Fakulteta za elektrotehniko,
računalništvo in informatiko
Smetanova ulica 17
2000 Maribor, Slovenija

Janez Kremzer

IMPLEMENTACIJA GRUČE APLIKACIJSKIH STREŽNIKOV S KNJIŽNICO OWIN

Diplomsko delo

Maribor, junij 2016

IMPLEMENTACIJA GRUČE APLIKACIJSKIH STREŽNIKOV S KNJIŽNICO OWIN

Diplomsko delo

Študent:	Janez Kremzer
Študijski program:	Visokošolski študijski program Računalništvo in informatika
Smer:	Programska oprema
Mentor:	doc. dr. Milan Ojsteršek
Lektorica:	Nuša Fujan, prof. slovenščine in ruščine

SKENIRAJ SKLEP

IMPLEMENTACIJA GRUČE APLIKACIJSKIH STREŠNIKOV S KNJIŽNICO OWIN

Ključne besede: aplikacijska gruča, OWIN, spletna aplikacija, porazdeljeni sistemi**,** Raft

UDK: xxxxxx

Povzetek

Aplikacijske gruče temeljijo na osnovi porazdeljevanja izvajanja aplikacije z več strežniki z namenom zagotavljanja boljše zanesljivosti delovanja in možnosti skaliranja aplikacije v primeru večje obremenitve sistema. V diplomskem delu smo se osredotočili na gručo, ki lahko izvaja eno ali več spletnih aplikacij. Spoznali bomo izravnalce obremenitve (angl. load balancer), katerih naloga je enakomerno porazdeliti izvajanje zahtevkov po celotni gruči. Omenili smo pomen interne komunikacije in pa algoritme soglasja, ki v aplikacijski gruči zanesljivo izmenjavo internih informacij. Kot primer algoritma soglasja bomo podrobneje pogledali algoritem Raft ter izvedli njegovo implementacijo. Prav tako smo s pomočjo knjižnice OWIN implementirali izravnalca obremenitve in aplikacijsko gručo.

IMPLEMENTATION OF AN APPLICATION SERVER CLUSTER USING THE OWIN LIBRARY

Key words: application cluster, web application, distributed system, Raft, OWIN

UDK: xxxxxx

Abstract

Application cluster based on the distribution of the load on multiple servers in order to ensure better operational reliability and possibility of scaling out applications in case of increasing load. Thesis is focused on a cluster of web applications. This kind of applications for external communication use HTTP protocol. We mentioned load balancers, their main task is to distribute load across the cluster. Cluster also needs reliable medium for exchange of internal information, which usually use solutions based on consensus algorithm. As an example of consensus algorithm, we used Raft and we implemented it. The last part of work was a practical implementation with OWIN library. We implemented a load balancer and an application cluster.

KAZALO

1 UVOD 1

2 OSNOVNI POJMI SPLETNE APLIKACIJE 4

2.1 Seja 4

2.2 Shranjevanje podatkov na strežniku 5

3 VRSTE GOSTOVANJA APLIKACIJSKE GRUČE 6

3.1 Postavitev na strojno opremo računalnika (na fizični strežnik) 6

3.2 Virtualni strežnik 7

3.3 Linux kontejner (LXC) 9

4 KOMUNIKACIJA Z ZUNANJIM SVETOM 11

4.1 Oglaševanje preko imenskega strežnika DNS 11

4.2 Uporaba plavajočih naslovov IP 12

5 INTERNA KOMUNIKACIJA V GRUČI 14

6 RAFT 15

6.1 Osnove algoritma »Raft« 17

6.2 Volitve vodje 17

6.3 Sprejemanje ukazov odjemalcev 18

7 OWIN 20

7.1 Definicija 20

7.2 Zgradba 21

7.2.1 Slovar okolja (angl. environment dictionary) 21

7.2.2 Generični funkcijski delegat (angl. generic func delegate). 22

8 IMPLEMENTACIJA ALGORITMA RAFT S KNJIŽNICO OWIN 23
8.1.1 Implementacija strežniške knjižnice z algoritmom Raft 23
8.1.2 Implementacija komunikacije odjemalca z gručo 26

9 IMPLEMENTACIJA IZRAVNALCA OBREMENITVE 28

10 PRIMERJAVA GRUČE S SAMOSTOJNO APLIKACIJO 30
10.1.1 Testiranje 30
10.1.2 Rezultati 30
10.1.3 Prednosti aplikacijske gruče na podlagi testiranja 32

11 SKLEP 33

12 LITERATURA 34

KAZALO SLIK

SLIKA 1: KOMUNIKACIJA ODJEMALEC/STREŽNIK (VIR: WIKIPEDIA) 4
SLIKA 2: SEJA MED ODJEMALCEM IN STREŽNIKOM 5
SLIKA 3: NADZORNA PLOŠČA ANSIBLE (VIR: ANSIBLE.COM) 7
SLIKA 4: NADZORNA PLOŠČA KUBERNETES (VIR: KUBERNETES.IO) 10
SLIKA 5: SLIKA PRIKAZUJE ORODJE NSLOOKUP, KI JE NAMENJEN VPOGLEDU V STREŽNIK DNS 11
SLIKA 6: PRIMER POSTAVITVE DVEH IZRAVNALCEV OBREMENITVE NGINX, KI SKRBITA ZA OBDELAVO ZAHTEVKOV MED UPORABNIKI IN ZALEDNIMI STREŽNIKI V GRUČI (VIR: NGINX.COM) 13
SLIKA 7: OSNOVE ALGORITMA RAFT (VIR: IN SEARCH OF AN UNDERSTANDABLE CONSENSUS ALGORITHM) (V ANGLEŠKEM JEZIKU) 16
SLIKA 8: SPREMINJANJE MANDATOV (VIR: IN SEARCH OF AN UNDERSTANDABLE CONSENSUS ALGORITHM) 17
SLIKA 9: GRAFIČNI PRIKAZ ZAPISOV V DNEVNIKU ZAPISOV (VIR: IN SEARCH OF AN UNDERSTANDABLE CONSENSUS ALGORITHM) 19
SLIKA 10: PRIMER GENERIČNEGA FUNKCIJSKEGA DELEGATA (VIR: WWW.CODEPROJECT.COM) 22
SLIKA 11: GNEZDENJE VMESNE KODE (VIR: WWW.CODEPROJECT.COM) 22
SLIKA 12: PRIMER ZAGONSKE METODE STREŽNIKA 23
SLIKA 13: RAZREDNI DIAGRAM STREŽNIKA RAFT 24
SLIKA 14: PRIKAZ PRINCIPA DELOVANJA INICIALIZACIJE VLOG STREŽNIKOV. V TEM PRIMERU JE PRVI STREŽNIK VODJA IN POŠILJA IDENTIFIKATOR ZA MANDAT 27 OSTALIM STREŽNIKOM. 25
SLIKA 15: SLIKA PRIKAZUJE STANJE STREŽNIKA, DOSEGLJIVEGA PREKO VMESNIKA ZA INTERNO KOMUNIKACIJO 25
SLIKA 16: PRIKAZ SEZNAMA SHRANJENIH VREDNOSTI 27
SLIKA 17: IMPLEMENTACIJA VMESNE KODE IZRAVNALCA 28
SLIKA 18: ODGOVOR ZALEDNEGA STREŽNIKA 30

KAZALO TABEL

TABELA 8.1: VSEBINA KLJUČEV ZAHTEVKOV SLOVARJA OKOLJA 21
TABELA 8.2: VSEBINA KLJUČEV ODGOVOROV SLOVARJA OKOLJA 21
TABELA 9.1: METODE, DOSTOPNE ODJEMALCU PREKO UKAZOV REST........ 26
TABELA 11.1: SAMOSTOJNI STREŽNIK 31
TABELA 11.2: EN ZALEDNI STREŽNIK 31
TABELA 11.3: DVA ZALEDNA STREŽNIKA 31
TABELA 11.4: TRIJE ZALEDNI STREŽNIKI 31
TABELA 11.5: ŠTIRJE ZALEDNI STREŽNIKI 31
TABELA 11.6: PET ZALEDNIH STREŽNIKOV 32

1 UVOD

V nalogi se bomo osredotočili na izvajanje aplikacije na več fizičnih ali virtualnih strežnikih (izvajanje na aplikacijski gruči). Namen aplikacijske gruče je zagotavljati nemoteno delovanje storitve. Aplikacija, ki teče na enem samem strežniku, je namreč občutljiva na različne dogodke, ki lahko povzročijo nedosegljivost nudenja storitve aplikacije zaradi izvajanja, ki se lahko konča zaradi ene same točke odpovedi (angl. single poinf of failure).

V primeru aplikacijske gruče drastično izboljšamo dosegljivost storitve. Sistem zasnujemo tako, da celotna storitev še vedno deluje, kljub odpovedi določenega člena.

V nadaljevanju se bomo osredotočili na aplikacijsko gručo, namenjeno spletni aplikaciji. Tipična spletna aplikacija nudi neko storitev, ki je za uporabnike dosegljiva preko protokola HTTP. Protokol HTTP je v tem primeru edina komunikacijska pot med zunanjim svetom in aplikacijo. Če želimo aplikacijo narediti bolj odporno na odpovedi, moramo zagotoviti več virov, ki odgovarjajo na odjemalčeve zahtevke http. Za komunikacijo med uporabnikom in spletno aplikacijo to pomeni, da lahko vsak zahtevek uporabnika servira drugi strežnik. Zato odpoved enega od strežnikov ni več pomembna za zagotavljanje storitve kot celote.

Spletne aplikacije navadno uporabniku ponujajo informacije oz. storitve. V kolikor so te informacije shranjene v datotekah html, nam zgornji model zadošča, saj ni pomembno, kateri od strežnikov nam je vrnil podatek. Vemo, da se podatki na nobenem strežniku ne morejo spremeniti, kar pomeni, da nam storitev vedno vrača konsistentne podatke.

V realnosti je sistemov z vnaprej pripravljenimi datotekami HTML zelo malo. Najpogosteje imajo pri spletnih sistemih prav uporabniki možnost, da podatke spreminjajo sami. Prav tako sodobne rešitve omogočajo iskanje po podatkih, pri katerih je eden od vhodnih argumentov funkcionalnosti iskalnika tudi domena uporabnika. V spletnih aplikacijah so v ta namen nastale uporabniške seje, podatki pa se obdelujejo preko podatkovnih baz.

Če predpostavimo, da imamo spletno aplikacijo, kjer uporabnik preko istega spletnega vmesnika obdeluje podatke, kasneje pa te svoje podatke želi priklicati in jih prikazati na različne tekstovne/grafične načine, ugotovimo, da sistem z enostavnim dodajanjem

spletnih strežnikov ne bo zadostoval našim potrebam. Najmanj, kar potrebujemo, je podatkovna baza, ki je dostopna z vseh strežnikov, poleg tega pa moramo zagotoviti tudi uporabniško sejo, do katere lahko vsi strežniki v gruči dostopajo.

Omogočanje dostopa do podatkovne baze ponavadi ni velik izziv. Skupna seja pa je najbolj običajno realizirana kot hramba tipa ključ/vrednost (key/value store) v lokalnem pomnilniku strežnika, kar pa je potrebno v primeru gruče realizirati drugače, in sicer tako, da je dostop do omenjene hrambe seje omogočen z vseh strežnikov. Če uporablja spletna aplikacija lokalne datoteke za hranjenje podatkov, je hranjenje potrebno realizirati z mrežnim dostopom, dosegljivim vsem strežnikom v gruči, ali pa z namensko storitvijo, ki enakovredno shranjuje datoteke na vse strežnike.

Delovanje storitve pa lahko ohromi tudi povečano število uporabnikov. Večanje števila uporabnikov vedno pomeni večjo porabo lokalnih virov. Ker pa imajo vsi viri omejitve, to lahko pripelje do slabe odzivnosti sistema ali celo do nedelovanja le-tega. Pri običajni, enostrežniški rešitvi, rešujemo situacijo s povečevanjem virov (angl. scale up), ki pa tudi ima zgornjo omejitev. Poleg tega večkratno spreminjanje osnovnega okolja pri enostrežniški rešitvi povzroči začasno nedelovanje zaradi planirane nadgradnje.

V primeru, ko storitev upravlja gruča, se celotna obremenitev virov porazdeli po gruči. Poleg tega pa obstaja več načinov, kako razširiti delovno gručo strežnikov med samim delovanjem, kar pomeni, da lahko povečamo razpoložljive vire med samim delovanjem storitve. Tak način je med prvimi omenjal Amazon z njihovo storitvijo EC2 (Elastic Computer Cloud) v komercialne namene.

Vendar pa je pri gruči zelo pomembno, da so vsi strežniki enako obremenjeni. To nalogo navadno opravljajo LoadBalancerji oz. »izravnalci obremenitve«. Odločitev, kateri strežnik bo opravil določeno nalogo, pa sprejmejo na podlagi algoritmov za razvrščanje obremenitev.

Sistemi, ki smo jih omenjali sedaj, so sestavljeni iz več strežnikov. Delovanje in obremenjenost teh strežnikov pa je smiselno spremljati na enem mestu. V ta namen nam služijo namenske nadzorne plošče, imenovane «orkestratorji«.

Podatke, ki si jih strežniki znotraj aplikacijske gruče izmenjujejo, je potrebno hraniti na zanesljiv in neodvisen način. V te namene se navadno uporabljajo rešitve, ki uporabljajo algoritme soglasja. Podrobneje si bomo ogledali algoritem Raft.

V nadaljevanju bomo pogledali še knjižnico OWIN in s pomočjo te knjižnice implementirali algoritem Raft ter izravnalca obremenitve.

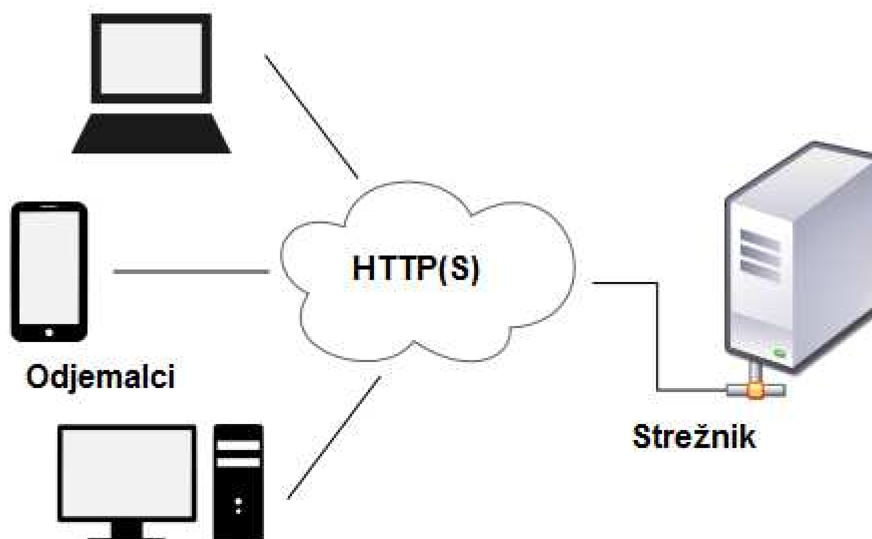
V drugem poglavju bo delo potekalo z implementacijo algoritma Raft. Na tak način bomo praktično spoznali enega od načinov izmenjevanja podatkov znotraj gruče. Preučili bomo koliko vpliva na gručo ima dodajanje novih strežnikov v gručo in koliko vpliva nedelovanje posamezni strežnik v gruči.

Nadaljevali pa bomo z implementacijo izravnalca obremenitve, ki opravlja pomembno vlogo v aplikacijski gruči. S pomočjo testiranja obremenitve, bomo preverili koliko vpliva število aplikacijskih strežnikov na boljše delovanje aplikacije. Zanimala na bo zanesljivost delovanja ter vpliv na odzivnost celotne aplikacije.

Obe omenjeni implementaciji bomo opravili s pomočjo knjižnice OWIN.

2 OSNOVNI POJMI SPLETNE APLIKACIJE

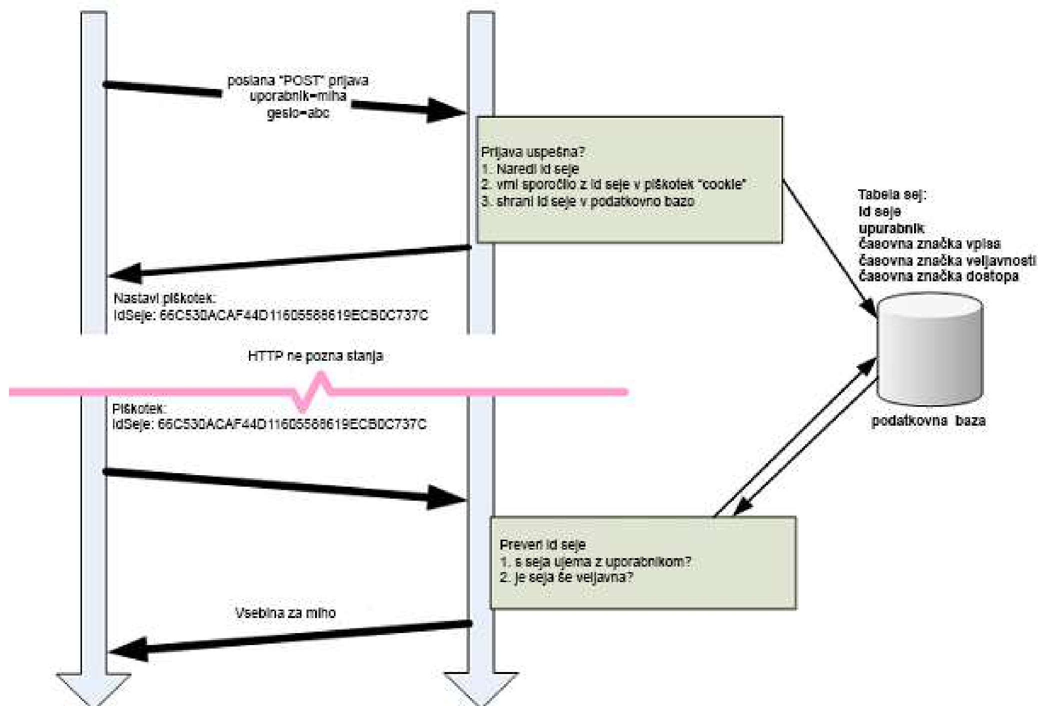
Spletna aplikacija je zgrajena po modelu odjemalec/strežnik. Vsi zahtevki med odjemalcem in strežnikom se izvajajo po protokolu HTTP(S). To pomeni, da se celoten uporabniški vmesnik in vsi podatki med strežnikom in odjemalcem prenesejo po omenjenem protokolu. Zahtevki HTTP(S) se na odjemalcu navadno izvedejo v spletnem brskalniku. Sodobne aplikacije navadno ločujejo zahtevke z uporabniškim vmesnikom in podatki, kar omogoča boljšo uporabniško izkušnjo, saj se uporabniški vmesnik prenese na začetku, kasneje pa se z interakcijo v aplikaciji prenašajo še podatki. Podatki se v tem primeru največkrat prenašajo v notaciji JSON.



Slika 1: komunikacija Odjemalec/strežnik (vir: Wikipedija)

2.1 Seja

Ko uporabnik prvič vzpostavi stik s spletno aplikacijo, se na strežniku zanj kreira seja. Sejo si lahko predstavljamo kot prostor na strani strežnika, kamor aplikacija lahko zapisuje začasne podatke, vezane na točno določenega uporabnika. Vez med uporabnikom in sejo je navadno identifikacijska številka seje, ki jo strežnik pošlje odjemalcu. Možna realizacija prenosa identifikacijske številke seje je piškotek (angl. cookie) ali pa paket znotraj glave http. Kasneje odjemalec številko seje vrača strežniku.



Slika 2: Seja med odjemalcem in strežnikom

2.2 Shranjevanje podatkov na strežniku

Kot smo omenili, v uvodu spletna aplikacija pogosto nudi uporabniku upravljanje nad podatki. Ti podatki so na strežniku najpogosteje shranjeni v podatkovno bazo, ki služi kot trajni medij za shranjevanje podatkov. Tip podatkovne baze je odvisen od vrste aplikacije, izbira pa je domena načrtovalca spletne aplikacije. Pri načrtovanju spletne aplikacije, ki bo delovala v gruči, je potrebno izbrati podatkovno bazo, ki omogoča enakovredni dostop do istih podatkov vsem strežnikov v gruči.

Občasno želimo na strežniku spletne aplikacije hraniti lokalne datoteke. Primer: pri spletnih aplikacijah se za multimedijske vsebine uporabnikov (slike, video …) pogosto odločimo, da te vsebine zaradi njihove velikosti shranjujemo direktno na datotečni sistem. V primeru spletne aplikacije v gruči moramo te datoteke bodisi prestaviti na mrežni disk, ki je dosegljiv iz vseh aplikacijskih strežnikov, bodisi moramo narediti storitev za interni dostop do omenjene vsebine preko drugega protokola.

3 VRSTE GOSTOVANJA APLIKACIJSKE GRUČE

Obstajajo trije načini vzpostavitve aplikacijske gruče in njenega izvajalnega okolja: postavitev na strojno opremo računalnika (na fizični strežnik),

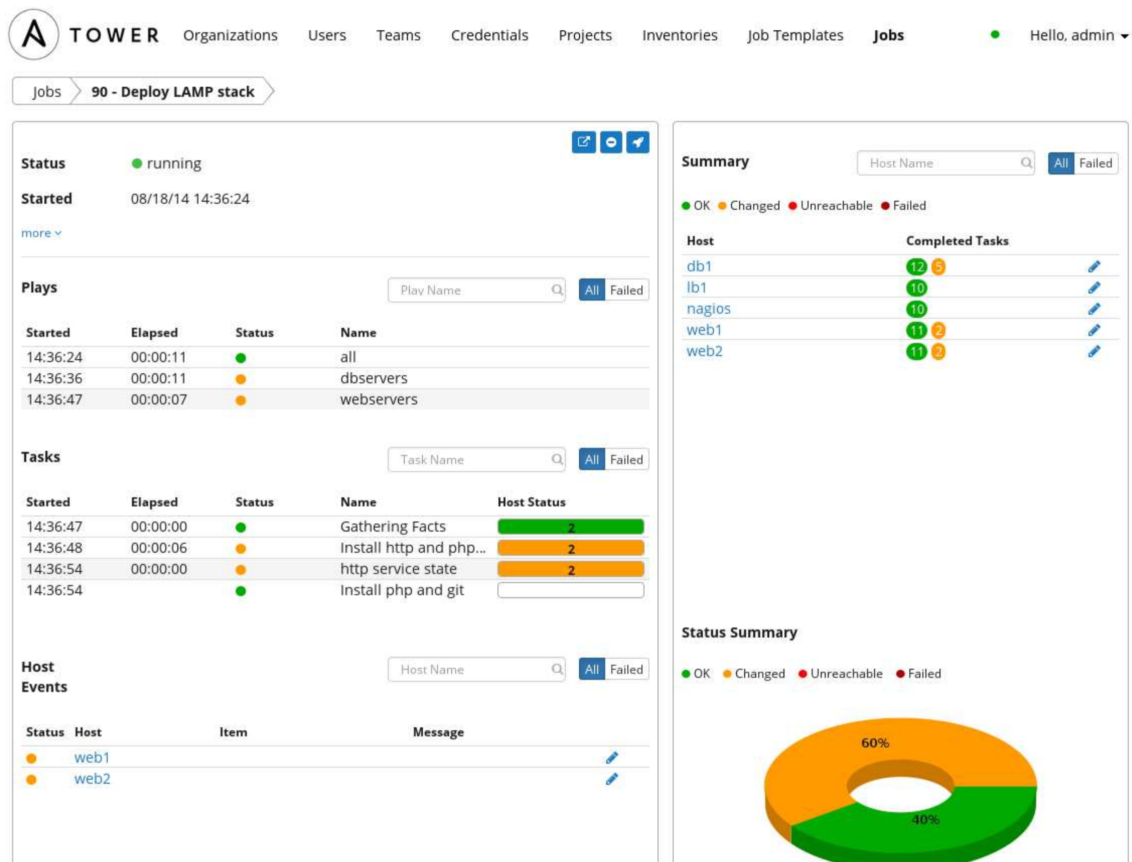
- postavitev na virtualni strežnik ali
- postavitev v Linux kontejnerja (LXC).

Pri vseh treh načinih govorimo o porazdelitvi strežniškega dela na različnem nivoju abstrakcije med izvajalnim okoljem in strežniškim izvajalnim okoljem. Aplikacijska gruča je odvisna od operacijskega sistema predvsem v tej meri, kako enostavno jo je razširiti z več pomnilnika, z več procesorji, z več diski ali na več računalnikov.

3.1 Postavitev na strojno opremo računalnika (na fizični strežnik)

V tem primeru namestimo operacijski sistem posameznega strežnika direktno na strojno opremo računalnika. Prednost tega načina je dostop do vseh strojnih virov, ki jih nudi računalnik. Dobra stran pa je tudi varnost sistema, predvsem na račun (fizične) izoliranosti med različnimi strežniki.

Slabša stran uporabe fizičnega strežnika je predvsem v bolj kompleksni namestitvi informacijskega sistema, ki ga je potrebno namestiti na strežnik. V velikih gručah si sicer lahko pomagamo z avtomatiziranim načinom namestitve okolja. Primer take rešitve je Red Hatova rešitev »Ansible«. Vendar zadeva nikoli ni tako enostavna kot pri virtualnem strežniku, saj zahteva dodaten strežnik s programsko opremo »Ansible«, ki nadzoruje gručo vanj povezanih strežnikov. Nam pa omenjena rešitev nudi tudi možnost spremljanja obremenitev in posodobitev posameznega strežnika ter prikaže morebitne okvare, kar je v večjih sistemih velika prednost.



Slika 3: Nadzorna plošča Ansible (vir: ansible.com)

3.2 Virtualni strežnik

Virtualni strežnik pomeni, da operacijski sistem strežnika namestimo na hipervizor, ki teče direktno na strojni opremi računalnika. Prednost tega načina je, da je operacijski sitem, kamor nameščamo aplikacijsko gručo, manj odvisen od strojne opreme, saj hipervizor skrbi tudi za določen nivo abstrakcije. Virtualni strežnik poganja hipervizor. Hipervizorji pa se navadno znajo povezati v gručo. Ta lastnost pomeni večjo zanesljivost gostovalnega virtualnega strežnika in enostavnejšo razširitev fizičnih virov celo med delovanjem gruče.

Veliko rešitev virtualizacije pa omogoča spremljanje in stanje porabljenih virov, tako fizičnih kot virtualnih (dodeljenih virtualnim računalnikom). Ker je hipervizor programski vmesnik in virtualni strežnik posebni proces, ki je na njem zagnan, nam tak pristop

omogoča enostavno možnost za avtomatiziran vmesnik za izdelavo novega virtualnega strežnika.
To nam v aplikacijski gruči omogoča možnost avtomatskega razširjanja gruče v okviru fizičnih virov.

3.3 Linux kontejner (LXC)

Linux kontejner v svetu programske opreme pomeni, da je eden ali več procesov zagnanih kot izolirana skupina na skupnem jedru Linux operacijskega sistema. Za gradnjo aplikacijske gruče je rešitev postala zanimiva predvsem z nastankom »Dockerja«. »Docker« je omogočil vmesnik, ki na zelo enostaven način omogoča porazdelitev in zagon aplikacije v kontejnerju. Aplikacija je v primeru »Dockerja« predstavljena z »docker sliko«. V sliki je zapisano vse potrebno za zagon aplikacije. »Docker sliko« največkrat sestavlja en proces, aplikacijo pa navadno sestavlja več procesov. V ta namen pozna »Docker« datoteko za porazdelitev več slik hkrati. Datoteka se imenuje »Docker Compose«. V njej je zapisan vrstni red zagona slik in odvisnosti med njimi.

Razširitev aplikacije z »Dockerjem« pomeni zagon več slik hkrati, kar predstavlja najbolj enostaven način strojnih virov od vseh do sedaj naštetih metod.

Naslednja prednost uporabe kontejnerjev je manjša mera abstrakcije, ki zmanjša izgube zmogljivosti in kapacitet strojnih virov.

V zadnjem času se je pojavilo veliko rešitev, ki omogočajo izgradnjo gruče kontejnerjev. Med najbolj odmevnimi je Docker Swarm, Kubernetes.

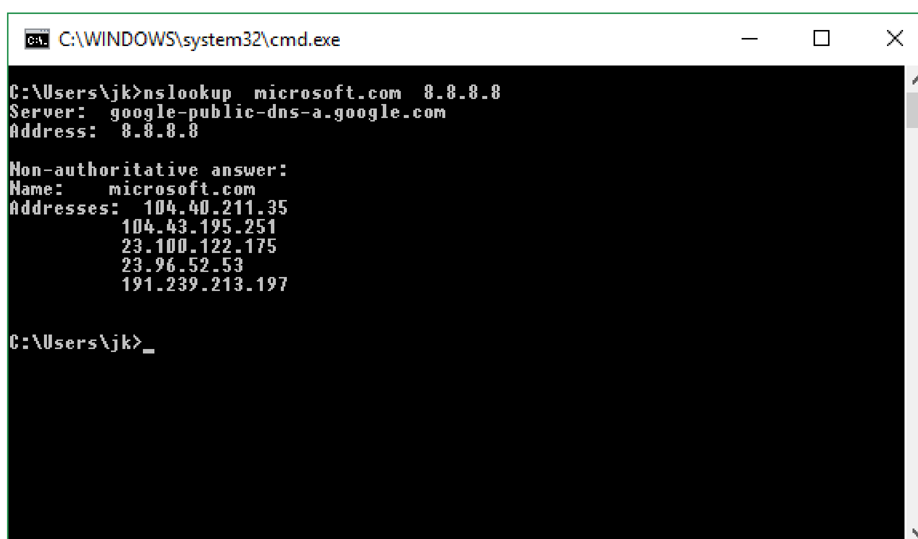
Slika 4: Nadzorna plošča Kubernetes (vir: kubernetes.io)

4 KOMUNIKACIJA Z ZUNANJIM SVETOM

Kot smo že uvodoma omenili, do spletne aplikacije dostopamo preko protokola HTTP. V primeru aplikacijske gruče smo aplikacijo razširili na več strežnikov. Vendar morajo uporabniki do aplikacije še vedno dostopati preko ene točke.

4.1 Oglaševanje preko imenskega strežnika DNS

Najbolj preprost način, kako oglaševati naslove IP strežnikov v gruči, je uporaba imenskega strežnika DNS.



Slika 5: Slika prikazuje orodje nslookup, ki je namenjeno vpogledu v strežnik DNS.

Ta način deluje na predpostavki, da vsak odjemalec uporabi prvi vrnjen naslov IP strežnika.
Obremenitev se nato izravnava tako, da imenski strežnik za vsak zahtevek vrne spremenjen vrstni red strežnikov, vpisanih pod določeno domeno. Uporabljen je algoritem krožnega dodeljevanja naslovov IP (angl. round-robin).

Ta metoda ne preverja, ali je posamezni strežnik v gruči dosegljiv ali bolj obremenjen, zato se lahko zgodi, da je uporabnik preusmerjen na nedelujoč strežnik.

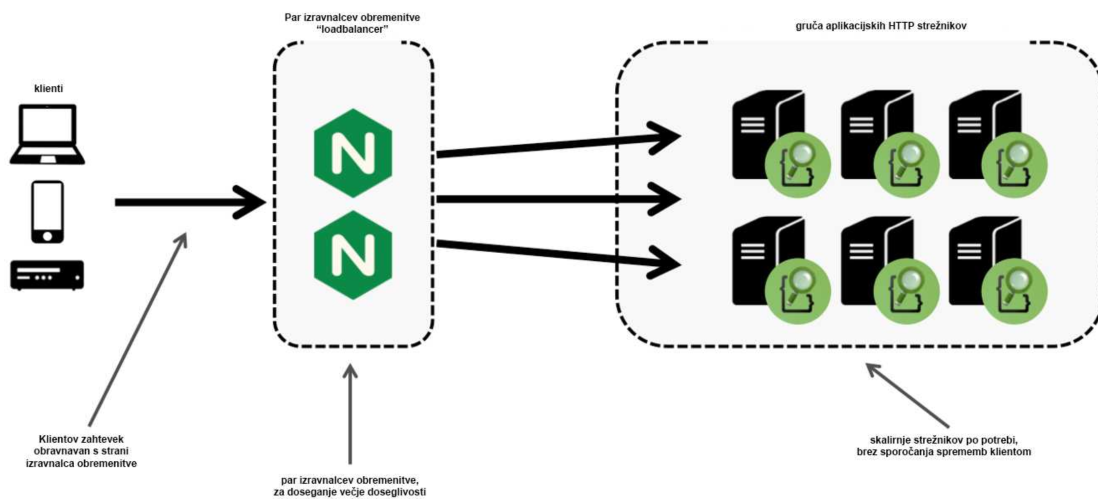
4.2 Uporaba plavajočih naslovov IP

Uporaba plavajočih naslovov IP (angl. floating IP) pomeni, da je vsem strežnikom v gruči dodan skupen naslov IP. Uporabnik zahtevke pošilja na omenjeni plavajoči IP, odgovor pa prejme od enega izmed strežnikov v gruči.

V okolju Microsoft Windows nam v ta namen služi aplikacija NLB (angl. network load balancing). V okolju Linux pa podobno funkcionalnost dosežemo z aplikacijo Keepalived. Obe rešitvi omogočata izravnavo obremenitve in delujeta na stopnji 4 modela OSI.

Ker obe rešitvi delujeta vključno na mrežnem protokolu in se ne zavedata stanja aplikacije, se navadno odločimo za dvonivojsko aplikacijsko gručo. Prvi nivo, ki je povezan z zunanjim svetom, je aplikacija, ki je namenjena izravnavi obremenitve (angl. load balancer). Ta aplikacija ima seznam zalednih strežnikov, ki izvajajo uporabnikove zahteve. Vse strežnike v seznamu aplikacija spremlja, zato ima v vsakem časovnem intervalu trenutno obremenitev posameznega strežnika in pa tudi morebitne okvare določenih strežnikov. Tak način nam omogoča enostavno dodajanje in odvzemanje strežnikov, ne da bi vplivali na ustavitev storitve.

Aplikacijo, namenjeno izravnavi obremenitve, pogosto postavljamo v paru. Na ta način povečamo zanesljivost celotne aplikacijske gruče, saj je v primeru uporabe ene instance aplikacije delovanje celotne gruče odvisno od delovanja omenjene instance.



Slika 6: Primer postavitve dveh izravnalcev obremenitve NGINX, ki skrbita za obdelavo zahtevkov med uporabniki in zalednimi strežniki v gruči (vir: nginx.com)

Računalništvo v oblaku in programsko definirana računalniška (kratica SDN – angl. Software defined network) mreža pogosto poznata termin plavajoči naslov IP. To pomeni, da kadar postavljamo aplikacijsko gručo v takem okolju, ni potrebe po dodatnih rešitvah za zagotavljanje plavajočega naslova IP na nivoju operacijskega sistema.

5 INTERNA KOMUNIKACIJA V GRUČI

Interna komunikacija v gruči poteka preko računalniškega omrežja, ki pa je navadno ločeno od javnega omrežja. V to skupino spada promet, ustvarjen med izravnalcem obremenitve in strežniki z vsebino. Pogosto pa po interni komunikaciji poteka promet, ki sporoča stanje posameznega elementa na strežniku.

Do stanja posameznega strežnika/gruče uporabniških sej je potreben dostop iz vseh strežnikov.

Za vse informacije o stanju gruče, uporabniških sej in ostalih za aplikacijo pomembnih podatkov uporabljamo upravljavca gruče. Primer takšne aplikacije sta Apache ZooKeeper in Peacemaker. Skupna tem aplikacijam je implementacija zanesljivega medija za shranjevanje in izmenjavo informacij. Podatkovna struktura medija za shranjevanje je pogosto kar ključ/vrednost z razliko, da so podatki zaradi zanesljivosti replicirani na več strežnikov, ki opravljajo nalogo upravljavca gruče.

Osnovna težava replikacije podatkov na teh strežnikih je zagotavljanje konsistentnega stanja vseh zapisov, saj je osnovno načelo gruče, da mora biti celotno delovanje neodvisno od odpovedi enega izmed strežnikov. Za reševanje te težave uporabljamo algoritme soglasja (angl. consensus algorithm), ki delujejo tako, da je določen zapis potrjen šele, ko ga potrdi večina strežnikov v gruči.

Med algoritme soglasja spadata algoritma Paxos in Raft. Slednjega bomo v nadaljevanju bolj podrobno opisali in izvedli preprosto implementacijo.

6 RAFT

Raft je algoritem za upravljanje repliciranega dnevnika zapisov in spada med algoritme soglasja. Kot smo že omenili, so algoritmi primerni za zagotavljanje zanesljivega medija shranjevanja informacij znotraj aplikacijske gruče. Pri algoritmih soglasja zahtevek postane veljaven, ko ga potrdi večina strežnikov v gruči.

Raft implementira soglasje z izvolitvijo vodilnega strežnika v gruči. Vodilni strežnik je odgovoren za upravljanje dnevnika zapisov podatkov. Vodja sprejme ukaze odjemalcev in zahtevek razpošlje na vse strežnike v svoji gruči. Ko dobi pozitivne odgovore vsaj polovice strežnikov, je zapis potrjen. V primeru, da postane vodilni strežnik nedosegljiv, ostali strežniki izvolijo novega vodjo po enakem postopku, kot je bil izvoljen predhodni vodja.

Raft razdeli problem soglasja na tri relativno neodvisne podprobleme:

- **Volitve vodje:** Ko se gruča zažene ali postane trenutni vodja nedosegljiv.
- **Replikacija** dnevnika zapisov**:** Vodja razpošlje odjemalčev ukaz po celotni gruči. Ko večina strežnikov zahtevek potrdi, je potrjen tudi zapis zahtevka.
- **Varnost:** Vsak zapis v dnevnika zapisov ima svoj identifikator. Strežniki ne potrdijo zapisa, kjer se ukaz in identifikator razlikujeta.

State

Persistent state on all servers:

(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:

(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§5.2):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

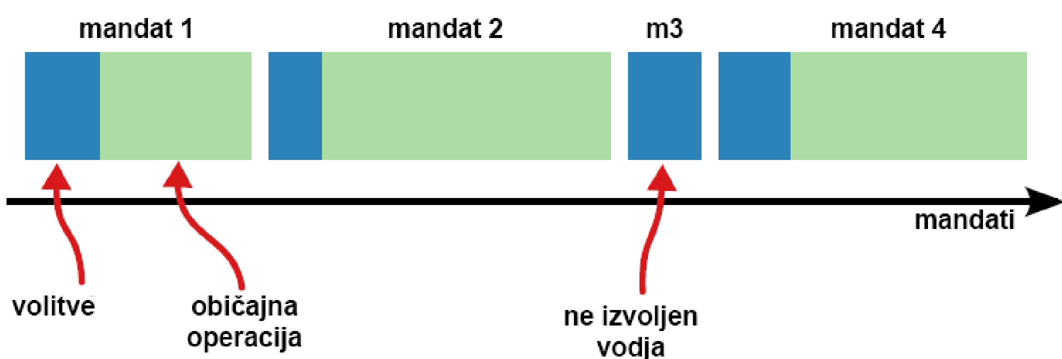
Slika 7: Osnove algoritma Raft (vir: In Search of an Understandable Consensus Algorithm[1]) (v angleškem jeziku)

[1] V viru Osnove algoritma Raft (Search of an Understandable Consensus Algorithm) (v angleškem jeziku) se (nagaja)nahaja poenostavljen prikaz algoritma. Ta prikaz je dobro izhodišče programerju za implementacijo.

6.1 Osnove algoritma »Raft«

»Raft« se običajno izvaja na petih strežnikih. Pet strežnikov pomeni, da odpoved dveh še ne pomeni odpovedi delovanja gruče. Vsak strežnik pozna tri stanja: vodja, sledilec in kandidat. Vodja je strežnik, ki je odgovoren za razpošiljanje dnevnika zapisov, sledilec je pasivni strežnik, ki sprejema ukaze vodje. Status kandidata pa pridobi strežnik, kadar kandidira za vodjo v procesu volitev. V stanju razpošiljanja zahtevka za oddajo volilne kandidature ostalim strežnikom do razgrnitve rezultatov ima strežnik status kandidata. Nato pa, če so bile volitve zanj uspešne, postane vodja. V nasprotnem primeru pa postane sledilec.

»Raft« razdeli časovne intervale na mandate. Mandat je predstavljen kot celo število in se monolitno povečuje z vsako kandidaturo na volitvah. V primeru uspešne izvolitve vodje številka mandata ostane nespremenjena in se uporablja pri izmenjavi dnevnika zapisov ukazov za preverjanje pristnosti zahtevka.



Slika 8: Spreminjanje mandatov (vir: In Search of an Understandable Consensus Algorithm)

6.2 Volitve vodje

Raft uporablja mehanizem srčnega utripa za sprožitev volitve vodje. Ob zagonu gruče vsi strežniki začnejo kot sledilci. Inicializacijo dobijo strežniki preko klica RPC, namenjenega pripenjanju zapisa v dnevnik zapisov (AppendLog). Vodja periodično v razmaku manj kot 150 milisekund kliče omenjeno proceduro na vseh strežnikih, tudi kadar nima zahtevka

odjemalca po upravljanju zapisov. Razlika je le v tem, da kadar vodja nima zapisa, sproži omenjeni klic brez podatkov v argumentih.

Če sledilec v času od 150 do 300 ms (čas med 150 ms in 300 ms se za vsako iteracijo izbere naključno) ne sprejme klica »AppendLog« s strani vodje, sledilec sklepa, da vodja ni izvoljen oz. je nedosegljiv. V tem primeru sam sproži novo zahtevo po volitvi.

Zahteva po volitvi pomeni prvo povečanje številke mandata za 1, nato pa z novo številko mandata razpošlje svojo kandidaturo. V tem trenutku postane sledilec kandidat. Če dobi kandidat pozitivni odgovor za vodjo od večine strežnikov, se stanje kandidata spremeni v vodjo. Kot vodja sebe potrdi tako, da vsem strežnikom pošlje »appendLog« klic z njegovo številko mandata. S tem se volitve zaključijo. Naloge vodje pa postanejo periodično pošiljanje «appendLoga« vsem prisotnim strežnikom v gruči in čakanje na ukaze odjemalcev.

V primeru, da kandidat ne prejme potrditve večine, se volitve razveljavijo. To pomeni, da se po pretečenem času izvede nov cikel za kandidaturo novega vodje.

6.3 Sprejemanje ukazov odjemalcev

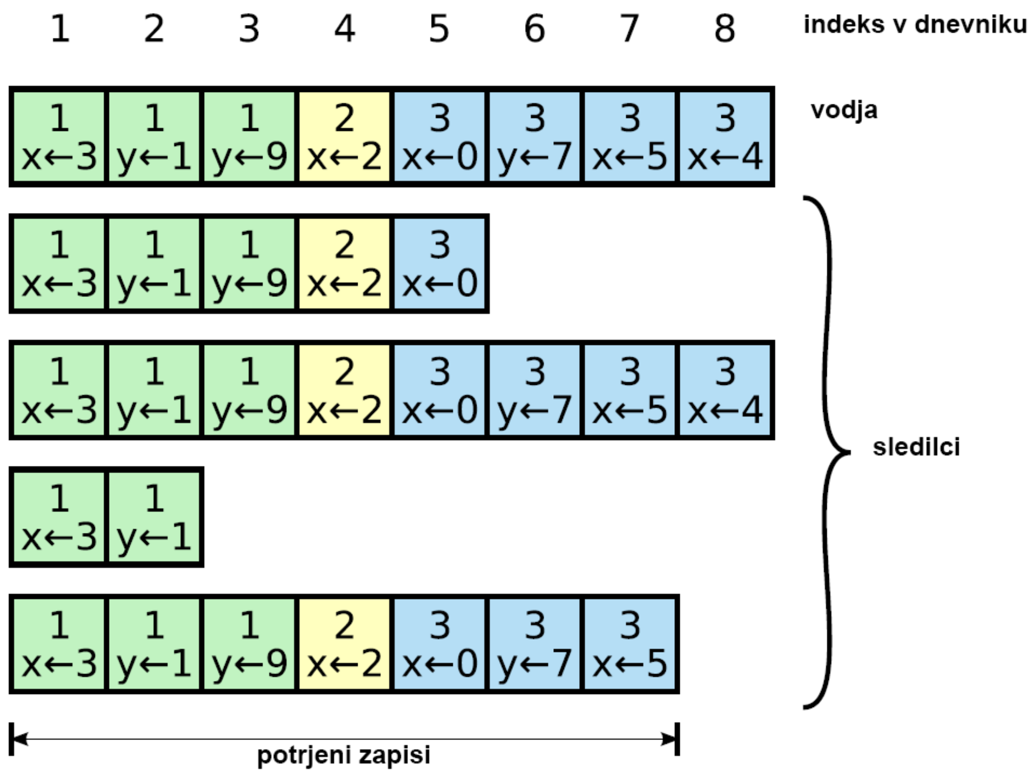
Vse ukaze odjemalca sprejema vodja. Ko vodja dobi nov ukaz, se ta novi ukaz zabeleži z zaporedno številko ukaza v seznam ukazov. Zadnji ukaz, z indeksom in zaporedno številko mandata, se razpošlje na strežnike sledilcev.

Sledilec potrdi zapis vodji, če se ujemata njegova številka mandata in številka mandata, poslanega v ukazu. Istočasno pa preveri, če je njegova zaporedna številka ukaza +1 enaka številki zapisa v ukazu.

Če je številka zapisa v ukazu manjša, pomeni, da mora ta strežnik pri sebi razveljaviti vse številke ukazov, ki so večje ali enake številki, prispeli v ukazu. In na koncu na pravo mesto zapiše in izvede ukaz strežnika.

Če je številka zapisa v ukazu večja, pa pomeni, da temu strežniku nekaj zapisov manjka. Zato v odgovor vodilnemu strežniku sporoči zaporedno številko njegovega zadnjega ukaza. Na to se vodilni strežnik odzove in mu pošlje razliko ukazov.

Ko vodilni strežnik prejme večino odgovorov pozitivnih, je ukaz dokončno izvršen in to tudi sporoči klientu.



Slika 9: Grafični prikaz zapisov v dnevniku zapisov (vir: In Search of an Understandable Consensus Algorithm)

7 OWIN

OWIN definira standardni vmesnik med .NET spletnim strežnikom in spletno aplikacijo. Cilj vmesnika je ločiti spletni strežnik od aplikacije in spodbuditi razvoj preprostih spletnih modulov za .NET.

Če temeljimo na tej predpostavki, je vmesnik primeren za izgradnjo gradnikov aplikacijske gruče. Z omenjenim orodjem bomo implementirali algoritem Raft ter aplikacijo izravnalca obremenitve (angl. load balancer).

Preden se lotimo eksperimenta, pa preglejmo zgradbo knjižnice OWIN.

7.1 Definicija

Definicija knjižnice OWIN definira strežnik, ki je namenjen sprejemanju in pošiljanju zahtevkov. Strežnik mora vsebovati zgradbo, ki pretvarja zahtevke v zahtevano zgradbo.

Ogrodje (angl. framework) je postavljeno na vrh knjižnice OWIN, njegova vloga je implementirati sloj, da lahko aplikacije lažje obdelujejo zahtevo. Primer takega ogrodja je »WebAPI«.

Spletna aplikacija je zgrajena na vrhu ogrodja. V njej je implementirana celotna programska logika aplikacije.

Vmesna koda (angl. middleware) je sestavljena na principu cevovoda. Postavljena je med strežnikom OWIN in ogrodjem. Njena naloga je preoblikovati oz. kakorkoli ovrednotiti zahtevke in odgovore, ki prihajajo v spletno aplikacijo in iz nje. Pogost primer vmesnega koda je modul za avtorizacijo zahtevkov. V našem primeru aplikacijske gruče pa bo, s pomočjo vmesne kode, realiziran izravnalec obremenitve (angl. load balancer).

7.2 Zgradba

OWIN je sestavljen iz dveh delov: slovarja okolja (angl. environment dictionary) in generičnega funkcijskega delegata (angl. generic Func delegate).

7.2.1 Slovar okolja (angl. environment dictionary)

Slovar okolja je implementiran kot struktura ključ/vrednost. Vsebuje informacije o zahtevku, odgovoru in tudi strežniku, ki gostuje spletno aplikacijo, gostovano v knjižnici OWIN. Vsa komunikacija med aplikacijo in strežnikom poteka prav preko te strukture in to omogoča zaganjanje knjižnice v različnih okoljih.

Tabela 7.1: Vsebina ključev zahtevkov slovarja okolja

Ključ	Opis
owin.RequestBody	Tok (angl. stream) zahtevka telesa.
owin.RequestHeaders	IDictionary<string, string[]> z vsebino glave zahtevkov.
owin.RequestMethod	Metoda zahtevka http (primer GET, POST).
owin.RequestPath	Pot zahtevka, relativna na koreno aplikacije. Metoda zahtevka http (primer. GET, POST).
owin.RequestPathBase	Pot osnovnega zahtevka. Koreno aplikacije.
owin.RequestProtocol	Protokol zahtevka (primer: "HTTP/1.0" ali "HTTP/1.1").
owin.RequestQueryString	Poizvedba zahtevka (angl. queryString).
owin.RequestScheme	Zahtevana shema URI (primer: HTTP ali HTTPS).
owin.RequestId	Opcijsko unikatna identifikacija zahtevka, ki je lahko določena tudi v vmesni kodi (angl. middleware) modified.
owin.RequestUser	Opcijska identiteta uporabnika, ki je prav tako lahko določena opcijsko v vmesni kodi.

Tabela 7.2: Vsebina ključev odgovorov slovarja okolja

Ključ	Opis
owin.ResponseBody	Tok (angl. stream) odgovora telesa.
owin.ResponseHeaders	IDictionary<string, string[]> z vsebino glave zahtevkov.
owin.ResponseStatusCode	Opcijsko celo število statusa odgovora HTTP. Število je definirano v RFC 2616 sekciji 6.1.1. Prevzeta vrednost je 200.
owin.ResponseReasonPhrase	Opcijski niz, ki vsebuje opis statusa. Če opis ni definiran, se za opis statusa vrne prevzeti niz, definiran v RFC 2616 sekciji 6.1.1.
owin.ResponseProtocol	Opcijsko protokol odgovora (primer: "HTTP/1.0" ali "HTTP/1.1").

7.2.2 Generični funkcijski delegat (angl. generic func delegate).

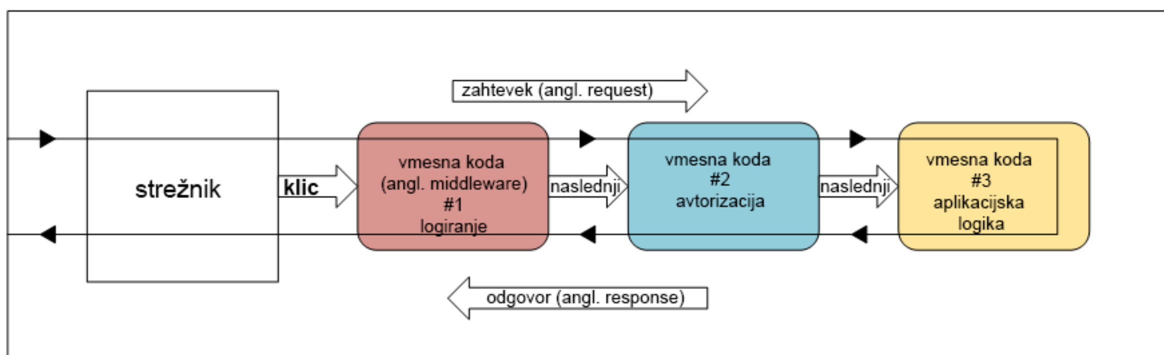
Generični funkcijski delegat vzame kot parameter slovar okolja, vrne pa opravilo (angl. task). Opravilo je posebna struktura, ki za telo vsebuje funkcijo, to izvede, kot rezultat pa vrne izračunano vrednost funkcije.

```
public AppFunc SomeMiddleware(AppFunc next)
{
    // Create and AppFunc using a Lambda expression:
    AppFunc appFunc = async (IDictionary<string, object> environment) =>
    {
        // Do some processing ("inbound")...
        // Invoke and await the next middleware component:
        await next.Invoke(environment);

        // Do additional processing ("outbound")
    };
    return appFunc;
}
```

Slika 10: Primer generičnega funkcijskega delegata (vir: www.codeproject.com)

Posebnost opravila v .NET okolju je zmožnost asinhronega in gnezdenega delovanja, kar daje knjižnici OWIN možnost zaporednega izvajanja več zaporednih opravil. Na tak način se zahtevek, ki prispe v okolje OWIN, izvede preko vseh funkcij, naloženih v aplikaciji. V tem okolju to imenujemo vmesna koda (angl. middleware).



Slika 11: Gnezdenje vmesne kode (vir: www.codeproject.com)

8 IMPLEMENTACIJA ALGORITMA RAFT S KNJIŽNICO OWIN

Implementacijo algoritma Raft smo izvedli v okolju .NET. Komunikacija med strežniki je izvedena z zahtevki HTTP na osnovi »WebApi« klicev. Celoten algoritem Raft je implementiran znotraj ene knjižnice v .NET, kar omogoča enostavno uporabo.

Za potrebe testiranja smo izvedli 5 konzolnih aplikacij, ki uporabljajo narejeno knjižnico. Konzolne aplikacije vsebujejo knjižnico OWIN za samostojno gostovanje. To pomeni, da je vsaka konzolna aplikacija ločen strežnik. Da smo ločili klice odjemalcev in interno komunikacijo gruče, smo metode za komunikacijo z odjemalcem ločili s kreiranjem samostojnega objekta za gostovanje HTTP. Na tak način vsaka konzolna aplikacija posluša na dveh lokacijah.

Testirali smo zaradi enostavnosti in boljšega pregleda nad dogajanjem v posameznem procesu na enem osebnem računalniku. Zato smo posamezne klice razporedili samo po različnih vratih TCP.

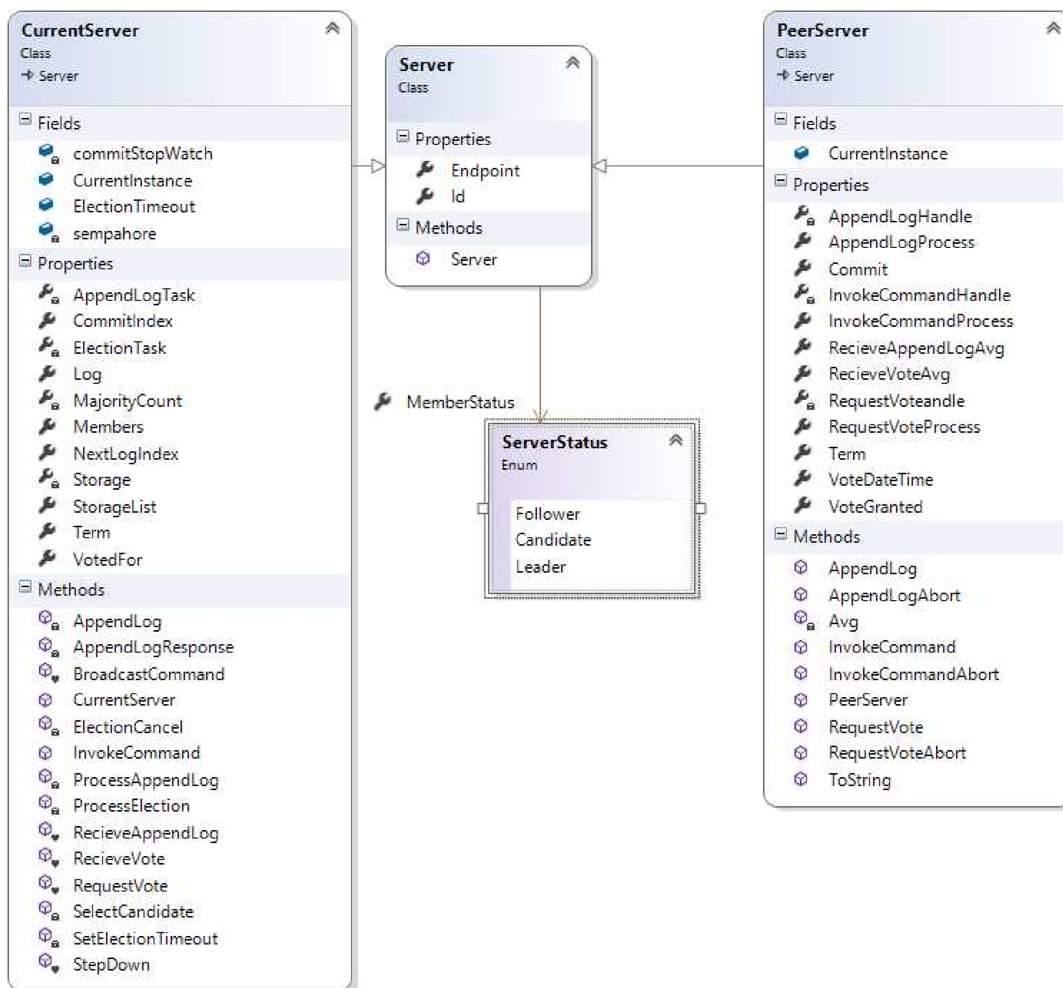
```
static void Main(string[] args)
{
    var members = new List<PeerServer>() {
        new PeerServer("1000", "http://localhost:1000"),
        new PeerServer("2000", "http://localhost:2000"),
        new PeerServer("3000", "http://localhost:3000"),
        new PeerServer("4000", "http://localhost:4000"),
        new PeerServer("5000", "http://localhost:5000")};
    var member = new CurrentServer("1000", "http://localhost:1000", "http://localhost:1001", members);
    Console.ReadLine();
}
```

Slika 12: Primer zagonske metode strežnika

8.1.1 Implementacija strežniške knjižnice z algoritmom Raft

Iz osnovnega razreda *Server* sta izpeljana dva razreda: *CurrentServer* in *PeerServer*. *CurrentServer* predstavlja glavni objekt strežnika. V njem je implementirano opravilo za oddajo zahtevka po volitvi (ElectionTask), opravilo za razpošiljanje ukazov

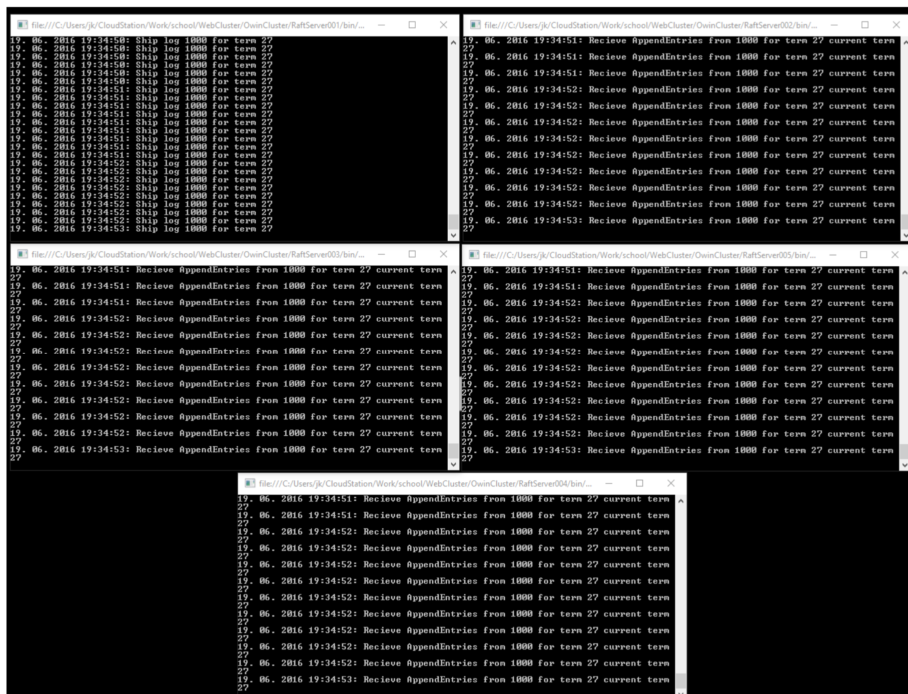
(AppendLogTask). Ta razred vsebuje lastnost shramba (Storage), ki opravlja fizično hranjenje podatkov, shranjenih na strežniku. Pomembna lastnost razreda so tudi člani (Members), ki so na seznamu ostalih strežnikov v gruči in so predstavljeni z razredom PeerServer.



Slika 13: Razredni diagram strežnika Raft

Razred PeerServer ima primarno nalogo razpošiljanja ukazov na posamezni soležni strežnik in prejemanja rezultatov teh zahtevkov.

Vsi zahtevki in njihovi odgovori so izvedeni z razredi, ki se nahajajo v imenskem prostoru RequestResponse.



Slika 14: Prikaz principa delovanja inicializacije vlog strežnikov:prvi strežnik je vodja in pošilja identifikator za mandat 27 ostalim strežnikom.

Slika 15: Slika prikazuje stanje strežnika, dosegljivega preko vmesnika za interno komunikacijo.

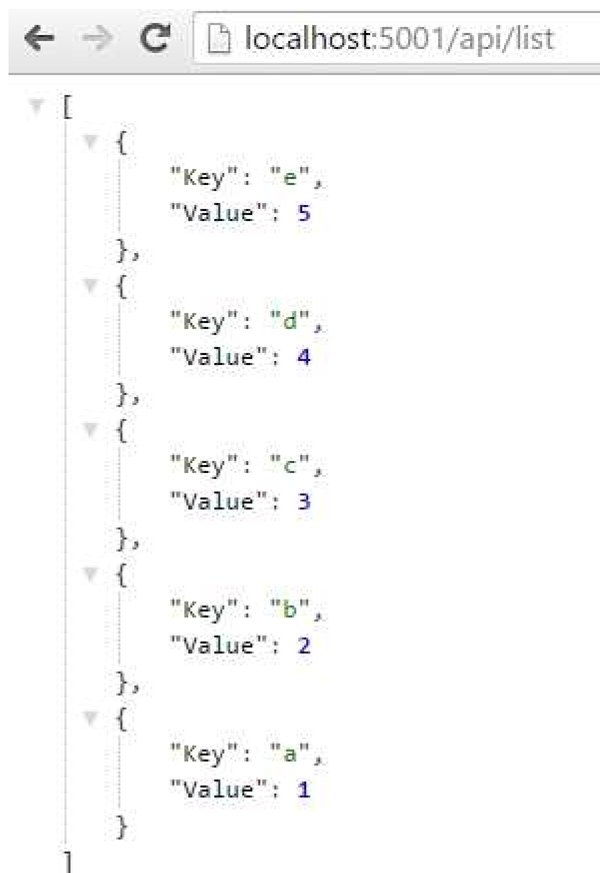
8.1.2 Implementacija komunikacije odjemalca z gručo

Za izvajanje komunikacije odjemalca z gručo skrbi razred *CommandInterpreter*. Razred sprejme ukaz in vrne odjemalcu potrditev ali zavrnitev ukaza. V algoritmu Raft za upravljanje ukazov v celoti skrbi izvoljeni vodilni strežnik. Ker pa uporabnik velikokrat ni seznanjen, kateri strežnik je vodilni oz. ga ta informacija ne zanima, lahko uporabnik izvede ukaz na katerem koli strežniku Raft. Strežnik, ki prejme ukaz, pa ga v primeru, da sam ni vodilni, posreduje pravemu vodji.

Izvedbo odjemalca smo izvedli s funkcijami, izvedenimi z ukazi REST.

Tabela 8.1: Metode, dostopne odjemalcu preko ukazov REST

Url	HTTP metoda	Opis
/api/list/	GET	Metoda vrne seznam vseh zapisov v gruči.
/api/{ključ}	GET	Metoda vrne vrednost ključa, ki ga podamo v argumentu.
/api/{ključ}	PUT	Metoda zapiše vrednost ključa, ki ga podamo v argumentu. Vrednost pošljemo preko telesa zahtevka.
/api/{ključ}	DELETE	Metoda pobriše vrednost ključa, ki ga podamo v argumentu.
/api/status/	GET	Metoda vrne status posameznega strežnika.



Slika 16: Prikaz seznama shranjenih vrednosti

9 IMPLEMENTACIJA IZRAVNALCA OBREMENITVE

Izravnalec obremenitve (angl. loadbalancer) ima pri spletni aplikaciji nalogo sprejemanja zahtevkov. Prejeti zahtevek nato pošlje na najbolj primeren zaledni strežnik, ki je izbran po določenem algoritmu. Prejeti rezultat iz zalednega strežnika pa posreduje nazaj uporabnikom.

V našem primeru smo se lotili izvedbe s knjižnico OWIN, ki deluje kot vmesna koda (angl. middleware). Z razliko, da je vmesna koda že zadnja v cevovodu, kar pomeni, da je rezultat izravnalca tudi rezultat, posredovan gostiteljskemu strežniku.

```
public class LoadBalancerMiddleware
{
    private readonly AppFunc next;
    private readonly LoadBalancerOptions options;

    0 references
    public LoadBalancerMiddleware(AppFunc next, LoadBalancerOptions options)
    {
        this.next = next;
        this.options = options;
    }

    0 references
    public async Task Invoke(IDictionary<string, object> env)
    {
        var loadBalancer = LoadBalancerBase.Create();
        await loadBalancer.Request(env);
        await next.Invoke(env);
    }
```

Slika 17: Implementacija vmesne kode izravnalca

Standardni vmesnik za izravnavo je definiran z vmesnikom *ILoadBalancer*, ki v lastnosti *SelectedServerUrl* določa spletni naslov zalednega strežnika (url), ki mu namerava poslati zahtevek ter *Request* in *Response* opravilo (Task).

Opravilo Request ima v tem primeru nalogo posredovanja zahtevka zalednemu strežniku. Opravilo Response pa sprejema odgovore zalednega strežnika.

Izbira zalednega strežnika poteka izključno znotraj lastnosti *SelectedServerUrl*. V našem eksperimentu smo uporabili poenostavljen algoritem naključnega izbiranja strežnikov na podlagi vnaprej znanega seznama spletnih strežnikov.

10 PRIMERJAVA GRUČE S SAMOSTOJNO APLIKACIJO

10.1.1 Testiranje

Za testiranje spletne gruče smo uporabili preprosto aplikacijo, ki je z zakasnitvijo treh sekund vrnila odgovor. Dolg odgovor smo uporabljali, da smo hitreje dosegli skrajne meje aplikacije. V odgovoru pa smo imeli zapisano ime zalednega strežnika, ki je poslal odgovor.

Slika 18: Odgovor zalednega strežnika

Omenjeno aplikacijo smo najprej testirali samostojno, nato smo testno okolje razširili tako, da smo pred spletni strežnik dodali še izravnalca obremenitve in zaledni strežnik. Najprej smo preverili testiranje z enim zalednim strežnikom. Kasneje pa smo dodajali po en strežnik, dokler nismo imeli gruče petih zalednih strežnikov.

Testiranje je potekalo na enem osebnem računalniku. Pri testu pa smo se osredotočali predvsem na teste obremenitve.

Za testiranje smo uporabili aplikacijo »ab - Apache HTTP server benchmarking tool«. Obremenitev pa smo izvajali na različnem številu hkratnih zahtevkov (št. zahtevkov: 16, 32, 64), vsak poizkus pa je moral izvesti skupaj 128 ali 256 zahtevkov. Rezultate smo primerjali z različnim številom zalednih strežnikov, vsak poizkus pa smo ponovili petkrat.

10.1.2 Rezultati

Rezultati prikazujejo število zahtevkov na sekundo ob posamezni obremenitvi. Večje število pomeni večjo prepustnost zahtevkov.

Tabela 10.1: Samostojni strežnik

	Št. hkratnih zahtevkov		
Poizkus	16	32	64
1	5,30	9,84	12,79
2	5,29	10,54	20,53
3	5,20	10,55	20,57
4	5,30	10,54	20,86
5	5,29	10,56	19,54

Tabela 10.2: En zaledni strežnik

	Št. hkratnih zahtevkov		
Poizkus	16	32	64
1	5,27	10,52	10,65
2	5,26	10,46	15,89
3	5,25	10,40	20,30
4	5,27	10,41	20,30
5	5,27	10,34	20,49

Tabela 10.3: Dva zaledna strežnika

	Št. hkratnih zahtevkov		
Poizkus	16	32	64
1	5,28	8,50	20,63
2	5,28	10,46	9,09
3	5,27	8,10	20,50
4	5,18	10,38	19,44
5	5,10	10,40	12,13

Tabela 10.4: Trije zaledni strežniki

	Št. hkratnih zahtevkov		
Poizkus	16	32	64
1	5,19	6,71	17,97
2	5,17	9,08	20,48
3	5,18	9,11	15,81
4	5,28	10,48	20,36
5	5,10	10,48	20,62

Tabela 10.5: Štirje zaledni strežniki

	Št. hkratnih zahtevkov		
Poizkus	16	32	64
1	5,26	9,38	13,54

2	5,27	8,23	11,66
3	5,18	9,05	18,17
4	5,08	8,35	20,72
5	4,90	9,33	15,14

Tabela 10.6: Pet zalednih strežnikov

	Št. hkratnih zahtevkov		
Poizkus	16	32	64
1	5,18	9,40	11,11
2	4,93	7,80	19,34
3	5,07	10,51	12,10
4	5,19	8,50	8,81
5	5,18	8,78	20,37

10.1.3 Prednosti aplikacijske gruče na podlagi testiranja

S testiranjem smo dokazali, da neke večje razlike o povečanju prepustnosti rešitev aplikacijska gruča ni pokazala. Ni bilo zaznati tudi poslabšanja prepustnosti. Veliko pa se je povečala zanesljivost aplikacije, kar je tudi glavni cilj izvedbe aplikacijske gruče.

11 SKLEP

V diplomskem delu smo pregledali osnovne načine za izgradnjo aplikacijske gruče. Prvo spoznanje je bilo, da aplikacijsko gručo sestavlja veliko med seboj bolj ali manj povezanih elementov. Zato je bilo za izgradnjo potrebno določeno znanje operacijskih sistemov ter komunikacije v računalniški mreži. Pri sami izgradnji smo naleteli na vprašanje, kako zagotavljati ujemljivost podatkov in spreminjajočih informacij o gruči, s tem da ne ogrozimo delovanja gruče od enega samega člena v procesu. Ugotovili smo, da ostali avtorji te probleme rešujejo z algoritmi soglasja. Zato smo se v nadaljevanju poglobili v delovanje algoritma Raft in s pomočjo knjižnice OWIN izdelali implementacijo omenjenega algoritma.

Knjižnico OWIN smo uporabljali tudi kasneje za izgradnjo izravnalca obremenitve. Spletne aplikacije prav s pomočjo izravnalca obremenitve razporejajo promet med zunanjim svetom in aplikacijsko gručo.

Spoznali smo, da je izravnalec obremenitev tehnično relativno preprosta rešitev, vendar pa je ključnega pomena pri prepustnosti celotne gruče. Zato smo opazili na tem področju še prostor za izboljšave.

V diplomskem delu tudi nismo podrobneje obdelali nadzornega modula gruče, ki nam kasneje v produkciji velikokrat olajša upravljanje celotnega sistema. Nadaljevanje upravljanja pa bi bil lahko tudi sistem za avtomatizirano namestitev gruče. Na ta način bi lahko omogočili aplikacijsko gručo, ki bi vire lahko prilagajala sprotni obremenitvi.

12 LITERATURA

[1] Computer cluster, Wikipedia
Dostopno na:
https://en.wikipedia.org/wiki/Computer_cluster [11. 6. 2016]

[2] Clustered Web Hosting
Dostopno na:
http://www.host-stage.net/clusteredwebhosting.php [11. 6. 2016]

[3] Amazon EC2 - Virtual Server Hosting
Dostopno na:
https://aws.amazon.com/ec2/ [11. 6. 2016]

[4] Application Clustering For Scalability and High Availability
Dostopno na:
https://dzone.com/articles/application-clustering [11. 6. 2016]

[5] Application Server Clustering
Dostopno na:
http://www.di.unipi.it/~ghelli/didattica/bdldoc/A97329_03/core.902/a92171/cluster.htm [11. 6. 2016]

[6] J2EE clustering, Part 1
Dostopno na:
http://www.javaworld.com/article/2075019/jndi/j2ee-clustering--part-1.html [11. 6. 2016]

[7] History of computer clusters
Dostopno na:
https://en.wikipedia.org/wiki/History_of_computer_clusters [11. 6. 2016]

[8] Round-robin scheduling
Dostopno na:
https://en.wikipedia.org/wiki/Round-robin_scheduling [12. 6. 2016]

[9] WHAT IS DNS LOAD BALANCING?
Dostopno na:
https://www.nginx.com/resources/glossary/dns-load-balancing/ [18. 6. 2016]

[10] NGINX & Elasticsearch: Better Together
Dostopno na:
https://www.nginx.com/blog/nginx-elasticsearch-better-together/ [19. 6. 2016]

[11] ZooKeeper: Because Coordinating Distributed Systems is a Zoo
Dostopno na:
https://zookeeper.apache.org/doc/trunk/ [19. 6. 2016]

[12] Consensus (computer science)
Dostopno na:
https://en.wikipedia.org/wiki/Consensus_(computer_science) [19. 6. 2016]

[13] The Raft Consensus Algorithm
Dostopno na:
https://raft.github.io/ [19. 6. 2016]

[14] Diego Ongaro and John Ousterhout, In Search of an Understandable Consensus Algorithm (Extended Version): Stanford University
Dostopno na:
https://raft.github.io/raft.pdf [19. 6. 2016]

[15] Diego Ongaro, CONSENSUS: BRIDGING THEORY AND PRACTICE
: Stanford University
Dostopno na:
http://purl.stanford.edu/qr033xr6097 [19. 6. 2016]

[16] OWIN: Open Web Server Interface for .NET
Dostopno na:
http://owin.org/html/spec/owin-1.0.1.html [19. 6. 2016]

[17] Open Web Interface for .NET
Dostopno na:
https://en.wikipedia.org/wiki/Open_Web_Interface_for_.NET [19. 6. 2016]

[18] ASP.NET: Understanding OWIN, Katana, and the Middleware Pipeline
Dostopno na:
http://www.codeproject.com/Articles/864725/ASP-NET-Understanding-OWIN-Katana-and-the-Middlewa [19. 6. 2016]

[19] ab - Apache HTTP server benchmarking tool
Dostopno na:
https://httpd.apache.org/docs/2.4/programs/ab.html [19. 6. 2016]