikacija očekuje se da se uređaj koji garantuje sigurnu komunikaciju (HTTPS – *Hypertext Transfer Protocol Secure*) i sprečava razne pokušaje napada na web aplikaciju nalazi ispred Reverse proxy klastera. Komunikacija sa Reverse proxy klasterom se odvija putem HTTP protokola.

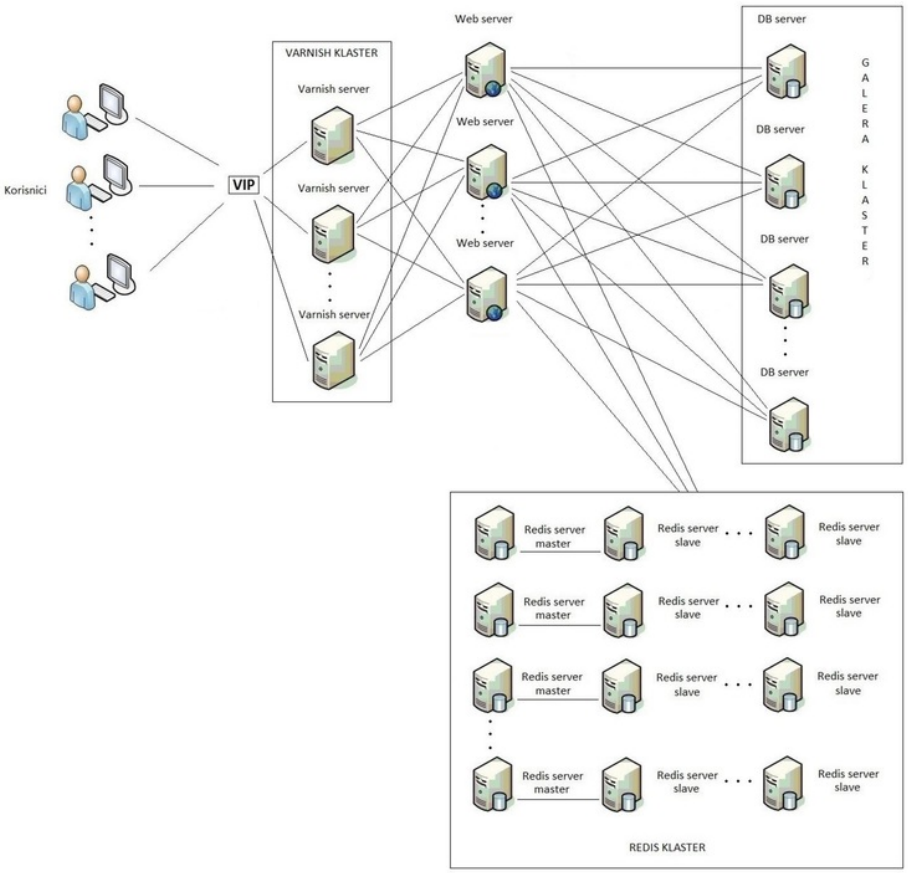
Sa izloženim konceptom arhitekture web aplikacija riješen je problem skalabilnosti sistema prilikom porasta broja korisnika web aplikacije, kao i dostupnosti cijelog sistema, ukoliko dođe do otkaza neke od njegovih komponenti.

4.2 Implementacija koncepta

Za implementaciju prethodno izloženog koncepta skalabilne arhitekture web aplikacija korišten je stek tehnologija opisan u 3. poglavlju. Istu arhitekturu je moguće realizovati i sa drugim stekom tehnologija. Sve korištene tehnologije za realizaciju predložene arhitekture web aplikacija (Slika 4.2) su *open source* i besplatne za korištenje. Opisani stek tehnologija, koji se preporučuje za realizaciju predložene arhitekture web aplikacija je dostupan na Linux operativnom sistemu.

Prilikom implementacije izloženog koncepta skalabilne arhitekture web aplikacija koristi se:

- Varnish klaster kao Reverse proxy klaster,
- Galera klaster kao DB klaster i
- Redis klaster kao DB keš klaster.



Slika 4.2 – Predložena arhitektura web aplikacija

Varnish klaster se koristi za keširanje korisnički nezavisnih podataka, odnosno podataka koji su isti za sve korisnike. Dodatno, vrši raspodjelu opterećenja ka web serverima. Varnish klaster radi u aktiv-pasiv modu. Jedan server je *master*, dok ostali serveri predstavljaju *slave* servere. U slučaju otkaza *master* servera, *slave* server preuzima njegovu ulogu. Varnish klaster je moguće naknadno proširivati sa *slave* serverima. Varnish server posjeduje mehanizam za ispitivanje „zdravlja" web servera, tako da se saobraćaj usmjerava samo na „zdrave" web servere. Teoretski, Varnish server bi mogao komunicirati sa N web servera i vršiti raspodjelu saobraćaja. Moguće je naknadno, na jednostavan način, dodavati nove web servere. *Pacemaker corosync* klaster omogućava klasterizaciju Varnish servera. VIP adresa se dodaje kao resurs u *Pacemaker corosync* klasteru i koristi se kao jedinstvena tačka pristupa za Varnish klaster.

Redis klaster se koristi za keširanje korisnički zavisnih podataka, odnosno podataka koji su specifični za svakog korisnika. Redis baza podataka je distribuirana na više *master* servera,

pri čemu se svakom *master* serveru može dodijeliti više *slave* servera. U slučaju da dođe do otkaza nekog *master* servera, *slave* server će preuzeti njegovu ulogu. Redis klaster se na prilično jednostavan način može naknadno proširivati.

Galera klaster je sinhroni multi-master klaster baza podataka, koji se u predloženoj arhitekturi web aplikacija koristi za klasterizaciju MariaDB baza podataka. S obzirom na to da se radi o sinhronom multi-master klasteru, web serveri mogu da šalju zahtjeve na bilo koji server baza podataka. Na ovaj način se postižu bolje performanse prilikom čitanja podataka. Omogućena je raspodjela opterećenja ka više servera baza podataka. Nešto lošije performanse se dobijaju prilikom dodavanja, ažuriranja i brisanja podataka. Potrebno je da se svaka izmjena nad bazom podataka izvrši na svim serverima baza podataka. Pošto je izložena arhitektura web aplikacija namijenjena isključivo za web aplikacije kod kojih je naglasak na čitanju, a ne na upisu podataka, upotrebom Galera klastera postižu se bolje performanse web aplikacije. Moguće je naknadno proširiti klaster, dodavanjem novih servera baza podataka.

4.2.1 Ograničenja predložene arhitekture web aplikacija

Ograničenja vezana za minimalan broj servera predložene arhitekture:

- kod Varnish klastera minimalan broj servera je 2, od kojih je jedan *master*, a drugi *slave* server – u slučaju otkaza *master* servera, *slave* server preuzima njegovu ulogu,
- minimalan broj web servera koji rade u paraleli je 2 – u slučaju da dođe do otkaza jednog web servera, web aplikacija će nastaviti sa radom,
- minimalan broj DB servera koji je potreban za ispravan rad Galera klastera je 3 – u slučaju da dođe do otkaza bilo kojeg servera baza podataka ili ukoliko dođe do dijeljenja mreže, web aplikacija će nastaviti sa radom,
- minimalan broj Redis servera koji je neophodan za ispravno funkcionisanje Redis klastera je 6, od kojih su 3 *master* i 3 *slave* servera – klaster će nastaviti da radi sve dok ne otkaže *master* i njegov *slave* server.

Neophodno je provjeriti i maksimalan dozvoljeni broj servera za svaku od korištenih tehnologija. Preporuka je da se provjera izvrši prilikom implementacije arhitekture, jer je moguće da kod nekih tehnologija maksimalan broj servera varira u zavisnosti od verzije korištene tehnologije.

5. PRAKTIČAN RAD

U okviru praktičnog dijela rada razvijena je web aplikacija *Online news*, koja se koristi za svrhe testiranja arhitekture web aplikacija. Početna arhitektura web aplikacija je bila troslojna sa jednim web serverom i jednim serverom baza podataka. Postepeno su se dodavale nove komponente, mjerile performanse čitanja podataka razvijene web aplikacije i analizirali dobijeni rezultati. Konačna arhitektura web aplikacija zasniva se na konceptu izloženom u 4. poglavlju ovog rada. Dat je uporedni prikaz dobijenih rezultata za početnu i konačnu arhitekturu u vidu tabele i grafikona.

5.1 Pregled projektnih zahtjeva

Potrebno je implementirati arhitekturu web aplikacija koja može opslužiti veliki broj korisnika, a istovremeno je otporna i na otkaze. Za skladištenje podataka potrebno je koristiti MariaDB bazu podataka i Galera klaster, koji predstavlja sinhroni multi-master klaster baza podataka. Za keširanje korisnički nezavisnih podataka i raspoređivanje opterećenja ka web serverima potrebno je koristiti Varnish servere koji rade u aktiv-pasiv klasteru. Varnish klaster bi trebalo da posjeduje dva Varnish servera, pri čemu bi jedan bio *master* a drugi *slave* server. U slučaju otkaza *master* servera *slave* server bi preuzeo njegovu ulogu. Predviđena je realizacija sa 3 web servera. Web serveri bi trebalo da komuniciraju sa Redis klasterom, koji bi bio zadužen za keširanje korisnički zavisnih podataka. Redis klaster bi trebalo da radi sa 3 *master* i 3 *slave* instance. Potrebno je razviti servisno orijentisanu jednostraničnu web aplikaciju za čitanje vijesti putem Interneta, koja bi se izvršavala na prethodno pomenutoj arhitekturi web aplikacija. Kod ove i njoj sličnih aplikacija više je zastupljeno čitanje od upisa podataka, tako da predstavlja idealan primjer za testiranje predložene arhitekture. Od klijentski zavisnih podataka, na Redis klasteru bi se keširale odabrane vijesti za svakog korisnika. Početna arhitektura trebalo bi da bude troslojna arhitektura web aplikacija sa jednim web serverom i jednim serverom baza podataka. Potrebno je postepeno dodavati nove komponente, mjeriti performanse čitanja podataka i analizirati dobijene rezultate.

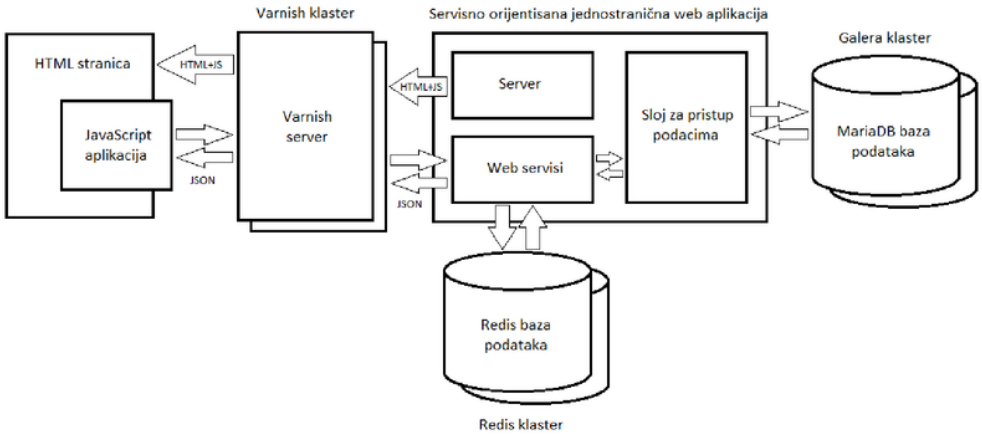
Pošto je naglasak na skalabilnosti, dostupnosti i brzini odziva web aplikacije, testiranje bi trebalo da se vrši iz lokalne mreže. Podrazumijeva se da su performanse virtuelnih mašina zadovoljavajuće.

5.2 Detalji implementacije i analiza rezultata

Za razvoj klijentskog dijela web aplikacije korišten je programski jezik JavaScript, uključujući HTML i CSS, kao i Webix (JavaScript i HTML 5) framework. Za serverski dio web aplikacije korišten je Java programski jezik, uključujući Spring, Hibernate i JPA (Java Persistence API).

Putem razvijene web aplikacije moguće je vršiti pregled vijesti. Vijesti je moguće filtrirati po kategoriji, potkategoriji i nazivu. Pored vijesti prikazuju se i oglasi/reklame. Postoje četiri tipa korisnika: administrator, urednik, regularni korisnik i gost. Administrator može izvršavati osnovne operacije (CRUD) nad vijestima, oglasima, stavkama menija, kategorijama vijesti, korisnicima, nivoima pristupa, spiskom gradova, regija i država. Dodatno, može izvršiti određena podešavanja koja utiču na rad aplikacije. Urednik može izvršavati osnovne operacije nad vijestima i oglasima, vršiti pregled postojećih vijesti, kao i filtrirati vijesti po kategorijama, potkategorijama i nazivu. Regularnom korisniku je ponuđen prikaz i filtriranje vijesti, prikaz oglasa, kao i dodavanje vijesti u odabrane, koje može naknadno čitati i filtrirati. Regularni korisnici se moraju registrovati da bi koristili web aplikaciju. Gostu se nudi prikaz i filtriranje vijesti, kao i prikaz oglasa, bez izvršene registracije na web aplikaciju. Za razliku od gosta, ostala tri tipa korisnika se moraju prijaviti za rad na web aplikaciji. Nakon uspješne autentikacije biće im obezbijeđene prethodno pomenute funkcionalnosti.

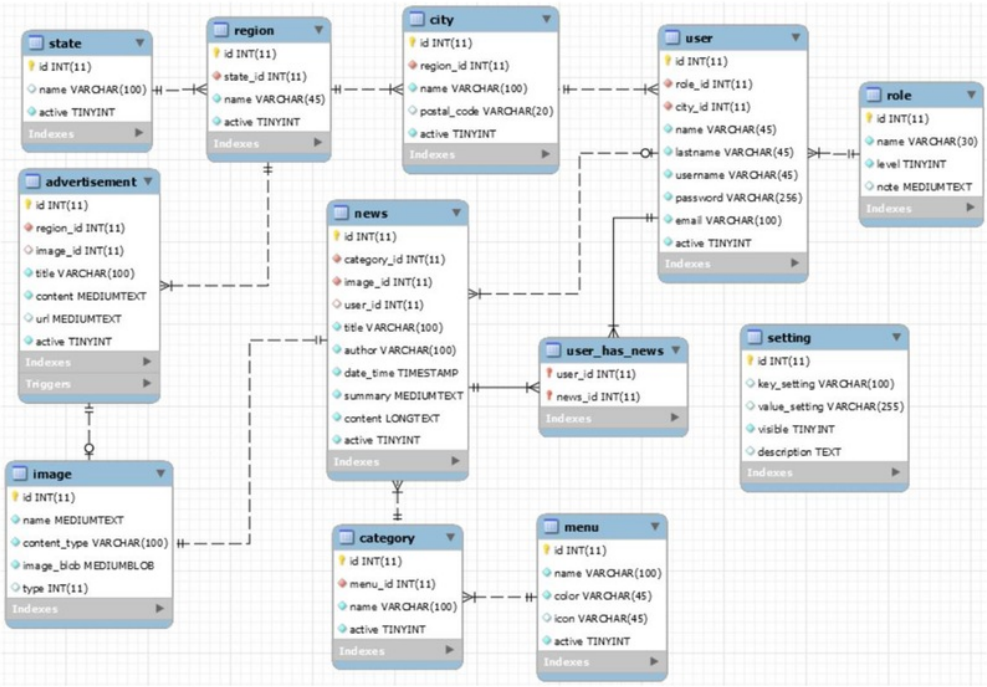
Web aplikacija je razvijena kao stand-alone Spring Boot aplikacija sa ugrađenim Tomcat serverom. Pokreće se kao servis na instaliranom operativnom sistemu.



Slika 5.1 – Grafički prikaz rada web aplikacije

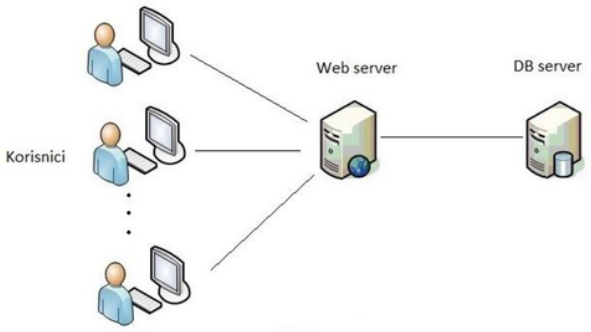
Za svrhe implementacije i testiranja rada arhitekture web aplikacija, koriste se virtuelne mašine koje se nalaze na HP serveru *ProLiant DL160 Gen9*. Korišteno je 14 virtuelnih mašina na kojima je instaliran CentOS 7 i jedna virtuelna mašina na kojoj se instaliran Windows 7. Virtuelne mašine sa instaliranim CentOS 7 operativnim sistemom se koriste za realizaciju konačne arhitekture web aplikacija, dok se virtuelna mašina sa Windows 7 operativnim sistemom koristi za svrhe testiranja performansi.

Za testiranje performansi web aplikacije koristi se *open source* alat *Gatling*. Performanse će se mjeriti za prikupljanje vijesti sa naslovne stranice, kao i za odabrane vijesti.



Slika 5.2 – Dijagram baze podataka za aplikaciju *Online news*

Početna arhitektura web aplikacija je troslojna i sastoji se od jednog web servera i jednog servera baza podataka. Na web serveru se nalazi prethodno razvijena web aplikacija *Online news*. U slučaju otkaza web servera ili servera baza podataka web aplikacija će biti nedostupna.



Slika 5.3 – Početna arhitektura web aplikacija

U bazi podataka se nalazi ~ 350 000 vijesti. Za prikupljanje vijesti prikazanih na naslovnoj stranici izvršava se sljedeći upit:

```
SELECT n.*,c.name AS category_name, m.color AS menu_color
FROM (SELECT * FROM news WHERE active=1 ORDER BY id DESC LIMIT 1000) n
  JOIN category c ON n.category_id=c.id
  JOIN menu m ON c.menu_id=m.id
ORDER BY id DESC;
```

Prikuplja se posljednjih 1000 vijesti iz baze podataka. Za izvršavanje upita potrebno je oko 6 sekundi. Moguće je na različite načine dobiti isti skup podataka. Sljedećim upitom se dobija isti skup podataka:

```
SELECT n.*,c.name AS category_name, m.color AS menu_color
FROM news n
  JOIN category c ON n.category_id=c.id
  JOIN menu m ON c.menu_id=m.id
WHERE n.active=1
ORDER BY id DESC LIMIT 1000;
```

Iako se dobija isti skup podataka, za izvršavanje prethodno navedenog upita potrebno je oko 60 sekundi. Razlog je izvršavanje JOIN operacije (Dekartov proizvod) svih vijesti sa zapisima *category* i *menu* tabela, umjesto da se JOIN operacija izvršava samo za 1000 najnovijih vijesti.

Vijesti koje se dobiju putem prvog upita sortiraju se po id-u vijesti u opadajućem redoslijedu, pri čemu se uzima prvih 1000 vijesti (najnovijih 1000 vijesti). Id vijesti je cijeli broj, primarni ključ tabele, vrijednost koja se automatski povećava prilikom

dodavanja nove vijesti i to je vrijednost koja je indeksirana. Prikupljanje vijesti za naslovnu stranicu je mnogo brže ukoliko se vijesti vraćaju sortirane na osnovu id-a vijesti, umjesto da se vrši sortiranje po datumu objavljivanja vijesti, odnosno trenutku kada je vijest upisana u bazu podataka (polje *date_time* u tabeli *news*). Ukoliko bi se vraćale vijesti na osnovu prvog upita, pri čemu bi sortiranje bilo na osnovu datuma i vremena umjesto id-a vijesti, bilo bi potrebno oko 10 sekundi za izvršavanje upita. Neophodno je voditi računa o optimizaciji upita.

Korištenjem programskog jezika *Scala* napisana je simulaciona skripta (eng. *simulation script*) koju je koristio program *Gatling* prilikom testiranja performansi. Testirale su se performanse prilikom prikupljanja vijesti za naslovnu stranicu od strane 1000 korisnika u okviru jedne sekunde. U programu je navedeno da se test ponavlja 5 puta, što znači da je pri svakom testu bilo upućeno 5000 zahtjeva.

Vijesti sa naslovne stranice koje su se prikupljale tokom testiranja zauzimale su oko 2.2MB memorije na heap-u. Pošto se testiranje vršilo za 1000 korisnika u okviru jedne sekunde, prilikom pokretanja web aplikacije dodijeljeno joj je 3GB memorije na heap-u, da ne bi došlo do situacije da GC (Garbage Collector) ne stiže počistiti memoriju usljed velikog broja zahtjeva. Aplikacija je pokrenuta sljedećom komandom:

```
java –Xmx3g –jar online-news.jar
```

Parametri za pristup bazi podataka iz aplikacije su sljedeći:

```
spring.datasource.url=jdbc:mysql://192.168.15.237/online_news?useUnicode=yes&characterEncoding=UTF-8
spring.datasource.username=root
spring.datasource.password=admin
spring.datasource.tomcat.max-wait=50000
spring.datasource.tomcat.test-on-borrow=true
spring.datasource.tomcat.max-active=1200
spring.datasource.tomcat.max-idle=1100
spring.datasource.tomcat.min-idle=800
```

Rezultati testa:

```
Simulation OnlineNewsTroslojnaArhitektura completed in 54 seconds
Parsing log file(s)...
Parsing log file(s) done
Generating reports...

================================================================================
---- Global Information --------------------------------------------------------
> request count                                       5000 (OK=797    KO=4203  )
> min response time                                     93 (OK=93     KO=175   )
> max response time                                  32240 (OK=32240  KO=16356 )
> mean response time                                 9060 (OK=17840  KO=7395  )
> std deviation                                      5756 (OK=7238   KO=3477  )
> response time 50th percentile                      7104 (OK=17236  KO=6438  )
> response time 75th percentile                     11008 (OK=22352  KO=9549  )
> response time 95th percentile                     21692 (OK=30704  KO=14733 )
> response time 99th percentile                     30556 (OK=31561  KO=15060 )
> mean requests/sec                                90.909 (OK=14.491 KO=76.418)
---- Response Time Distribution ------------------------------------------------
> t < 800 ms                                           28 (  1%)
> 800 ms < t < 1200 ms                                  4 (  0%)
> t > 1200 ms                                         765 ( 15%)
> failed                                             4203 ( 84%)
```

Od ukupno 5000 zahtjeva upućenih ka web serveru, 4203 su bila neuspješna. Greška koja se javila na web serveru je „Too many connections". Uzrok ove greške je maksimalan broj konekcija koje MariaDB baza podataka može da prihvati. Maksimalan broj konekcija se može provjeriti sa komandom:

```
SHOW variables LIKE 'max_connections';
```

Podrazumijevani broj konekcija je 151.

Maksimalan broj konekcija se može promijeniti sljedećom komandom:

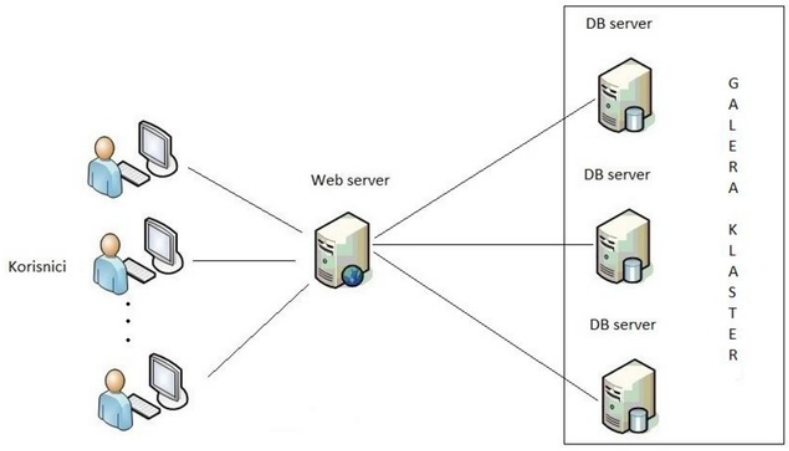
```
SET GLOBAL max_connections=500;
```

Nakon izvršavanja prethodne komande nad bazom podataka dobijamo sljedeće rezultate:

```
Simulation OnlineNewsTroslojnaArhitektura completed in 575 seconds
Parsing log file(s)...
Parsing log file(s) done
Generating reports...

================================================================================
---- Global Information --------------------------------------------------------
> request count                                       5000 (OK=5000   KO=0     )
> min response time                                   1843 (OK=1843   KO=-     )
> max response time                                 136795 (OK=136795 KO=-     )
> mean response time                                105588 (OK=105588 KO=-     )
> std deviation                                      27478 (OK=27478  KO=-     )
> response time 50th percentile                     115842 (OK=115842 KO=-     )
> response time 75th percentile                     120771 (OK=120771 KO=-     )
> response time 95th percentile                     126494 (OK=126494 KO=-     )
> response time 99th percentile                     129781 (OK=129781 KO=-     )
> mean requests/sec                                  8.681 (OK=8.681  KO=-     )
---- Response Time Distribution ------------------------------------------------
> t < 800 ms                                           0 (  0%)
> 800 ms < t < 1200 ms                                 0 (  0%)
> t > 1200 ms                                       5000 (100%)
> failed                                               0 (  0%)
================================================================================
```

Može se uočiti da su svi zahtjevi ka web serveru prošli uspješno. Međutim, test se izvršavao ukupno 575 sekundi, pri čemu je minimalno vrijeme odgovora bilo 1.843 sekunde, maksimalno 136.795 sekundi, dok je srednje vrijeme odgovora bilo 105.588 sekundi. Ovo su veoma loše performanse i ukoliko bi došlo do situacije da se objavi neka udarna vijest, web aplikaciji bi pristupio veliki broj korisnika istovremeno. Većina njih bi napustila sajt, izgubila interesovanje i koristila neke konkurentske web sajtove.

U ovakvoj arhitekturi web aplikacija server baza podataka predstavlja usko grlo i ideja je da se izvrši raspodjela opterećenja na nivou baze podataka, tako što bi se umjesto jednog servera baza podataka koristio Galera klaster, koji bi radio sa 3 MariaDB servera. Moguće je naknadno proširiti klaster dodavanjem novih MariaDB servera baza podataka.



Slika 5.4 –Arhitektura web aplikacija sa Galera klasterom

Nakon uspješne konfiguracije Galera klastera, sa sljedećom komandom se može provjeriti broj instanci u klasteru na bilo kojem od servera baza podataka:

```
SHOW GLOBAL STATUS LIKE 'wsrep_cluster_size';
```

Rezultat izvršavanja komande je 3, što znači da se u klasteru nalaze 3 instance. Pošto je Galera sinhroni multi-master klaster, upis i čitanje podataka se može vršiti na bilo kojem od servera. U slučaju otkaza neke instance klaster nastavlja normalno da funkcioniše.

Da bi web aplikacija vršila raspodjelu opterećenja ka sva tri konfigurisana servera, neophodno je da se promijene parametri za pristup bazi podataka iz aplikacije (konekcioni string):

```
spring.datasource.url=jdbc:mysql:loadbalance://192.168.15.237,192.168.15.238,192.168.1
5.239/online_news?useUnicode=yes&characterEncoding=UTF-8
spring.datasource.username=root
spring.datasource.password=admin
spring.datasource.tomcat.max-wait=50000
spring.datasource.tomcat.test-on-borrow=true
spring.datasource.tomcat.max-active=1200
spring.datasource.tomcat.max-idle=1100
spring.datasource.tomcat.min-idle=800
```

Rezultati testiranja nakon implementacije Galera klastera su sljedeći:

```
Simulation OnlineNewsTroslojnaArhitektura completed in 121 seconds
Parsing log file(s)...
Parsing log file(s) done
Generating reports...

================================================================================
---- Global Information --------------------------------------------------------
> request count                                       5000 (OK=5000   KO=0     )
> min response time                                    123 (OK=123    KO=-     )
> max response time                                  30287 (OK=30287  KO=-     )
> mean response time                                 22025 (OK=22025  KO=-     )
> std deviation                                       4974 (OK=4974   KO=-     )
> response time 50th percentile                      23371 (OK=23371  KO=-     )
> response time 75th percentile                      24675 (OK=24675  KO=-     )
> response time 95th percentile                      26763 (OK=26763  KO=-     )
> response time 99th percentile                      28742 (OK=28742  KO=-     )
> mean requests/sec                                 40.984 (OK=40.984 KO=-     )
---- Response Time Distribution ------------------------------------------------
> t < 800 ms                                             9 (  0%)
> 800 ms < t < 1200 ms                                   0 (  0%)
> t > 1200 ms                                         4991 (100%)
> failed                                                 0 (  0%)
```

Svi zahtjevi ka web serveru su prošli uspješno. Test je trajao 121 sekundu, pri čemu je minimalno vrijeme odgovora bilo 0.123 sekunde, maksimalno 30.287 sekundi, dok je srednje vrijeme odgovora bilo 22.025 sekundi. Može se uočiti da je srednje vrijeme odgovora palo sa 136.795 sekundi na 22.025 sekundi.

Moguće je naknadno dodavati nove servere baza podataka, ali porastom broja korisnika u nekom trenutku će doći do situacije kada će i web server postati usko grlo i neće moći isprocesirati sve korisničke zahtjeve. U slučaju otkaza web servera, web sajt postaje nedostupan. Ovo je jedina gora situacija od „sporog" odgovora.

Ukoliko se prostor na heap-u smanji, padaju i performanse web aplikacije. Dodatno vrijeme se troši na oslobađanje memorije od strane *GC-a*.

Ako se web aplikacija pokrene sa 1.5GB prostora na heap-u, dobijaju se sljedeći rezultati:

```
Simulation OnlineNewsTroslojnaArhitektura completed in 244 seconds
Parsing log file(s)...
Parsing log file(s) done
Generating reports...

================================================================================

---- Global Information --------------------------------------------------------

> request count                                       5000 (OK=5000   KO=0     )

> min response time                                   2240 (OK=2240   KO=-     )

> max response time                                  83098 (OK=83098  KO=-     )

> mean response time                                 45120 (OK=45120  KO=-     )

> std deviation                                      11956 (OK=11956  KO=-     )

> response time 50th percentile                      46255 (OK=46255  KO=-     )

> response time 75th percentile                      51304 (OK=51304  KO=-     )

> response time 95th percentile                      62552 (OK=62552  KO=-     )

> response time 99th percentile                      70275 (OK=70275  KO=-     )

> mean requests/sec                                 20.408 (OK=20.408 KO=-     )

---- Response Time Distribution ------------------------------------------------

> t < 800 ms                                            0 (  0%)
> 800 ms < t < 1200 ms                                  0 (  0%)
> t > 1200 ms                                        5000 (100%)
> failed                                                0 (  0%)
```

Kako bi se simulirala situacija kada web aplikacija ne može obraditi sve korisničke zahtjeve, pokrenuta je web aplikacija sa 1GB prostora na heap-u, pri čemu se testiranje vršilo na isti način.

Rezultati izvršenog testa su sljedeći:

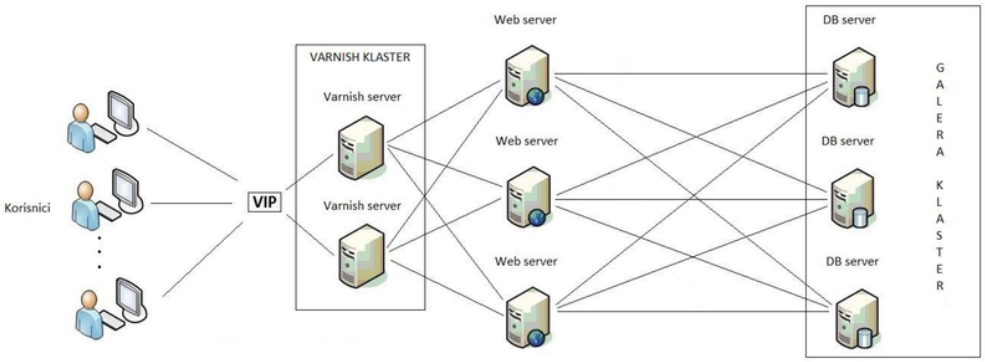
```
Simulation OnlineNewsTroslojnaArhitektura completed in 1506 seconds
Parsing log file(s)...
Parsing log file(s) done
Generating reports...

================================================================================

---- Global Information --------------------------------------------------------

> request count                                       5000 (OK=17     KO=4983  )

> min response time                                      0 (OK=97     KO=0     )

> max response time                                 594382 (OK=562708 KO=594382)

> mean response time                                 28533 (OK=366734 KO=27379 )

> std deviation                                     105504 (OK=169311 KO=103343)

> response time 50th percentile                          0 (OK=363962 KO=0     )

> response time 75th percentile                          0 (OK=497811 KO=0     )

> response time 95th percentile                     281421 (OK=557355 KO=273832)

> response time 99th percentile                     540278 (OK=561637 KO=540244)

> mean requests/sec                                  3.318 (OK=0.011  KO=3.307 )

---- Response Time Distribution ------------------------------------------------

> t < 800 ms                                            2 (  0%)
> 800 ms < t < 1200 ms                                  0 (  0%)
> t > 1200 ms                                          15 (  0%)
> failed                                             4983 (100%)
```

Usljed upućenog velikog broja zahtjeva, tokom testiranja web aplikacije javila se greška „java.lang.OutOfMemoryError: GC overhead limit exceeded”. GC ne stiže da oslobodi

memoriju na heap-u, novi korisnički zahtjevi pristižu, a web aplikacija nije više u stanju da obradi nove zahtjeve. Za operacije koje se inače izvršavaju u relativno kratkom vremenskom periodu sada je potrebno nekoliko minuta. Nakon nekog vremena dolazi do prekida konekcije između klijenta i servera, što je u prethodnom testu rezultovalo sa velikim brojem neuspješnih zahtjeva.

Performanse web aplikacije se mogu poboljšati vertikalnim skaliranjem, odnosno povećanjem performansi samog web servera. Povećanje performansi servera je ograničeno i u nekom trenutku server će doći do zasićenja. Rješenje je horizontalno skaliranje, odnosno dodavanje još web servera koji će raditi u paraleli i na kojima će se raspoređivati opterećenje. Na ovaj način, pored boljih performansi postiže se i veća dostupnost. U slučaju da otkaže bilo koji od web servera, web aplikacija će nastaviti sa radom i web sajt će biti dostupan krajnjim korisnicima.

Ideja je da se doda aktiv-pasiv Varnish klaster koji bi vršio raspodjelu opterećenja ka web serverima i keširanje korisnički nezavisnih podataka.



Slika 5.5 – Arhitektura web aplikacija sa Varnish klasterom, tri web servera koja rade u paraleli i Galera klasterom

Varnish klaster bi radio sa dva Varnish servera. Jedan od njih bio bi *master*, a drugi bi bio *slave*. U trenutku otkaza *master* servera, *slave* server bi preuzeo njegovu ulogu. *Master* server bi vršio raspoređivanje saobraćaja ka tri web servera. Web serveri ne komuniciraju međusobno i nemaju informaciju o postojanju drugih web servera. Novi web server se dodaje prilično jednostavno, s obzirom na to da se radi o Spring Boot aplikaciji. Nakon podešavanja web servera, potrebno je rekonfigurisati Varnish servere kako bi se novi web server pustio u rad. Za klasterizaciju Varnish servera koristi se *Pacemaker corosync* klaster. VIP predstavlja jedinstvenu tačku pristupa za Varnish klaster.

Kako bi se ilustrovalo unepređenje performansi arhitekture web aplikacija primjenom samo horizontalnog skaliranja web servera, koristili su se web serveri na kojima je pokrenuta web aplikacija koja koristi 1280MB prostora na heap-u. Testirana je arhitektura web aplikacija sa jednim web serverom i arhitektura web aplikacija sa Varnish klasterom i tri web servera. Varnish klaster prilikom ovog testa nije keširao odgovore od web servera. Da bi se ovo postiglo, u odgovoru web servera je dodano zaglavlje *Cache-control,* koje posjeduje vrijednost *no-cache*.

Rezultati testiranja pri korištenju arhitekture web aplikacija sa jednim web serverom:

```
Simulation OnlineNewsWebServer completed in 467 seconds
Parsing log file(s)...
Parsing log file(s) done
Generating reports...

================================================================================

---- Global Information --------------------------------------------------------

> request count                                       5000 (OK=5000   KO=0     )

> min response time                                   6111 (OK=6111   KO=-     )

> max response time                                 137285 (OK=137285 KO=-     )

> mean response time                                 85926 (OK=85926  KO=-     )

> std deviation                                      20440 (OK=20440  KO=-     )

> response time 50th percentile                      89631 (OK=89631  KO=-     )

> response time 75th percentile                      97670 (OK=97670  KO=-     )

> response time 95th percentile                     109532 (OK=109532 KO=-     )

> response time 99th percentile                     121427 (OK=121427 KO=-     )

> mean requests/sec                                 10.684 (OK=10.684 KO=-     )

---- Response Time Distribution ------------------------------------------------

> t < 800 ms                                            0 (  0%)
> 800 ms < t < 1200 ms                                  0 (  0%)
> t > 1200 ms                                        5000 (100%)
> failed                                                0 (  0%)
```

Rezultati testiranja pri korištenju arhitekture web aplikacija sa Varnish klasterom i tri web servera bez keširanja podataka:

```
Simulation OnlineNewsVarnish completed in 343 seconds
Parsing log file(s)...
Parsing log file(s) done
Generating reports...

================================================================================

---- Global Information --------------------------------------------------------

> request count                                       5000 (OK=5000   KO=0     )
> min response time                                     86 (OK=86     KO=-     )
> max response time                                 219967 (OK=219967 KO=-     )
> mean response time                                 62718 (OK=62718  KO=-     )
> std deviation                                      25950 (OK=25950  KO=-     )
> response time 50th percentile                      63917 (OK=63917  KO=-     )
> response time 75th percentile                      76401 (OK=76401  KO=-     )
> response time 95th percentile                     102738 (OK=102738 KO=-     )
> response time 99th percentile                     141859 (OK=141859 KO=-     )
> mean requests/sec                                 14.535 (OK=14.535 KO=-     )

---- Response Time Distribution ------------------------------------------------

> t < 800 ms                                            26 (  1%)
> 800 ms < t < 1200 ms                                   5 (  0%)
> t > 1200 ms                                         4969 ( 99%)
> failed                                                 0 (  0%)
```

Na osnovu dobijenih rezultata može se uočiti da arhitektura web aplikacija sa Varnish klasterom, koji vrši raspodjelu opterećenja ka tri web servera, daje bolje rezultate. Test je trajao 343 sekunde, pri čemu je minimalno vrijeme odgovora bilo 0.086 sekundi, maksimalno 219.967 sekundi, dok je srednje vrijeme odgovora bilo 62.72 sekunde. Može se uočiti da je srednje vrijeme odgovora palo sa 85.93 sekunde na 62.718 sekundi. Iako postoji napredak, i dalje se uočava velika razlika između maksimalnog i minimalnog vremena odziva.

Vijesti sa naslovne stranice web sajta su iste za sve korisnike. U pitanju su korisnički nezavisni podaci. Keširanje ovakvih podataka se vrši od strane Varnish servera. Da bi se omogućilo keširanje podataka potrebno je ukloniti *Cache-control* zaglavlje iz odgovora web servera.

Rezultati testiranja pri korištenju arhitekture web aplikacija sa Varnish klasterom i tri web servera uključujući i keširanje podataka su sljedeći:

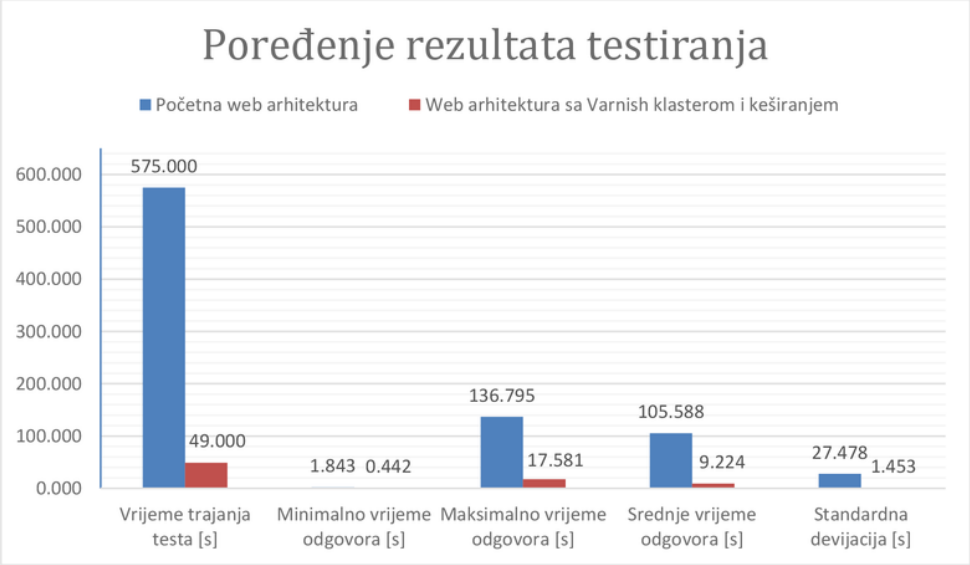
```
Simulation OnlineNewsVarnish completed in 49 seconds
Parsing log file(s)...
Parsing log file(s) done
Generating reports...

================================================================================

---- Global Information --------------------------------------------------------

> request count                                       5000 (OK=5000   KO=0     )
> min response time                                    442 (OK=442    KO=-     )
> max response time                                  17581 (OK=17581  KO=-     )
> mean response time                                  9224 (OK=9224   KO=-     )
> std deviation                                       1453 (OK=1453   KO=-     )
> response time 50th percentile                       9465 (OK=9465   KO=-     )
> response time 75th percentile                       9871 (OK=9871   KO=-     )
> response time 95th percentile                      10696 (OK=10696  KO=-     )
> response time 99th percentile                      14805 (OK=14805  KO=-     )
> mean requests/sec                                    100 (OK=100    KO=-     )

---- Response Time Distribution ------------------------------------------------

> t < 800 ms                                             3 (  0%)
> 800 ms < t < 1200 ms                                   4 (  0%)
> t > 1200 ms                                         4993 (100%)
> failed                                                 0 (  0%)
```

Keširanjem podataka postižu se znatno bolji rezultati u odnosu na prethodna testiranja. Kada Varnish server kešira podatke, samo zahtjev prvog korisnika se šalje ka web serveru. Odgovor od web servera se kešira definisani vremenski period (podrazumijevano 120 sekundi). U okviru tog perioda svi naredni korisnički zahtjevi dobijaju isti odgovor kao i prvi korisnik. Vrijeme prvog odgovora zavisi od performansi Varnish servera, web servera i servera baza podataka. Vrijeme ostalih odgovora, u periodu dok su podaci keširani, zavisi direktno od performansi Varnish servera. Svi zahtjevi su prošli uspješno. Test je trajao 49 sekundi, pri čemu je minimalno vrijeme odgovora bilo 0.442 sekunde, maksimalno 17.581 sekunda, dok je srednje vrijeme odgovora bilo 9.224 sekunde. U odnosu na prethodne testove, standardna devijacija je znatno manja i iznosi 1.453 sekunde.

Testiranje je vršeno za 5000 zahtjeva, odnosno 5 puta se slalo po 1000 zahtjeva u okviru jedne sekunde. Poređenje rezultata testiranja početne arhitekture web aplikacija sa jednim web serverom i jednim serverom baza podataka u odnosu na arhitekturu web aplikacija sa Varnish klasterom, tri web servera koja rade u paraleli i Galera klasterom, izvršeno je tabelarno i u vidu grafikona.

Tabela 5.1 – Tabelarno poređenje rezultata testiranja početne arhitekture web aplikacija i arhitekture web aplikacija sa Varnish klasterom

	Vrijeme trajanja testa [s]	**Minimalno vrijeme odgovora [s]**	**Maksimalno vrijeme odgovora [s]**	**Srednje vrijeme odgovora [s]**	**Standardna devijacija [s]**
Početna arhitektura web aplikacija	575	1.843	136.795	105.588	27.478
Arhitektura web aplikacija sa Varnish klasterom i keširanjem	49	0.442	17.581	9.224	1.453



Slika 5.6 – Poređenje rezultata testiranja početne arhitekture web aplikacija i arhitekture web aplikacija sa Varnish klasterom

Varnish server znatno ubrzava rad web aplikacije keširanjem korisnički nezavisnih podataka, odnosno podataka koji su isti svim korisnicima web sajta. Ukoliko se pojave podaci koji su korisnički zavisni, takav tip podataka se ne kešira od strane Varnish servera.

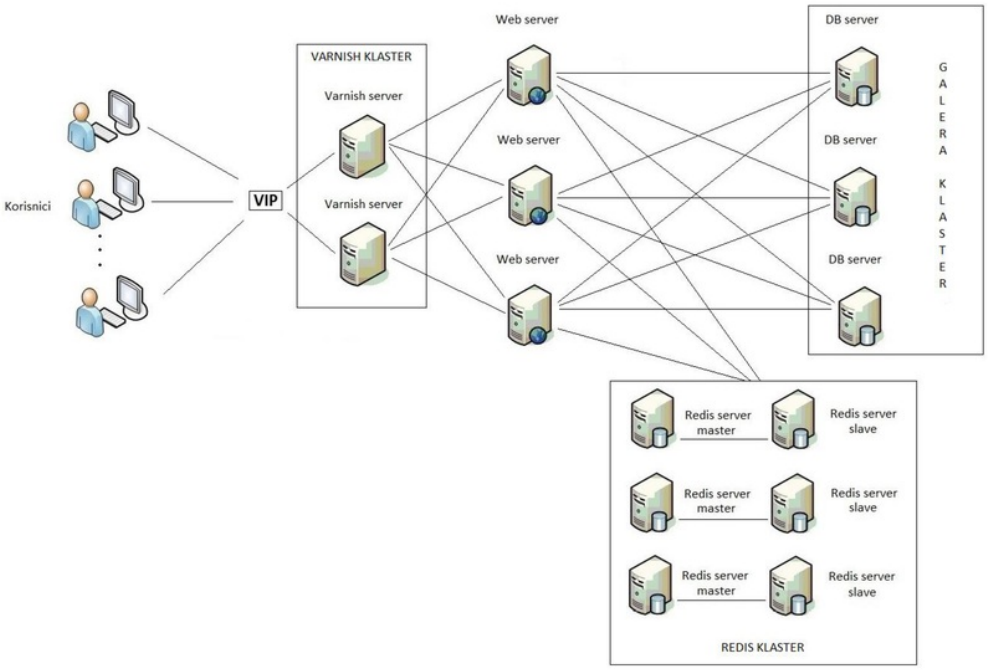
Da bi se prikupili korisnički zavisni podaci zahtjev mora stići do web servera na kojem se nalaze podaci o *sesiji* korisnika. Za primjer korisnički zavisnih podataka web aplikacije *Online news* mogu se uzeti odabrane vijesti. Svaki regularni korisnik web aplikacije nakon uspješne prijave dobija funkcionalnost za dodavanje i pregled odabranih vijesti. Odabrane vijesti se razlikuju za svakog korisnika. U tabeli *user_has_news* (Slika 5.2) čuvaju se podaci o odabranim vijestima korisnika i broj zapisa u toj tabeli iznosi ~ 320 000.

Sljedeći upit se koristi za prikupljanje 1000 najnovijih odabranih vijesti za jednog korisnika (user_id=37):

```
SELECT n.*,c.name as category_name, m.color as menu_color
FROM (SELECT * FROM news WHERE id IN (SELECT news_id FROM user_has_news
WHERE user_id=37) AND active=1 ORDER BY id DESC LIMIT 1000) n
  JOIN category c ON n.category_id=c.id
  JOIN menu m ON c.menu_id=m.id
ORDER BY n.id DESC;
```

Za izvršavanje prethodnog upita kojim se prikuplja 1000 odabranih vijesti od ~ 50 000 koje je korisnik odabrao potrebno je oko 3 sekunde. Povećavanjem broja vijesti koje se prikupljaju povećava se i potrebno vrijeme za izvršavanje upita.

Kako bi se ubrzao rad web aplikacije prilikom prikupljanja korisnički zavisnih podataka, ideja je da se u prethodno izloženoj arhitekturi web aplikacija koristi Redis kao predstavnik ključ-vrijednost NoSQL baza podataka. Redis baza podataka bi se koristila za keširanje korisnički zavisnih podataka. Da bi se dodatno osigurala i dostupnost umjesto jedne instance Redis baze podataka koristiće se Redis klaster. Prilikom zahtijevanja korisnički zavisnih podataka Varnish serveri ne keširaju podatke, zahtjeve propuštaju ka web serverima, a web serveri vraćaju odgovor krajnjim korisnicima putem Varnish servera. Da bi se saobraćaj za istog korisnika usmjeravao na isti pozadinski server od strane Varnish servera korištena je *The Hash Director* strategija za usmjeravanje saobraćaja.



Slika 5.7 – Arhitektura web aplikacija sa Varnish klasterom, tri web servera koja rade u paraleli, Redis klasterom i Galera klasterom

Funkcionisanje web aplikacije prilikom prikupljanja odabranih vijesti za nekog korisnika svelo bi se na sljedeće korake:

1. prilikom prijavljivanja korisnika na web aplikaciju prikupljaju se sve odabrane vijesti za tog korisnika i smještaju u Redis bazu podataka u vidu heš mape (idVijesti:vijest, vijest je u JSON formatu),
2. ukoliko zapis za prijavljenog korisnika već postoji u Redis bazi podataka briše se i smješta se novi,
3. prilikom prikupljanja odabranih vijesti korisnik pristupa Redis bazi podataka.

Ukoliko korisnik dodaje novu odabranu vijest ili uklanja postojeću, zahtjevi se šalju direktno ka MariaDB bazi podataka i dodatno se pokreće jedna nit koja dodaje ili briše vijest iz Redis baze podataka, kako se ne bi usporilo vraćanje odgovora korisniku.

Napisana je nova simulaciona skripta u programskom jeziku *Scala*, koja se koristi od strane programa *Gatling* za testiranje performansi prilikom prikupljanja korisnički zavisnih podataka. Testirale su se performanse prilikom prikupljanja 1000 najnovijih odabranih vijesti za jednog korisnika, koji je odabrao ~ 50 000 vijesti. Korisnik se prijavljuje na

aplikaciju i zahtjevi se šalju sukcesivno. Šalje se ukupno 100 zahtjeva za odabrane vijesti od strane jednog korisnika.

Rezultati testiranja prilikom prikupljanja odabranih vijesti bez korištenja Redis klastera su sljedeći:

```
Simulation OnlineNewsMariaDB completed in 275 seconds
Parsing log file(s)...
Parsing log file(s) done
Generating reports...

================================================================================
---- Global Information --------------------------------------------------------
> request count                                        101 (OK=101    KO=0     )
> min response time                                    602 (OK=602    KO=-     )
> max response time                                   3481 (OK=3481   KO=-     )
> mean response time                                  2716 (OK=2716   KO=-     )
> std deviation                                        340 (OK=340    KO=-     )
> response time 50th percentile                       2686 (OK=2686   KO=-     )
> response time 75th percentile                       2874 (OK=2874   KO=-     )
> response time 95th percentile                       3219 (OK=3219   KO=-     )
> response time 99th percentile                       3422 (OK=3422   KO=-     )
> mean requests/sec                                  0.366 (OK=0.366  KO=-     )
---- Response Time Distribution ------------------------------------------------
> t < 800 ms                                             1 (  1%)
> 800 ms < t < 1200 ms                                   0 (  0%)
> t > 1200 ms                                          100 ( 99%)
> failed                                                 0 (  0%)
```

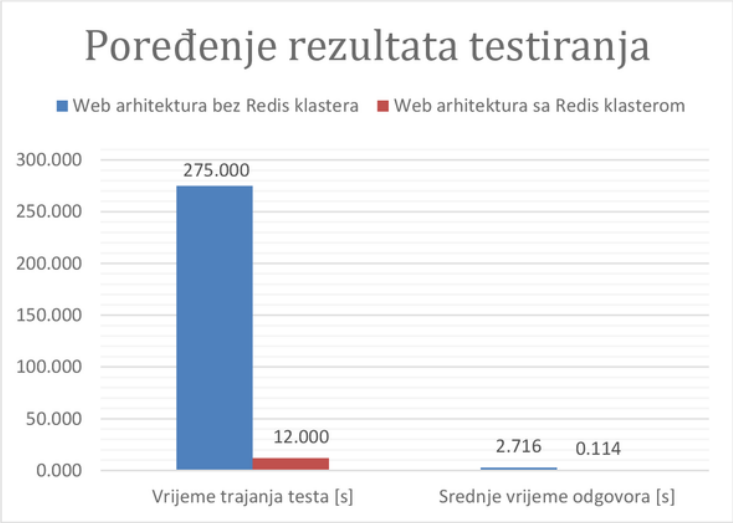
Svi zahtjevi su prošli uspješno. Za jedan zahtjev odziv je bio manji od 0.8 sekundi. To je minimalno utrošeno vrijeme od 0.602 sekunde i odnosi se na prijavljivanje korisnika. Vrijeme odgovora kod ostalih 100 zahtjeva za odabranim vijestima trajalo je preko 1.2 sekunde.

Rezultati testiranja prilikom prikupljanja odabranih vijesti korištenjem Redis klastera:

```
Simulation OnlineNewsRedis completed in 12 seconds
Parsing log file(s)...
Parsing log file(s) done
Generating reports...

================================================================================
---- Global Information --------------------------------------------------------
> request count                                        101 (OK=101    KO=0     )
> min response time                                     42 (OK=42     KO=-     )
> max response time                                   3287 (OK=3287   KO=-     )
> mean response time                                   114 (OK=114    KO=-     )
> std deviation                                        318 (OK=318    KO=-     )
> response time 50th percentile                         71 (OK=71     KO=-     )
> response time 75th percentile                         85 (OK=85     KO=-     )
> response time 95th percentile                        138 (OK=138    KO=-     )
> response time 99th percentile                        212 (OK=212    KO=-     )
> mean requests/sec                                 7.769 (OK=7.769  KO=-     )
---- Response Time Distribution ------------------------------------------------
> t < 800 ms                                           100 ( 99%)
> 800 ms < t < 1200 ms                                   0 (  0%)
> t > 1200 ms                                            1 (  1%)
> failed                                                 0 (  0%)
```

U oba slučaja proslijeđen je ukupno 101 zahtjev, pri čemu je jedan zahtjev upućen za prijavu korisnika, a ostalih 100 zahtjeva za prikupljanje odabranih vijesti. Korištenjem Redis klastera znatno se ubrzava prikupljanje korisnički zavisnih podataka. Svi zahtjevi su prošli uspješno. Test sa Redis klasterom trajao je 12 sekundi, pri čemu je minimalno vrijeme odgovora bilo 0.042 sekunde, maksimalno 3.287 sekundi, dok je srednje vrijeme odgovora bilo 0.114 sekunde. Standardna devijacija iznosi 0.318 sekundi. Može se uočiti da je srednje vrijeme odgovora palo sa 2.716 sekundi na 0.114 sekundi. Kod arhitekture web aplikacija bez Redis klastera, odabrane vijesti se prikupljaju iz MariaDB baze podataka prilikom svakog zahtjeva za odabranim vijestima. Za razliku od arhitekture web aplikacija bez Redis klastera, kod arhitekture web aplikacija sa Redis klasterom odabrane vijesti se prikupe samo jednom iz MariaDB baze podataka prilikom prijavljivanja korisnika, dok se svaki sljedeći zahtjev za odabranim vijestima šalje ka Redis bazi podataka. Na osnovu rezultata može se primijetiti da je za samo jedan zahtjev utrošeno više od 1.2 sekunde i da je to maksimalno vrijeme odgovora od 3.287 sekundi, a odnosi se upravo na prijavljivanje korisnika. Prijavljivanje korisnika je razlog veće devijacije, nego što bi bila da se posmatra samo prikupljanje odabranih vijesti iz Redis baze podataka. Vrijeme odgovora kod svih ostalih 100 zahtjeva za odabranim vijestima trajalo je ispod 0.8 sekundi.

Pri radu sa Redis klasterom, prijavljivanje korisnika znatno utiče na maksimalno vrijeme i standardnu devijaciju i čini oko četvrtine ukupno utrošenog vremena na test, a pri tom je neophodno prilikom prikupljanja odabranih vijesti. Upravo iz ovih razloga potrebno ga je uzeti u obzir prilikom poređenja rezultata ukupnog vremena trajanja testa i srednjeg vremena odgovora sa arhitekturom web aplikacija bez Redis klastera. Slična situacija se dešava i kod minimalno utrošenog vremena na odgovor kada se ne koristi Redis klaster. To je vrijeme koje se utroši na prijavljivanje korisnika. Pošto je logika funkcionisanja web aplikacije prilikom prikupljanja odabranih vijesti i korištenja arhitekture web aplikacija sa Redis klasterom promijenjena u odnosu na arhitekturu web aplikacija u kojoj se ne koristi Redis klaster, poređenje rezultata ima smisla jedino za ukupno vrijeme trajanja testa i srednje vrijeme odgovora.



Slika 5.8 - Poređenje rezultata testa arhitekture web aplikacija bez Redis klastera i arhitekture web aplikacija sa Redis klasterom

6. ZAKLJUČAK

Svakim danom bilježi se porast korištenja Interneta i vremena koje ljudi provode na Internetu, što se karakteriše pojmom *Big users*. Generiše se velika količina podataka putem društvenih mreža, geoinformacionih sistema, niza različitih senzora, senzorskih mreža, mobilnih aplikacija, aplikacija za praćenje logova i niza drugih sistema. Javlja se potreba za skladištenjem i obradom velike količine podataka (eng. *Big Data*), što je stvorilo nove izazove i potrebe za rješavanjem navedenih problema. Forma podataka sve više prelazi iz strukturisane u polustrukturisanu ili nestrukturisanu. Mnogi traže rješenja za probleme rada sa velikom količinom podataka u NoSQL bazama podataka, koje su mnogo fleksibilnije u odnosu na relacione baze podataka. NoSQL baze nemaju striktno definisanu šemu modela podataka i nude određenu slobodu prilikom skladištenja podataka. To nekad može predstavljati i problem, kada sa istom bazom podataka radi više različitih ljudi. Potrebno je uvoditi i usvojiti određene konvencije kako ne bi došlo do nekonzistentnosti podataka. Nestrukturisani podaci dolaze do izražaja sa pojavom uređaja koji se povezuju na Internet, počevši od mobilnih telefona, tableta, računara, kućnih aparata, pa do sistema instaliranih u automobilima, skladištima, bolnicama i niza drugih sistema. Javlja se novi pojam *Internet of Things* koji se veže za sve uređaje koji komuniciraju putem Interneta. Kako uređaji primaju razne informacije iz okruženja, kao što su lokacija, temperatura, vlažnost zraka i niz drugih mjernih parametara, koriste se od strane inovativnih preduzeća koja pokušavaju da unaprijede svoje poslovanje, povećaju efikasnost i smanje troškove. Nakon prikupljanja svih tih podataka i skladištenja u okviru relacionih i NoSQL baza podataka, potrebno ih je obraditi i na što je moguće jednostavniji način prikazati korisniku. Javlja se potreba za razvojem niza različitih aplikacija koje zadovoljavaju potrebe korisnika. Dolazi do pojave pojma *Cloud Computing* koji se veže za rad aplikacija u javnom, privatnom ili hibridnom oblaku (eng. *Cloud*). Najviše su zastupljene aplikacije koje se izvršavaju u javnom oblaku i pristupa im se putem Interneta. Da bi korisnici bili zadovoljni prilikom korištenja aplikacija, neophodno je obezbijediti da aplikacija radi nesmetano u slučaju otkaza bilo kojeg dijela sistema, ali i da ima dobar odziv kako korisnik ne bi izgubio interesovanje. Pored dobrih hardverskih komponenti sistema, dobrog mrežnog protoka i dobro razvijene aplikacije, neophodno je posvetiti pažnju i samoj arhitekturi aplikativnog sistema, kako bi se izvukao maksimum iz svih komponenti i kako bi sve prethodno navedeno funkcionisalo kao jedna cjelina. Aplikacija bi trebalo da održi performanse i nakon porasta broja korisnika.

U ovom radu dat je prijedlog višeslojne arhitekture web aplikacija sa visokim nivoom saobraćaja, koja je skalabilna i otporna na otkaze. Arhitektura se jednostavno proširuje sa porastom broja korisnika. Opisan je jedan stek tehnologija kojim se postiže dostupnost i skalabilnost cijelog sistema. Korištene su *open source* tehnologije i alati. Pošto je naglasak bio na skalabilnosti i dostupnosti sistema, sigurnost sistema nije razmatrana, ali je veoma bitna za rad aplikacije i potrebno je posvetiti posebnu pažnju toj oblasti. Izložena arhitektura web aplikacija je namijenjena isključivo za aplikacije kod kojih je naglasak na čitanju, a ne na upisu podataka. Ideja je bila da se ubrza rad takvih aplikacija i poveća njihova dostupnost. Keširanjem podataka postižu se znatno bolje performanse. Pokazana je velika prednost takve arhitekrure u odnosu na arhitekturu sa jednim web serverom i jednim serverom baza podataka. Pored boljih performansi, klasterizacijom je postignuta i veća dostupnost cijelog sistema.

Prije razvoja aplikacije i osmišljavanja cijele arhitekture na kojoj će se aplikacija izvršavati, neophodno je razumjeti potrebe krajnjih korisnika aplikacije, kao i šta se od cijelog sistema očekuje. Cilj je realizovati arhitekturu koja će podržati budući razvoj aplikacija i biti skalabilna, a istovremeno je potrebno razmišljati i o troškovima cijelog sistema. Svaka nova mašina, bilo da je virtuelna ili fizička, zahtijeva dodatne resurse i iziskuje dodatne troškove. Za krajnjeg korisnika je najbitniji intuitivan rad na aplikaciji, brz odziv prilikom upućenog zahtjeva i dostupnost aplikacije. Aplikacija se razvija zbog krajnjih korisnika i osnovni cilj je da korisnici budu zadovoljni.

Nakon implementacije odgovarajuće arhitekture i završenog razvoja aplikacije, neophodno je detaljno testirati rad cijelog sistema prije puštanja u produkciju, kako bi se što je moguće ranije uočili nedostaci sistema i otklonile greške. Naravno, teško je očekivati da sistem nakon puštanja u produkciju nema niti jednu grešku, ali potrebno je što je moguće više svesti te greške na minimum.

Dalje istraživanje u ovoj oblasti trebalo bi usmjeriti ka sigurnosti cijelog sistema i sigurnom prenosu podataka. Svaka web aplikacija koja je dostupna putem Interneta je podložna napadima malicioznih korisnika. Sigurnost web aplikacija je centralna komponenta web orijentisanih sistema. Jedan od zanimljivih aspekata istraživanja bio bi i unapređenje razvojnog i produkcionog okruženja uvođenjem monitoring sistema, koji bi vršili nadzor, analizu performansi izvršavanja i alarmiranje u slučaju problema u funkcionisanju web aplikacija. Takođe, zanimljiv pravac istraživanja bio bi implementacija

izloženog koncepta arhitekture web aplikacija korištenjem alternativnih tehnologija i komparacija sa implementiranim rješenjem u ovom radu.

LITERATURA

[1] "Web application architecture", posjećeno: 15.11.2017., https://stackify.com/web-application-architecture/.

[2] "What is N-Tier Architecture? How It Works, Examples, Tutorials, and More", posjećeno: 27.11.2017., https://stackify.com/n-tier-architecture/.

[3] "HTTP Request Methods", posjećeno: 20.12.2017., https://www.w3schools.com/tags/ref_httpmethods.asp.

[4] "3 Types Of Web Application Architecture", posjećeno: 22.12.2017., https://mobidev.biz/blog/3_types_of_web_application_architecture.

[5] "AJAX Introduction", posjećeno: 25.12.2017., https://www.w3schools.com/xml/ajax_intro.asp.

[6] D. Stjepanović, "Poređenje performansi Memcached i Redis servera", Elektrotehnički fakultet, Banja Luka, 2015.

[7] "Proxy servers", posjećeno: 23.01.2018, https://www.tutorialspoint.com/internet_technologies/proxy_servers.htm.

[8] T. Feryn, "Getting Started with Varnish Cache", O'Reilly Media, Inc., 2017.

[9] "Pacemaker Explained", posjećeno: 02.02.2018., https://clusterlabs.org/pacemaker/doc/en-US/Pacemaker/1.1/html/Pacemaker_Explained/index.html.

[10] "Corosync", posjećeno: 02.02.2018., http://corosync.github.io/corosync/.

[11] B. G. Raggad, "Information Security Management: Concepts and Practice", Taylor & Francis Group, 2010.

[12] "What is a web server", posjećeno: 01.03.2018., https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_web_server.

[13] S. Marić, D. Brđanin, Relacione baze podataka, Univerzitet u Banjoj Luci Elektrotehnički fakultet, Banja Luka, 2012.

[14] Sadalage P. J., Fowler M., "NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence", Crawfordsville, Indiana, 2012.

[15] "Galera cluster documentation", posjećeno: 12.03.2018., http://galeracluster.com/documentation-webpages/index.html.

[16] "About MariaDB", posjećeno: 12.03.2018., https://mariadb.org/about/.

[17] "MariaDB", posjećeno: 12.03.2018., https://mariadb.com/sites/default/files/MariaDB.pdf.

[18] "An introduction to Redis data types and abstractions", posjećeno: 02.04.2018., http://redis.io/topics/data-types-intro.

[19] "Redis Cluster Specification", posjećeno: 03.04.2018., https://redis.io/topics/cluster-spec.

Arhitektura web aplikacija sa visokim nivoom saobraćaja

ORIGINALITY REPORT

0%

SIMILARITY INDEX

PRIMARY SOURCES

EXCLUDE QUOTES	OFF	EXCLUDE MATCHES	< 1%
EXCLUDE BIBLIOGRAPHY	OFF		