UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Boštjan Merčun

POVEČANJE ZANESLJIVOSTI IN ZMOGLJIVOSTI SISTEMA ZA PRODAJO VSTOPNIC

DIPLOMSKO DELO
NA VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

Mentor: doc. dr. Mojca Ciglarič

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Preučite področje zagotavljanja zanesljivosti, visoke razpoložljivosti in visoke zmogljivosti strežniških sistemov. Teoretične ugotovitve prenesite v prakso na primeru sistema za spletno prodajo vstopnic. Analizirajte slabosti obstoječega sistema in predlagajte rešitve za izboljšanje performančnih parametrov in dostopnosti. Predlagajte ustrezno stopnjo podvojenosti sistemskih komponent in mehanizme usklajevanja med njimi. Ovrednotite pričakovano število sočasnih uporabnikov, ki jih bo sistem lahko podpiral. Predlagano arhitekturo realizirajte in v praksi ocenite, ali izpolnjuje pričakovanja.

Zahvala

Zahvaljujem se podjetju Dhimahi, d. o. o., ker mi je omogočilo izdelavo diplomskega dela, mentorici doc. dr. Mojci Ciglarič za konstruktivno pomoč in staršem za potrpežljivost.

Kazalo

Seznam uporabljenih kratic in simbolov

- **LTS** (angl. Long Term Support) - distribucija Ubuntu s petletno podporo.
- **LVM** (angl. Logical Volume Management) - sistem za kreiranje prilagodljivih diskovih particij.
- **SSL** (angl. Secure Sockets Layer) - protokol za šifriranje povezav preko spleta.
- **TLS** (angl. Transport Layer Security) - protokol za šifriranje povezav preko spleta. Novejši in boljši kot SSL.
- **MD5** (angl. Message-Digest 5) - zastarela zgoščevalna funkcija, ki je nihče več ne bi smel uporabljati.
- **SHA** (angl. Secure Hash Algorithm) - novejša zgoščevalna funkcija, ki je nadomestila MD5.
- **PHP** (angl. PHP: Hypertext Preprocessor) - programski jezik za programiranje spletnih strani.
- **NFS** (angl. Network File System) - omrežni datotečni sistem
- **CA** (angl. Certificate Authority) - overovitelj SSL certifikatov.
- **PEM** (angl. Privacy Enhanced Mail) - format zapisa SSL certifikatov.
- **IPS** (angl. Intrusion Prevention Systems) - sistem za preprečevanje vdorov.
- **SSH** (angl. Secure Shell) - šifrirana terminalska povezava.

- **DNS** (angl. Domain Name System) - mehanizem za pretvarjanje domen v IP naslove in obratno.
- **RAID** (angl. Redundant Array of Independent Disks) - način povezovanja poceni trdih diskov v redundančna polja.
- **UPS** (angl. Uninterruptible Power Supply) - naprava za zagotavljanje brezprekinitvenega napajanja.
- **NIC** (angl. Network Interface Controller) - omrežna kartica.
- **MAC** (angl. Media Access Control) - omrežni naslov na Ethernet nivoju.
- **OSI** (angl. ISO Open Systems Interconnection) - model, ki standardizira funkcijo omrežnih protokolov na različnih nivojih.
- **VRRP** (angl. Virtual Router Redundancy Protocol) - protokol za avtomatično dodeljevanje omrežnih prehodov.
- **NAT** (angl. Network Address Translation) - način za preslikavo celotnega omrežja v en IP naslov.
- **NAS** (angl. Network-Attached Storage) - omrežni diskovni sistem, ki prenaša podatke na nivoju datoteke.
- **SMB** (angl. Server Message Block) - Microsoftov protokol za deljenje datotek preko omrežja.
- **CIFS** (angl. Common Internet File System) - ena od verzij SMB protokola.
- **WLC** (angl. Weighted Least-Connection) - način razvrščanja povezav pri katerem dobi vsak strežnik enako število povezav upoštevajoč njegovo utež.
- **DRBD** (angl. Distributed Replicated Block Device) - blokovna naprava za prenašanje podatkov na drug strežnik.
- **AES256** (angl. Advanced Encryption Standard) - simetrični šifrirni algoritem z 256-bitno dolžino ključa.
- **XML** (angl. Extensible Markup Language) - razširljivi označevalni jezik.

- **STONITH** (angl. Shoot The Other Node In The Head) - mehanizem za onemogočanje nepravilno delujočih strežnikov v grozdu.
- **SCSI** (angl. Small Computer System Interface) - standard za priklop trdih diskov v strežnik.
- **SAS** (angl. Serial Attached SCSI) - zaporedni način priklopa SCSI diskov.
- **SATA** (angl. Serial ATA) - standard za priklop trdih diskov v strežnik.
- **iSCSI** (angl. Internet Small Computer Systems Interface) - način za priklop podatkovnih naprav preko omrežja.
- **ARP** (angl. Address Resolution Protocol) - način za pretvarjanje IP v naslove MAC.

Povzetek

V času, ko spletne storitve uporabljamo na večini področij v življenju, je zelo pomembno, da so te storitve uporabnikom vedno dosegljive. Visoko razpoložljive storitve so nas, uporabnike, razvadile do te mere, da pričakujemo delovanje vseh storitev v vsakem trenutku, zaradi česar se je znižala tudi naša toleranca do njihovih nedosegljivosti. Vsak izpad delovanja lahko za podjetje pomeni izgubo strank in posledično dobička.

V diplomskem delu bomo prikazali primer arhitekture za prodajo vstopnic preko spleta. Najprej bomo pogledali definicijo visoke razpoložljivosti, zakaj je pomembna in kako jo dosežemo. Nato bomo prikazali osnovne komponente, ki jih taka arhitektura potrebuje, definirali slabosti ter analizirali zanesljivost komponent. V nadaljevanju bomo ugotovljene slabosti odpravili in naredili arhitekturo visoko razpoložljivo ter s tem preprečili daljše izpade aplikacije in posledično izgubo strank.

Visoko razpoložljivo arhitekturo bomo prilagodili še veliki količini sočasnih uporabnikov. Pri prodaji večjih dogodkov, ki se začnejo ob določeni uri, je uporabnikov ponavadi več, kot jih sistem zmore obvladati. Da nam zaradi obremenitve aplikacija ne bi odpovedala, bomo arhitekturi dodali čakalno vrsto, ki bo poskrbela, da bodo strežniki stregli le toliko uporabnikom, kot jih zmorejo obvladati.

Na koncu bomo ovrednotili končno arhitekturo in pogledali, kako smo testirali in izmerili izboljšave.

Ključne besede:

splet, prodaja vstopnic, zanesljivost, razpoložljivost, čakalna vrsta

Abstract

At a time when Internet services are being used in most areas of life, it is very important that these services are always available. Users have become accustomed to highly available services to such an extent that they expect internet services to be available at any time, resulting in no tolerance of downtime. Each downtime can mean loss of customers and therefore of profit.

In this thesis we will show an example of an architecture needed for selling tickets online. First we will look at the definition of high availability and explain why it is important and how we can achieve it. Then we will show the basic components of such architecture, define their drawbacks and analyze the reliability of individual components which we will later improve in order to prevent longer downtimes and consequently loss of customers.

Highly available architecture will be further improved to handle high number of parallel users. When events are sold at a certain time, more users can visit the site than the system can handle. In order to avoid crashing the system, we will add a waiting room to the architecture which will ensure that web servers only get as many users as they are able to serve.

At the end, we will evaluate the final architecture and evaluate how we tested and measured the improvements.

Keywords:

internet, ticket shop, reliability, availability, waiting room

Poglavje 1

Uvod

1.1 Razpoložljivost

Definicija razpoložljivosti je dosegljivost tiste storitve, zaradi katere uporabniki pridejo na našo spletno stran. Sistem ali aplikacija sta razpoložljiva, če lahko uporabniki opravijo tisto, kar so želeli opraviti. V našem primeru je to nakup vstopnice. V kolikor je ne morejo kupiti, se aplikacija šteje za nerazpoložljivo. To hkrati pomeni, da se aplikacija, ki deluje prepočasi, da bi uporabnik dokončal svoj namen, prav tako šteje za nerazpoložljivo.

Za namen merjenja razpoložljivosti spletnih aplikacij se je uveljavil princip *devetk*, ki meri odstotek razpoložljivosti sistema v določeni časovni enoti. Večji, kot je odstotek, bolj je sistem razpoložljiv. 99,9% pomeni, da bo sistem deloval 99,9 odstotkov časa. Enostavna formula za izračun razpoložljivosti je naslednja:

$$A = \frac{MTBF}{MTBF + MTTR} \tag{1.1}$$

A predstavlja odstotek razpoložljivosti, MTBF povprečen čas med napakami in MTTR najdaljši čas za odpravo napake. Vidimo, da se odstotek razpoložljivosti bliža 100 odstotkom, ko se povprečen čas za odpravo napak bliža 0. Poleg tega ima najdaljši čas za odpravo napak manjši vpliv na odstotek razpoložljivosti, ko se povprečen čas med napakami povečuje[1].

Ni potrebno poudarjati, da je implementacija visoke razpoložljivosti izredno draga, zaradi česar se za izjemno visoko razpoložljivost odločajo samo velika podjetja, ki si ta strošek lahko privoščijo. Strošek implementacije raste eksponentno glede na povečanje odstotka razpoložljivosti. Najbolj razpoložljivi sistemi sodijo v nivo *šestih devetk* (99,9999%), njihov strošek v evrih je

Odstotek	Letni izpad	Tedenski izpad
99%	3,65 dni	1 ura in 41 minut
99,8%	17 ur in 30 minut	20 minut in 10 sekund
99,9%	8 ur in 54 minut	10 minut in 5 sekund
99,99%	52,5 minut	1 minuta
99,999%	5,25 minut	6 sekund
99,9999%	31,5 sekund	0,6 sekund

Tabela 1.1: Tabela prikazuje največjo dovoljeno količino izpadov na časovno enoto pri dani razpoložljivosti.

pa še kakšno mesto daljši od števila devetk.

Ob tem moramo omeniti razliko med zanesljivostjo in razpoložljivostjo. Izraz zanesljivost se uporablja za posamezno napravo sistema, medtem ko se razpoložljivost uporablja za cel sistem oziroma za glavno storitev. To pomeni za tisto storitev, za katero je bil sistem primarno postavljen. Zanesljivost posameznih naprav vpliva na celotno razpoložljivost, vendar izpad ene naprave ne pomeni nujno izpada sistema. Redundanca manj zanesljivih naprav je lahko še vedno visoko razpoložljiva. Vsak sistem je sestavljen iz verige logičnih komponent, ki morajo delovati, da deluje tudi aplikacija. Kot komponento si lahko predstavljamo spletni strežnik, ki je lahko samo en, lahko pa postavljen kot grozd, kot bomo opisali v nadaljevanju dela. To pomeni, da lahko posamezna naprava znotraj grozda odpove, vendar bo komponenta še vedno delovala. Primer komponente je lahko tudi omrežje, ki je sestavljeno iz redundančno povezanih omrežnih stikal. V kolikor odpove eno stikalo, komponenta še deluje, če pa jih odpove več, odpove cela komponenta. Komponente niso nujno le strojne, za neprekinjeno delovanje aplikacije morata delovati tudi programsko okolje za aplikacijo (spletni strežnik in podpora programskemu jeziku, v katerem je aplikacija napisana) in sama aplikacija. Pri ovrednotenju razpoložljivosti celotnega sistema je potrebno upoštevati tudi napake v spletnem strežniku ali izvorni kodi aplikacije.

Komponente, ki so potrebne za delovanje naše aplikacije, so med drugim:

- električno napajanje;
- lokalno omrežje;
- spletna povezava;
- spletni strežnik;

- podatkovni strežnik;
- datotečni strežnik;
- okolje za spletno aplikacijo;
- spletna aplikacija;
- vzdrževanje (nastavljanje, posodabljanje, nadgrajevanje ...).

Vsaka od teh komponent pomembno vpliva na razpoložljivost celotne aplikacije, zato je za oceno stopnje razpoložljivosti potrebno analizirati vsako od njih. Ker je razpoložljivost celotnega sistema zmnožek razpoložljivosti posameznih komponent, nobena od njih ne sme predstavljati šibkega člena v verigi in zelo odstopati od želenega odstotka razpoložljivosti celotnega sistema. Poudariti je potrebno tudi zadnjo točko, saj raziskave kažejo, da je človeška napaka med bolj pogostimi razlogi za izpad delovanja sistemov oziroma aplikacij[1].

1.2 Osnovni opis sistema in dobrih praks pri namestitvi strežnikov

Namen diplomskega dela je postavitev sistema za prodajo vstopnic na odprtokodnih rešitvah, ki bi se po zmožnostih lahko primerjal s komercialnimi rešitvami. V tem poglavju bomo predstavili tri obvezne komponente osnovne arhitekture in jih nastavili tako, da bomo dobili osnovno arhitekturo, ki bo lahko gostila aplikacijo za prodajo vstopnic. Ko bo osnovna arhitektura postavljena, bomo v tretjem poglavju pogledali, na kakšen način lahko osnovni arhitekturi izboljšamo razpoložljivost. V četrtem poglavju bomo arhitekturo še dodatno izboljšali tako, da bo zmožna obvladovati veliko število uporabnikov hkrati.

Omejili se bomo predvsem na sistemsko administracijo take arhitekture. Razvoj in vzdrževanje izvorne kode je domena razvijalcev, kar v podjetju ni moje delo in zato marsikatere podrobnosti aplikacije ne poznam. Za potrebe diplomskega dela je bilo moje delo večinoma neodvisno od njenega delovanja.

V diplomskem delu bomo sistem za prodajo vstopnic nameščali kronološko; najprej osnovne komponente, ki jih bomo kasneje izboljšali. Kljub temu, da v času pisanja produkcijsko okolje že deluje, bo v diplomskem delu sistem nameščen od začetka. Zaradi tega bomo namestitev sistema za potrebe dela opisovali v prihodnjiku, na produkcijski sistem se bomo sklicevali pa v preteklem času. Znanje, izkušnje in zahteve za v tem delu opisan sistem, temeljijo

na produkcijskem okolju, ki smo ga v podjetju Dhimahi, d. o. o., namestili za naročnika, zaradi tega bomo uporabili iste tehnologije in verzije protokolov tudi za potrebe diplomskega dela.

V delu bomo izraz *sistem* praviloma uporabljali za celoten sistem, ki bo vključeval strežnike, omrežno opremo in aplikacijo. Izraz *arhitektura* bomo uporabili, ko bomo želeli poudariti način povezave strežnikov ali uporabo protokolov za okolje, na katerem bo tekla aplikacija. Kot primer omenimo arhitekturo, pri kateri je datotečni strežnik ločen od spletnih. Ta podrobnost je pomembna s stališča omogočanja, da nam omogoča postavitev grozda spletnih strežnikov, medtem ko nam arhitektura, kjer so statične datoteke na spletnem strežniku, tega ne omogoča. Izraz *aplikacija* bomo v delu uporabljali za programsko opremo, ki jo bodo uporabljali uporabniki. Izraza *osnovni sistem* ali *osnovna arhitektura* bomo uporabljali za sistem, ki je zadosten za prodajo vstopnic (torej vsebuje vse osnovne komponente), ni pa visoko razpoložljiv ali prilagojen veliki količini uporabnikov. Izraz *začetna arhitektura* bomo uporabljali za arhitekturo, ki je predstavljala začetno stanje arhitekture pri postavitvi produkcijskega okolja. Izboljšave začetne arhitekture so vsebina mojega diplomskega dela, rezultatu bomo rekli pa *končna arhitektura*. Če v kateri podrobnosti ne bo popolnoma identična *produkcijski*, bomo po potrebi uporabili tudi slednji izraz. Izraz *strežnik* bo lahko pomenil strojno, včasih pa programsko opremo. Kot primer navedimo *spletni strežnik*, ki bi v prvem primeru pomenil fizični strežnik, na katerem teče programska oprema, v drugem pa strežnik Apache[3], ki na tem fizičnem strežniku teče. Ko bo govora o arhitekturi, se bo v večini primerov nanašal na strojno opremo, v kontekstu aplikacije pa na programski strežnik.

Naša **začetna arhitektura** je vsebovala en datotečni in en nadomestni strežnik, na katerega so se enkrat na noč zrcalili konfiguracija in podatki. Oba sta imela nameščeno isto programsko opremo. V primeru odpovedi osnovnega strežnika bi moral skrbnik na nadomestni strežnik namestiti zadnjo konfiguracijo ter prekonfigurirati omrežje tako, da bi se slednji predstavljal kot osnovni strežnik.

Tudi podatkovni strežniki niso bili redundančni, vsi so imeli en (isti) nadomestni strežnik, ki bi ga uporabili v primeru odpovedi kateregakoli osnovnega strežnika. Baze so se replicirale na rezervni strežnik vsakih osem ur. V primeru odpovedi kateregakoli podatkovnega strežnika, bi moral skrbnik restavrirati ustrezne baze ter prenastaviti omrežne nastavitve rezervnega strežnika, tako da bi se predstavljal kot osnovni.

Spletni strežniki so bili že nastavljeni kot grozd (angl. cluster). Za nadzor nad grozdom smo uporabili program *Piranha*[2], ki je med prekonfiguriranjem

velikokrat odpovedal, kar je povzročilo mnogo nepotrebnih kratkih izpadov ob vzdrževalnih delih. Zaradi tega smo iskali boljšo rešitev.

Glavna **pomanjkljivost začetne arhitekture** je bila dolžina izpada aplikacije v primeru odpovedi posamezne komponente. Za začetek bi bilo potrebno vsako odpoved rešiti ročno. Že čas do začetka odpravljanja napake je lahko v takem primeru dolg, če skrbnik ni takoj dosegljiv. Pri odpravi izpada je obstajala tudi velika možnost človeške napake in v nesrečnem primeru bi lahko z napačno konfiguracijo omrežja onemogočili še nadomestni strežnik. Nezanemarljiva je tudi izguba podatkov. V povprečju bi pri datotečnem strežniku izgubili za pol dneva podatkov, pri podatkovnem pa štiri ure. To je tolikšna količina podatkov, da bi jo uporabniki že zaznali. Kot skrbnika me je pa omenjeno stanje motilo predvsem zato, ker nobena moja odsotnost ni bila brezskrbna.

Razlogov za izboljšanje razpoložljivosti sistema je bilo dovolj, s povečano razpoložljivostjo pa ne bi zgolj izboljšali uporabniške izkušnje ter razveselili naročnika, temveč tudi sebi olajšali spanec.

Osnovne komponente sistema za prodajo vstopnic so naslednje:

- **spletni strežnik**, na katerem teče aplikacija;
- **podatkovni strežnik**, ki hrani podatke o uporabnikih in nakupih;
- **datotečni strežnik**, ki hrani statično vsebino za spletni strežnik.

Za zelo majhne trgovine ali namene razvoja so lahko vsi trije na istem fizičnem strežniku, za večje število nakupov ali uporabnikov je pa smiselno vsak strežnik postaviti na svojem fizičnem ali virtualnem strežniku. Na ta način med seboj ne tekmujejo za vire, kar aplikacijo dodatno pohitri. Tudi strežniki so med seboj izolirani, zaradi česar je lažje poskrbeti za varnost. Ker so bile zahteve za naš produkcijski sistem že v začetku visoke, bomo že osnovno arhitekturo postavili na treh ločenih strežnikih.

Ker ničesar ne bi smeli odpreti v splet brez požarnega zidu, bomo tudi požarni zid vključili v osnovno arhitekturo, čeprav neposredno ne pripomore k sami prodaji vstopnic. Prav tako ni potrebe, da bi bili vsi strežniki povezani neposredno v splet, zato jim bomo nastavili zasebne naslove IP. Na požarnem zidu bomo nastavili javne naslove IP, ki jih bomo nato preslikali le do tistih strežnikov, ki bodo morali biti dostopni s spleta. Za našo osnovno arhitekturo bo to spletni strežnik.

Vsak skrbnik ima za namestitev in administracijo operacijskega sistema Linux in ostalih strežnikov svoja pravila, vseeno je pa dobro upoštevati nekaj smernic:

- **Particije** naj bodo smiselno razporejene in velike, da na njih ne bo zmanjkovalo prostora in bodo med sabo izolirane. Na podatkovnem strežniku je recimo pametno bazo hraniti na od sistemskih datotek ločeni particiji. Na ta način ena polna particija ne bo vplivala na ostale. V kolikor velikost particij ni znana vnaprej, je najboljše uporabiti LVM[10] (angl. Logical Volume Management) in particije sproti povečevati.

- Na strežniku naj bo nameščena **samo potrebna programska oprema**. Nepotrebna oprema na strežniku po nepotrebnem porablja prostor in odpira dodatne možnosti za težave. Če so nameščene preko omrežja dostopne storitve, ogrožajo tudi varnost.

- Vse, kar je nameščeno na strežniku, vključno z operacijskim sistemom, naj bo redno **posodobljeno**. Nezakrpane ranljivosti so najpogostejši razlog za vdor!

- Vsak strežnik, tudi če je na notranjem omrežju, naj ima vključen in nastavljen **požarni zid**, skozi katerega je dovoljen le tisti promet, ki mora do strežnika dejansko priti. Nikoli ne vemo, kdaj se bo na strežniku pognalo nekaj, česar nismo predvideli. V primeru vdora lahko požarni zid omeji škodo. Prav tako je priporočljivo uporabiti knjižnico **libwrap**[4], ki se s požarnim zidom lepo dopolnjuje. Knjižnica libwrap za vsako povezavo preveri izvorni naslov in na podlagi pravil, ki jih nastavimo v datotekah */etc/hosts.allow* in */etc/hosts.deny*, povezavo bodisi dovoli bodisi zapre.

- Za avtentikacijo naj bodo uporabljena **močna gesla**. To so gesla, ki vsebujejo velike in male črke, številke in posebne znake ter so dolga vsaj 12 znakov. Morajo biti čimbolj naključna in ne enostavno uganljiva, kot so recimo imena družinskih članov, datum rojstva ali registrska oznaka avtomobila. Za ssh je dobro geslo zamenjati s **ključem**, ki naj bo pri uporabniku prav tako zaščiten z močnim geslom. Za VPN je priporočljivo uporabiti **certifikate**, ki naj uporabljajo sodobne šifrirne algoritme (AES-256) in zgoščevalne funkcije (SHA-2) ter ključe dolge vsaj 2048 bitov, ki naj bodo prav tako zaščiteni z močnimi gesli.

- Kjer je možno, naj se za prenos podatkov uporablja **šifriranje**. Šifrirni algoritmi naj bodo sodobni in močni, ključi pa dovolj dolgi. Za asimetrično šifriranje, kot je značilno za SSL in TLS, naj bodo ključi dolgi vsaj 2048 bitov. V našem primeru bomo šifriranje uporabljali za varovanje košarice uporabnikov. To naredimo s protokolom https. Naj omenim, da

so verzije protokola SSL zastarele in ranljive ter naj bi bile onemogočene. Namesto njih uporabljajmo TLS. Uporaba zgoščevalnih funkcij MD5 in SHA1 zaradi zastarelosti ni več priporočljiva, zato raje uporabljajmo SHA2.

- Za popoln **nadzor nad sistemom** je zelo priporočljiva uporaba programske opreme, ki spremlja stanje strežnikov in o nepredvidenih dogodkih tudi obvešča skrbnika sistema. Zazna lahko deviacije, kot so preveč ali premalo procesov nekega strežnika, nedosegljivost aplikacije na določenih vratih, zasedenost diska ali spomina in čas odziva aplikacije preko omrežja. Z uporabo lastnih skript kot vmesnikov so možnosti nadzora skoraj neomejene. Primer take aplikacije je Nagios[5]. V povezavi z napravo za pošiljanje kratkih sporočil (SMS) je nepogrešljiv del visoko razpoložljivega sistema.

- Ne smemo pozabiti na **varnostne kopije podatkov**. V primeru težav je nujno, da lahko podatke restavriramo. Ustrezno je potrebno načrtovati tudi čas hranjenja varnostnih kopij. Za nekatere podatke je dovolj enkrat na dan, za druge morda na vsakih nekaj ur. Varnostne kopije in arhivski podatki so lahko uporabni tudi pri sledenju spremembam ali za vpogled v preteklost podatkov.

Poglavje 2

Osnovna arhitektura

V tem poglavju si bomo pogledali štiri komponente osnovne arhitekture, ki jih aplikacija za prodajo vstopnic nujno potrebuje. To so spletni, podatkovni in datotečni strežnik ter požarni zid. V tem primeru govorimo o programskih strežnikih, torej programih, ki tečejo na operacijskem sistemu Linux. Kot smo že omenili, bi lahko vse namestili na isti strojni strežnik, vendar bomo zaradi performančnih in varnostnih razlogov vsakega od strežnikov namestili na svojo strojno opremo.

Na produkcijskem okolju smo za operacijski sistem izbrali distribucijo Linux CentOS[6], saj velja z dolgim razvojnim ciklom in kakovostnim testiranjem za stabilno in zanesljivo. Slaba stran dolgega razvojnega cikla je neažurno nadgrajevanje verzij programske opreme, kar ni po godu razvijalcem, ki želijo vedno najnovejše funkcije. Zaradi tega smo distribucijo CentOS na spletnih strežnikih zamenjali za zadnji Ubuntu LTS[7]. V času pisanja je zadnja verzija 14.04. Omenjeni distribuciji bomo uporabili tudi za nastavitev naše osnovne arhitekture.

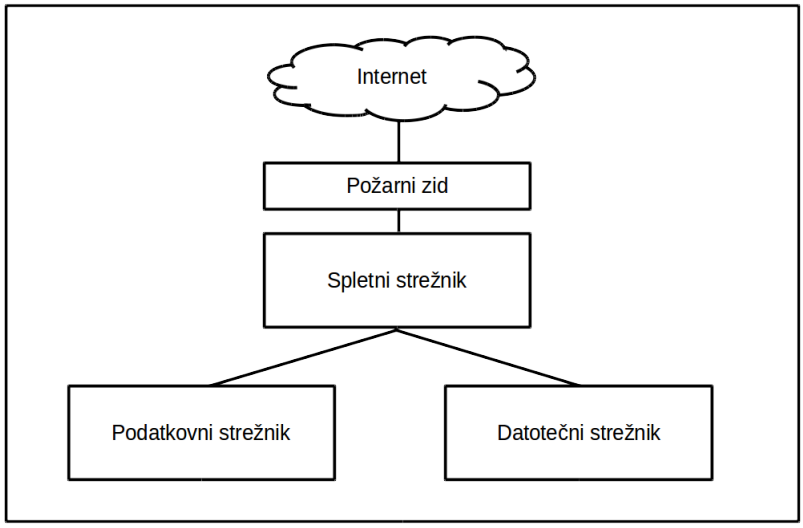
Za potrebe naše arhitekture bomo določili nekaj fiktivnih podatkov, ki jih bomo uporabljali v nadaljevanju in s katerimi bomo nastavili strežnike:

```
omrežne nastavitve:
notranje omrežje: 192.168.1.0/24
spletni strežnik: 192.168.1.10
podatkovni strežnik: 192.168.1.20
datotečni strežnik: 192.168.1.30
zunanje omrežje: 1.1.1.0/24

Datotečni strežnik:
izvožen direktorij: /var/podatki
```

```
Podatkovni strežnik:
ime baze: trgovina
uporabnik za bazo: pg_uporabnik
geslo za bazo: pg_geslo

Spletni strežnik:
domena: www.trgovina.tld
Statični direktorij: /var/www/html/podatki
```



Slika 2.1: Slika prikazuje osnovno arhitekturo za prodajo vstopnic. Zaradi varnosti vsebuje tudi požarni zid, čeprav ta nima vloge pri sami prodaji vstopnic. Ko uporabnik odpre stran, se v resnici poveže na požarni zid. Ta ga preusmeri na spletni strežnik, na katerem se zanj poženejo potrebne PHP skripte, ki lahko za izvršitev uporabnikovih zahtev uporabljajo tudi podatkovni in datotečni strežnik. Ko se zahteva zaključi, se rezultat vrne uporabniku.

2.1 Datotečni strežnik

V tem poglavju bomo namestili datotečni strežnik, ki bo hranil statično vsebino. To je vsebina, ki je ne bo izvajal PHP[8]. Sem sodijo grafike, ki jih bo aplikacija prikazovala, podatki, ki jih bo generiral uporabnik (vstopnice), in vse, kar bo zapisoval spletni strežnik (na primer dnevniške datoteke). Datotečni strežnik sicer ni procesorsko zahteven, vendar ga bomo zaradi kasnejših optimizacij sistema že vnaprej ločili od spletnega strežnika. Če bi imela naša aplikacija zelo malo obiska, datotečnega strežnika ne bi potrebovali, saj bi lahko statično vsebino stregel kar spletni strežnik. Slaba stran tega bi bila, da je strežnik Apache, ki ga bomo uporabljali na spletnem strežniku, procesorsko in spominsko zahteven in bi za veliko količino zahtev porabil veliko virov. Ker bomo datotečni strežnik ločili od spletnega, bomo lahko uporabili bistveno bolj učinkovit strežnik Lighttpd[11]. Za priklop statičnih podatkov na spletni strežnik bomo uporabili protokol NFS[9] (verzijo 3), ker nam bo omogočal uporabo istih uporabniških identitet na vseh strežnikih. Alternativno bi lahko uporabili protokol SMB[12], vendar za priklop na SMB potrebujemo uporabniško ime in geslo, preko katerega potem dostopamo do vseh datotek. NFS omogoča, da imajo podatki na datotečnem strežniku več lastnikov, vsakega s svojimi dovoljenji, pri čemer bodo lastništva in dovoljenja datotek enaka tudi na spletnih strežnikih. Verzijo 3 bomo uporabili zaradi podobnosti s produkcijskim sistemom. Produkcijska aplikacija to že dolgo uporablja, prehod na verzijo 4 pa ni trivialen, zato bomo tudi za naš sistem uporabili verzijo 3.

Strežnik NFS potrebuje za delovanje še nekaj pomožnih podprogramov, ki se morajo pognati ob zagonu sistema:

- **rpcbind**
- **mountd**
- **statd**
- **lockd**

NFS privzeto komunicira preko vrat 2049(TCP in UDP), rpcbind pa preko 111(TCP in UDP). Po potrebi lahko privzeta vrata spremenimo v datoteki v */etc/sysconfig/nfs*. Če to storimo, moramo nato ista vrata odpreti v požarnem zidu. Mi bomo pustili privzete nastavitve.

Na strežnik najprej namestimo NFS strežnik z ukazom *yum install nfs-utils*. V */etc/exports* določimo, komu bomo izvozili kateri direktorij. Zaenkrat je to samo naslednji:

```
/var/podatki 192.168.1.10(rw,sync,root_squash,no_all_squash)
```

Vrstica pove, da bomo podatke v direktoriju /var/podatki izvozili strežniku z naslovom IP 192.168.1.10 (to je naš spletni strežnik). Strežnik bo lahko podatke bral, spreminjal in ustvarjal nove (rw), ki se bodo prenesli na datotečni strežnik takoj, ko bodo na spletnem strežniku spremenjeni (sync). Skrbniški uporabniški račun (root) bo spremenjen v drugo identiteto, tako da na datotečnem strežniku ne bo imel skrbniških privilegijev (root_squash). Ostali računi bodo preslikani na datotečni strežnik z isto identiteto, kot jo bodo imeli na spletnem strežniku (no_all_squash).

2.2 Podatkovni strežnik

Podatkovni strežnik bo hranil vsebino, ki jo bodo uporabljale skripte PHP. Tako kot na produkcijskem sistemu bomo uporabili Postgresql 9.2[13].

Strežnik Postgresql namestimo z ukazom *yum install postgresql-server*. Privzeto komunicira preko vrat 5432 in hrani baze v direktoriju /var/lib/pgsql. Tam je tudi konfiguracija strežnika (/var/lib/pgsql/data/postgresql.conf). Ker ima distribucija CentOS privzeto vključen požarni zid, moramo prometu odpreti vrata 5432. Po namestitvi je potrebnih nekaj nastavitev, preden bomo lahko začeli uporabljati bazo. Najprej inicializiramo baze na disku, naredimo bazo z imenom *trgovina* in uporabniški račun ter geslo za dostop do nje. Za avtentikacijo in nadzor dostopa uporablja Postgresql mehanizem, imenovan Host based access. Nastavitve so v datoteki /var/lib/pgsql/data/pg_hba.conf. Vsaka vrstica vsebuje naslednje stolpce:

- **tip dostopa** - lahko je omrežni ali lokalni;
- **baza** - ime baze, do katere bo imel uporabnik dostop;
- **uporabnik** - uporabnik, ki mu dajemo dostop do baze;
- **naslov** - omrežni naslov (IP) s katerega lahko uporabnik dostopa do baze;
- **metoda** - način avtentikacije.

V skladu z navodili dodamo v datoteko naslednjo vrstico:

```
host trgovina pg_uporabnik 192.168.1.10/32 md5
```

Zgornja vrstica pomeni, da bo lahko uporabnik *pg_uporabnik* dostopal do baze *trgovina*, če se bo povezal preko omrežja z naslova *192.168.1.10* in se avtenticiral z geslom. Ob tem naj omenim, da je md5 edina zgoščevalna funkcija, ki jo strežnik Postgresql podpira. Kljub temu, da smo v prvem poglavju omenili, da je md5 zastarel in ga ne bi smeli več uporabljati, tu nimamo druge možnosti, zaradi česar je dobro skrbniške naloge na bazi opravljati preko šifrirane povezave. Dostop s spletnih strežnikov bo potekal samo preko lokalnega omrežja in zgoščena gesla ne bodo dostopna nikomur, v kolikor nihče ne bo prisluškoval prometu v lokalnem omrežju.

Poleg navedenega uporabniškega računa je dobro ustvariti še enega za skrbniška opravila, za katerega bo potrebno dodati novo vrstico v *pg_hba.conf* datoteko. Uporabniku *postgres*, ki ima neomejen dostop do vseh baz, je potrebno nastaviti dobro geslo.

2.3 Spletni strežnik

Spletni strežnik bo glavna komponenta naše aplikacije za prodajo vstopnic. Tu se bo izvajala vsa koda, ki bo poskrbela, da bo uporabnik lahko uspešno rezerviral in kupil svojo vstopnico. Za nakupni proces bodo skripte PHP po potrebi uporabljale tudi podatkovni in datotečni strežnik ter poskrbele, da bo nakup tudi ustrezno dokumentiran v bazi in sistemskih dnevniških datotekah. Apache bomo izbrali zato, ker je najbolj znan spletni strežnik s podporo za PHP. PHP podpira že zelo dolgo in je zaradi tega preverjeno stabilen. Podpira tudi veliko drugih možnosti, kot so nadzor dostopa, avtentikacija, preusmerjanje in prepisovanje spletnih naslovov.

Namesto strežnika Apache bi lahko uporabili hitrejši in učinkovitejši spletni strežnik, kot sta strežnika Lighttpd[11] ali Nginx[16] in PHP-FPM[17]. Strežnika Lighttpd in Nginx sta učinkovitejša predvsem zato, ker podpirata manj funkcij kot strežnik Apache in morata posledično pri vsakem novem procesu v spomin naložiti manj kode ter ne zaganjata novega procesa za vsako povezavo. Velik del te kode predstavlja podpora za PHP. Strežnik Apache za vsak zagnan proces naloži v spomin tudi podporo za PHP, ki se potem izvaja znotraj tega procesa. PHP-FPM deluje pa kot samostojen proces in je ločen od spletnega strežnika. Strežnik zato zahteve PHP pošilja procesu preko lokalne vtičnice (angl. socket) ali omrežja. Ko se skripta PHP konča, se rezultat preko iste povezave vrne spletnemu strežniku. Prednost takega pristopa je manjša poraba pomnilnika in hitrejše izvajanje skript PHP. Žal pa PHP-FPM ne ponuja čisto identičnega okolja kot Apache, zato bi bilo potrebno produkcijsko aplikacijo prilagoditi. Ker tega nismo uspeli storiti, na produkcijskem okolju ostajamo na strežniku Apache.

Za spletni strežnik bomo uporabili Apache 2.4 s podporo za PHP in vse module, ki jih aplikacija potrebuje. Med njimi bo vsaj podpora za bazo. Spletni strežnik bo prikazoval nešifrirane (http) in šifrirane (https) strani. Za šifrirane strani bomo potrebovali certifikate, ki naj bodo za resno aplikacijo overjeni od enega od znanih overoviteljev[14] (angl. Certificate Authority). Overovitelji so podjetja, ki izdajajo spletne certifikate. Ob izdaji preverijo, če je naročnik zares oseba ali podjetje, za katerega bo certifikat izdan. Tak certifikat podpišejo in jamčijo za vsebino certifikata. Ko se brskalnik poveže na šifrirano stran in dobi certifikat, lahko preko overovitelja preveri, če je certifikat pristen.

Na spletnem strežniku bomo namestili strežnik Apache (*apt-get install apache2*) in podporo za PHP (*apt-get install libapache2-mod-php5*). Odvisnosti v paketih bi morale namestiti še preostale potrebne pakete. Spletni strežnik privzeto komunicira preko vrat 80/tcp (http) in 443/tcp (https). Stran prikazuje

iz direktorija /var/www/html. Ker trenutno ni potrebe po spremembi privzetih vrednosti, bomo po namestitvi strežnika stran naložili v /var/www/html.

Najprej bomo v /etc/apache2/ssl odložili zasebni ključ in certifikat za trgovino ter oba poimenovali www.trgovina.tld. Vsak bo imel svojo končnico; zasebni ključ ima po navadi končnico *key*, certifikat pa *crt*. Konfiguracijo za našo spletno stran bomo zapisali v datoteko www.trgovina.tld.conf v direktoriju /etc/apache2/sites-available:

```
<VirtualHost 192.168.1.10:80>
    DocumentRoot /var/www/html
    ServerName www.trgovina.tld
    CustomLog "/var/log/apache2/trgovina_access_log" combined
    ErrorLog "/var/log/apache2/trgovina_error.log
</VirtualHost>
<VirtualHost 192.168.1.10:443>
    DocumentRoot /var/www/html
    ServerName www.trgovina.tld
    CustomLog "/var/log/apache2/trgovina_access_log" combined
    ErrorLog "/var/log/apache2/trgovina_error.log
    SSLEngine on
    SSLProtocol all -SSLv3
    SSLCertificateFile /etc/apache2/ssl/www.trgovina.tld.crt
    SSLCertificateKeyFile /etc/apache2/ssl/www.trgovina.tld.key
</VirtualHost>
```

Po nastavitvi strani je ne smemo pozabiti omogočiti, pa tako tudi modula za SSL.

Direktorija za statično vsebino pri osnovni arhitekturi sicer ne potrebujemo, vendar ga bomo nastavili za kasneje. Najprej preverimo, če imamo dovoljenje za priklop s strani datotečnega strežnika. Z ukazom *showmount -e 192.168.1.30* lahko vidimo, kateri direktoriji so nam na razpolago:

```
/var/podatki 192.168.1.10
```

To pomeni, da lahko direktorij pripnemo na spletni strežnik, kar naredimo tako, da v datoteko */etc/fstab* dodamo naslednjo vrstico:

```
192.168.1.30:/var/podatki /var/www/html/podatki nfs \
rw,nfsvers=3,auto 0 0
```

Vrstica pove, da bomo z datotečnega strežnika *192.168.1.30* pripeli vsebino direktorija */var/podatki*. Podatki bodo dostopni v */var/www/html/podatki*. Določimo tudi, da gre za verzijo 3 protokola NFS in da bomo podatke lahko tudi spreminjali (rw). Podatki naj se pripnejo avtomatično ob zagonu strežnika (auto). Zadnji dve vrednosti določata prioritete pri arhiviranju in preverjanju podatkov na particiji, vendar za našo aplikacijo nista pomembni.

Karkoli bomo zapisali v /var/www/html/podatki, se bo v resnici shranilo preko omrežja na datotečni strežnik v /var/podatki. Zapisovanje bo sicer počasnejše, vendar nam bo v nadaljevanju omogočalo postavitev več spletnih strežnikov z isto vsebino. V kombinaciji s predpomnjenjem podatkov na spletnih strežnikih bo dostop za branje skoraj enako hiter, kot če bi bili podatki shranjeni lokalno na spletnih strežnikih.

2.4 Požarni zid

Požarni zid sicer v samem sistemu za prodajo vstopnic nima funkcije, vendar brez požarnega zidu ne bi smeli ničesar izpostavljati spletu. Za našo produkcijsko aplikacijo uporabljamo napravo Cisco ASA[20] (v nadaljevanju *požarni zid*, saj je to njegova glavna funkcija), ki ima v našem primeru tri funkcije:

- blokiranje neželenega prometa s spleta;
- zaznavanje in preprečevanje vdorov - IPS (angl. intrusion prevention system);
- preslikava zunanjih IP naslovov v notranje.

2.4.1 Blokiranje neželenega prometa

Blokiranje neželenega prometa je doseženo tako, da je skozi požarni zid spuščen izključno tisti promet, ki ga potrebujemo. V našem primeru sta to le http in https. Za administracijo se omogoči še varno lupino SSH[18], ostalo pa lahko ostane zaprto.

2.4.2 Zaznavanje in preprečevanje vdorov

Za razliko od blokiranja deluje zaznavanje in preprečevanje vdorov na prometu, ki je spuščen skozi požarni zid. Če nek protokol ali vrata potrebujemo, jih ne moremo kar blokirati. To pomeni, da do notranjih strežnikov spustimo tudi

vse varnostne grožnje in poizkuse vdora, ki so del tega prometa. Zaznavanje in preprečevanje vdorov deluje na dva načina. Bazo podatkov ranljivosti, ki jo vzdržuje Cisco, naprava redno prenaša k sebi in primerja zahteve z opisi groženj. Če se opis ujema, lahko zahtevo ustavi ali pa zgolj obvesti skrbnika omrežja. Drug način je hevristično iskanje groženj, pri katerem se promet analizira in s pomočjo algoritmov išče odstopanja, ki se v normalnem primeru ne bi smela zgoditi. Naprava pozna tudi vrsto protokolov, zato lahko vsakega izlušči iz prometa in interpretira ter tako najde morebitne anomalije.

2.4.3 Preslikava zunanjih naslovov IP v notranje

Zunanje naslove IP preslikamo v notranje zato, da so vsi strežniki v notranjem omrežju in hkrati tudi arhitektura skriti zunanjemu svetu. Vsi uporabniki se povežejo na zunanje IP naslove, ki so nastavljeni na požarnem zidu, prevajalnik omrežnih naslovov[19] (angl. Network Address Translation) pa preusmeri promet na ustrezne notranje naslove.

V našem primeru je potrebno v DNS strežniku za domeno www.trgovina.tld vpisati omrežni naslov 1.1.1.1, za slednjega pa na požarnem zidu nastaviti, da se vrata 80 in 443 preslikajo na naslov 192.168.1.10.

2.5 Zaključek

V tem poglavju smo pokazali nastavitve osnovnih komponent, ki jih potrebujemo za aplikacijo za prodajo vstopnic. S postavljeno arhitekturo (prikazana na sliki 2.1) bomo sicer lahko prodajali vstopice, vendar ima ta svoje slabosti. Največja pomanjkljivost je razpoložljivost, saj vsaka strojna ali programska odpoved pomeni izpad delovanja celotne aplikacije, kar ni sprejemljivo. V naslednjem poglavju bomo zato pogledali, kako lahko poskrbimo za redundanco strežnikov in aplikaciji drastično povečamo razpoložljivost.

Poglavje 3

Zanesljivost sistema in redundanca

V prejšnjem poglavju smo postavili osnovno arhitekturo, ki jo potrebujemo za prodajo vstopnic. Ker je arhitektura izredno ranljiva za strojne in programske odpovedi, bomo v tem poglavju pogledali, kako lahko to slabost izboljšamo. Pri povečevanju redundance in s tem razpoložljivosti sistema si bomo postavili tri cilje:

- vsaka komponenta naj bo že sama po sebi čim bolj zanesljiva;
- vsaka komponenta mora imeti svojo redundanco;
- preklop na redundančno komponento mora biti avtomatičen.

Pri omrežni opremi je naloga enostavna, saj so dražje naprave že same po sebi zelo zanesljive. Razen tega, da kupimo naprave priznanih znamk in požarne zidove povežemo na dve ločeni veji napajanja, pri zanesljivosti posamezne naprave ne moremo storiti ničesar več.

Pri strežnikih je opcij za povečanje zanesljivosti več. Da bi dosegli čim večjo zanesljivost strežnikov, se bomo držali naslednjih pravil:

- **Uporabljali bomo strežnike z dvojnim napajanjem**. Dva napajalnika bosta v prvi vrsti nudila redundanco pri odpovedi enega napajalnika, pri čemer bo strežnik nemoteno deloval naprej, napajanje pa bo prevzel delujoči napajalnik. Kot drugo nam bosta omogočala priklop strežnika na dve veji napajanja; v kolikor bo ena veja odpovedala, bo strežnik napajan preko druge. Za drzne skrbnike nudita dva napajalnika tudi možnost preklopa kablov (bodisi na dve veji napajanja bodisi zgolj zaradi urejanja) med delovanjem.

- **Diske bomo združili v redundančna diskovna polja (RAID)**. Redundančna polja nam bodo omogočala nemoteno delovanje strežnika v primeru odpovedi enega ali dovolj majhnega števila trdih diskov. Tudi zamenjava in sinhronizacija polja bo potekala v ozadju in brez izpada strežnika.

- **Uporabljali bomo kakovostne in strojne krmilnike RAID**. Morda je najmanj pomembno pravilo, vendar so krmilniki za SATA diske in programski krmilniki manj zanesljivi in predvsem ne omogočajo enakih možnosti po odpovedi diskov. Pri nekaterih se na primer diska ne da zamenjati ob delovanju strežnika, kar je za strežnike velika hiba.

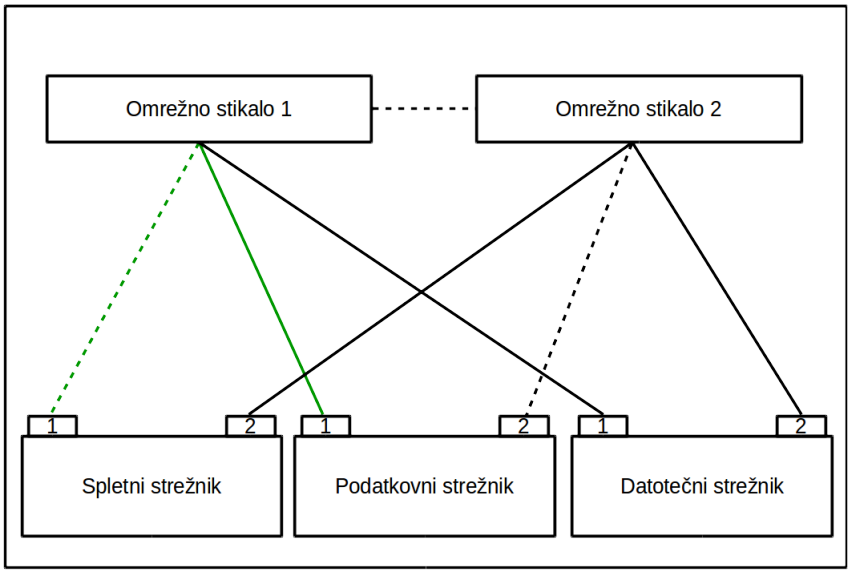
- **Omrežne kartice bomo združevali v redundančni način**. Podobno kot pri diskovnih poljih RAID bomo tudi omrežne kartice združevali zaradi redundance. V primeru odpovedi ene omrežne kartice bo začel promet avtomatično teči preko druge. Načelo bo bolj podrobno opisano kasneje.

- **Vsak napajalnik v napravi bomo priključili na ločen dovod električne energije**. S tem korakom bomo ustvarili redundanco tako za napajalnika, kot tudi za dovod električne energije. V primeru izpada enega ali drugega bo delo prevzel delujoči napajalnik ali delujoča veja električne energije.

- **Dovode električne energije bomo povezali preko naprave za brezprekinitveno napajanje (angl. UPS)**. Naprava za brezprekinitveno napajanje nudi napajanje kratek čas po izpadu električne energije, prav tako pa varuje strojno opremo pred prevelikimi nihanji električne napetosti.

Od navedenih pravil se bomo bolj podrobno ustavili le pri združevanju omrežnih povezav (angl. ethernet bonding, link aggregation, NIC teaming). Pri tem načinu dve ali več omrežnih kartic nastavimo tako, da lahko vse prenašajo isti promet. Stikalo mora imeti na drugi strani za iste povezave enake nastavitve, da začneta v primeru odpovedi ene omrežne kartice ali vrat na omrežnem stikalu oba komunicirati preko delujoče povezave. Kartice lahko delujejo v več načinih, od katerih sta za nas zanimiva predvsem dva; uravnoteženo izenačevanje obremenitve (angl. balance round robin - **mode 0**) in aktivno nadomeščanje (angl. active backup - **mode 1**). Prvi način pošilja polovico prometa po eni, polovico pa po drugi povezavi, kar pomeni, da je skupna hitrost dvakrat višja od hitrosti ene omrežne kartice. Ta način nudi

tudi redundanco v primeru odpovedi ene omrežne kartice. Drugi način pomeni, da bo šel promet preko ene kartice, dokler ta (ali stikalo na drugi strani) deluje. Omrežna kartica in stikalo si v vnaprej nastavljenih intervalih pošiljata kratka sporočila in dokler dobivata eden od drugega odgovore, povezava deluje. Če eden odgovora ne dobi, meni, da je povezava odpovedala in začne promet pošiljati preko druge povezave.



Slika 3.1: Slika prikazuje združevanje omrežnih povezav. Vsak strežnik je povezan v dve omrežni stikali. Zeleni črti prikazujeta pot povezave s spletnega strežnika na podatkovni strežnik. V primeru odpovedi omrežne kartice 1 na podatkovnem strežniku bo potekala ista povezava tako kot označujejo prekinjene črte.

Za združevanje omrežnih povezav je potrebno v operacijskem sistemu Linux naložiti modul *bonding* in fizičnim omrežnim povezavam nastaviti, da so sužnji za združeno povezavo. Združena povezava je samo navidezna, saj ji bo promet v resnici prenašala ena od fizičnih povezav. Opcije za modul se lahko doda kar pri nalaganju modula ali pa pri nastavitvi same povezave. Najpomembnejši sta opciji *mode* in *miimon*. Prva pove, na kakšen način naj komunicirata fizični povezavi, recimo kot izenačevanje obremenitve ali aktivno nadomeščanje, druga pa, na vsakih koliko milisekund bosta omrežna kartica in stikalo preverjala, če je povezava še aktivna. Na produkcijski arhitekturi uporabljamo

način aktivnega nadomeščanja in čas preverjanja 100 milisekund. Razlog za uporabo načina aktivnega nadomeščanja in ne prvega, pri katerem bi pridobili tudi na hitrosti, je, da se stikala včasih zmedejo, če MAC (angl. Media Access Control) naslov enega strežnika skače z enih vrat na druga. Ker s hitrostjo omrežja nimamo težav, smo se odločili za varnejšo opcijo.

Spodaj je primer združevanja omrežnih povezav za distribucijo CentOS za strežnik, ki ima dve povezavi: *eth0* in *eth1*. Vsaki omrežni povezavi moramo nastaviti, da je suženj, in določiti glavno povezavo. V datotekah ifcfg-eth v direktoriju */etc/sysconfig/network-scripts* bomo to storili z naslednjima vrsticama:

```
MASTER=bond0
SLAVE=yes
```

Združeni povezavi bomo v datoteki */etc/sysconfig/network-scripts/ifcfg-bond0* nastavili način povezovanja in pogostost preverjanja delovanja fizičnih povezav:

```
BONDING_OPTS="mode=1 miimon=100"
```

Primer izpisa ukaza *ip addr list* na strežniku z združenimi omrežnimi povezavami:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
2: eth0: <BROADCAST,MULTICAST,SLAVE,UP,LOWER_UP> mtu 1500 qdisc
mq master bond0 state UP qlen 1000
    link/ether 00:1d:09:03:c6:44 brd ff:ff:ff:ff:ff:ff
3: eth1: <BROADCAST,MULTICAST,SLAVE,UP,LOWER_UP> mtu 1500 qdisc
mq master bond0 state UP qlen 1000
    link/ether 00:1d:09:03:c6:44 brd ff:ff:ff:ff:ff:ff
4: bond0: <BROADCAST,MULTICAST,MASTER,UP,LOWER_UP> mtu 1500
qdisc noqueue state UP
    link/ether 00:1d:09:03:c6:44 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.20/24 brd 192.168.1.255 scope global bond0
```

Na podoben način se nastavi združevanje omrežnih povezav tudi na distribuciji Ubuntu.

Če se vrnemo na sliko 1.2, vidimo, da ima naš sistem 5 komponent:

- požarni zid;
- omrežno stikalo;
- spletni strežnik;
- podatkovni strežnik;
- datotečni strežnik.

Vsaka od teh komponent je lahko razlog za odpoved sistema, zato bomo morali zagotoviti redundanco za vsako od njih.

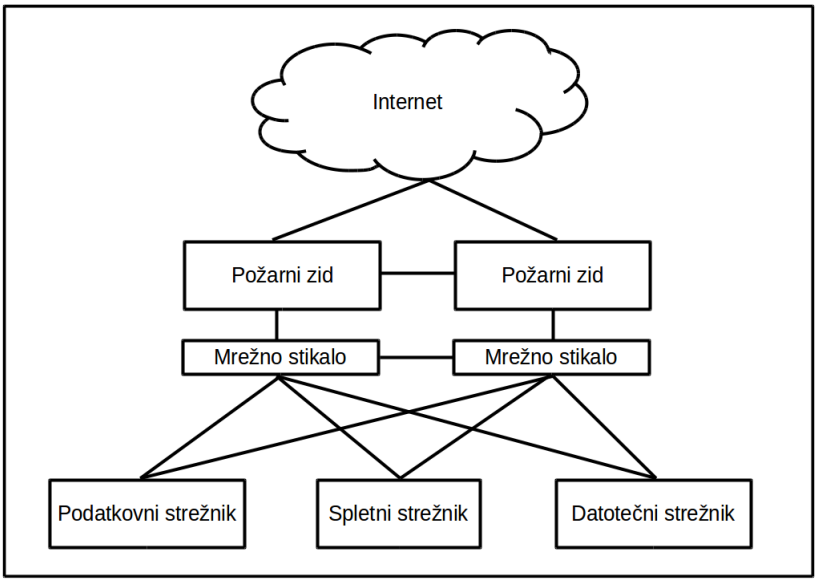
3.1 Požarni zid in omrežna oprema

Prvi dve točki bom združil v eno, saj je rešitev za obe enaka: visoko razpoložljiv grozd (angl. cluster). Pri grozdu dve ali več naprav povežemo med sabo, tako da lahko komunicirajo ena z drugo. V primeru odpovedi ene, druga naprava prevzame njeno delo. Dražje naprave imajo to rešitev podprto tako strojno (priključke za priklop kablov) kot tudi programsko (podpora za grozd). Požarni zid bomo torej razširili na dve napravi Cisco ASA, ki bosta medsebojno povezani. Medsebojno si bosta izmenjevali tudi trenutno stanje povezav, tako da bo druga vedno na tekočem s stanjem povezav na spletne strežnike.

Stikala imajo podobno rešitev, imenovano kopica (angl. stack). Z izjemo razlike v poimenovanju je rešitev skoraj identična, tako da bomo tudi stikala medsebojno povezali s temu namenjenimi podatkovnimi kabli. V primeru odpovedi enega bo njegovo delo prevzel drugi. Posledica takega povezovanja je, da morajo biti tudi strežniki povezani v obe stikali hkrati, kar pomeni, da združenih omrežnih povezav ne povežemo več v isto stikalo, temveč eno omrežno kartico v primarno stikalo, drugo omrežno kartico pa v sekundarno.

3.2 Spletni strežnik in izenačevalec obremenitve

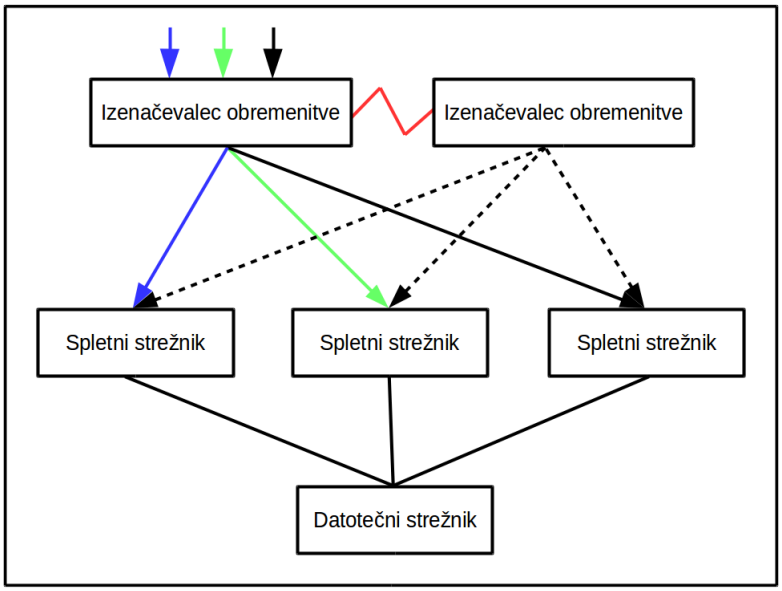
Spletni strežnik je najbolj obremenjena komponenta našega sistema, zaradi česar mora biti rešitev razširljiva in cenovno ugodna. Hkrati bi želeli pri spletnih strežnikih poleg razpoložljivosti povečati tudi hitrost. Dodajanje nadomestnega strežnika je v tem primeru nesmiselno, saj bi bilo potrebno ob nadgradnji menjati oba, obenem bi bil pa en neizkoriščen. Dejstvo, da smo



Slika 3.2: Slika prikazuje arhitekturo za aplikacijo s podvojeno omrežno opremo. Vsak strežnik je povezan na obe stikali in ima hkrati nastavljeno združevanje omrežnih povezav, tako da ob odpovedi ene mrežne povezave promet avtomatično preusmeri na drugo. Isto se zgodi, če odpove stikalo ali omrežna kartica na strežniku.

datotečni strežnik ločili od spletnega, nam omogoča narediti grozd strežnikov z izenačevanjem obremenitve (angl. load balancing). Grozd deluje tako, da je pred spletnimi strežniki dodatna komponenta, ki tem strežnikom deli delo. Ker je delitev dela enostavna, opravljanje dela pa zahtevno, je lahko izenačevalec obremenitve samo en za veliko spletnih strežnikov. Zaradi zanesljivosti moramo tudi novo komponento podvojiti, zato naredimo vroče pripravljen grozd (angl. hot standby cluster, hot failover cluster). Rešitev ima poleg redundance in hitrejšega delovanje še več prednosti, ki jih bomo omenili na koncu poglavja.

Grozd spletnih strežnikov bomo naredili z aplikacijo *Keepalived*[21], ki usmerja zahteve na transportni plasti (4. nivo) in preverja delovanje spletnih strežnikov na sejni plasti (5. nivo) po ISO/OSI standardu. Aplikacija deluje na način, da na določen čas odpira povezave http na spletne strežnike in vzdržuje seznam delujočih (tistih, ki odgovorijo pravilno in v ustreznem času). Tem strežnikom posreduje tudi uporabniške zahteve. V kolikor spletni strežnik odpove, aplikacija to zazna in strežnik odstrani s seznama. Ko



Slika 3.3: Slika prikazuje redundančno postavitev izenačevalcev obremenitve in spletnih strežnikov. Izenačevalec obremenitve deli zahteve vsem spletnim strežnikom. V kolikor en strežnik odpove, ga izenačevalec obremenitve označi kot nedelujočega in mu preneha pošiljati zahteve. V kolikor odpove izenačevalec obremenitve, njegovo delo prevzame nadomestni izenačevalec obremenitve.

je določena količina preverjanj ponovno uspešna, doda spletni strežnik nazaj na seznam delujočih. Na koliko časa aplikacija preverja zdravje spletnih strežnikov in koliko poskusov mora biti neuspešnih, da ga odstrani s seznama delujočih, ali uspešnih, da ga na seznam spet doda, se nastavi v konfiguraciji. Tu je potrebno poudariti, da se delovanje preverja za vsako stran posebej, ne za celoten strežnik. Če preverbi ena stran odgovori, druga pa ne, se samo tista, ki ni odgovorila, označi za nedelujočo.

Na podoben način si tudi izenačevalca obremenitve pošiljata sporočila, le da so ta v obliki protokola VRRP (angl. Virtual Router Redundancy Protocol). Paketi se izmenjujejo preko naslova za večvrstno oddajanje (angl. multicast), ki jih dobivata oba strežnika. Z njimi v resnici glasujeta, kdo od njiju bo deloval. Njuni funkciji nista enakovredni, že v konfiguraciji je eden določen za glavnega in drugi za nadomestnega. Dokler delujeta oba, delo opravlja glavni. Če glavni odpove, njegovo delo prevzame nadomestni izenačevalec obremenitve, vse dokler glavni ne začne ponovno delovati. Izenačevalca obremenitve

imata nastavljen vsak en naslov IP, ki je njun, ostale si pa delita med sabo (plavajoči naslovi IP) in jih ima nastavljene tisti, ki opravlja delo. Ko delo prevzame nadomestni izenačevalec obremenitve, se vsi plavajoči naslovi prenesejo nanj. Oba imata plavajoče naslove IP na obeh straneh omrežja, notranji in zunanji, ki se skupaj premikajo po strežnikih. Zunanji naslovi IP so nastavljeni za vsako spletno stran, ki bo na spletnih strežnikih. Promet, ki pride na te naslove, je usmerjen na enega od strežnikov. Na notranji strani mora biti vsaj en plavajoči naslov IP in tega uporabljajo spletni strežniki kot privzeti prehod. Zaradi načina delovanja (opravljata NAT - angl. Network Address Translation) morata imeti izenačevalca obremenitve na vsaki strani naslove IP iz drugega omrežja.

Spletni strežniki morajo biti nastavljeni enako kot bi bil en sam, le da morajo imeti privzeti prehod nastavljen na notranji naslov IP izenačevalca obremenitve in vsi na razpolago iste datoteke. Zaradi tega moramo datotečni strežnik nastaviti kot napravo NAS (angl. Network Attached Storage) in iste podatke pripeti vsem spletnim strežnikom.

Za potrebe nastavitev izenačevalcev obremenitev si bomo izmislili naslednje nastavitve:

```
eth0 notranja omrežna kartica
192.168.1.5 glavni izenačevalec
192.168.1.6 nadomestni izenačevalec
192.168.1.100 naslov IP za privzeti prehod spletnih strežnikov

eth1 zunanja omrežna kartica
192.168.2.5 glavni izenačevalec
192.168.2.6 nadomestni izenačevalec
192.168.2.100 plavajoči IP naslov za stran

192.168.1.10 prvi spletni strežnik
192.168.1.11 drugi spletni strežnik
```

Na izenačevalcih obremenitve bomo najprej namestili aplikacijo Keepalived z ukazom *yum install keepalived*. Aplikacija ima samo eno konfiguracijsko datoteko(*/etc/keepalived/keepalived.conf*), kjer je nastavljeno vse potrebno. Konfiguracijsko datoteko bomo razložili po delih. V *global_defs* bomo najprej nastavili nekaj splošnih nastavitev, kot so naslov, kamor se bodo pošiljala obvestila o spremembah, poštni strežnik in unikatno ime strežnika. Pošiljanje obvestil ni obvezno, zato lahko, če nimamo te potrebe ali možnosti, ta del izpustimo.

```
global_defs {
  notification_email {
    admin@trgovina.tld
  }
  notification_email_from lb@trgovina.tld
  smtp_server 192.168.1.?
  smtp_connect_timeout 100
  lvs_id trgmaster
}
```

Sledita definiciji obeh omrežnih kartic in njunih nastavitev. Tu bomo videli, da so pri obeh med drugim nastavljene prioritete (po tem se ločita glavni in nadomestni strežnik), intervali za obveščanje s protokolom VRRP, osnovna avtentikacija in vsi naslovi IP, ki jih mora omrežna povezava nastaviti.

```
vrrp_instance zunanja {
  state MASTER
  interface eth1

  lvs_sync_daemon_interface eth1
  virtual_router_id 20
  priority 100
  advert_int 1
  smtp_alert
  authentication {
    auth_type PASS
    auth_pass geslo1
  }

  virtual_ipaddress {
    192.168.2.100
  }
}

vrrp_instance notranja {
  state MASTER
  interface eth0
  lvs_sync_daemon_interface eth0
  virtual_router_id 21
  priority 100
```

```
  advert_int 1
  smtp_alert
  authentication {
    auth_type PASS
    auth_pass geslo2
  }
  virtual_ipaddress {
    192.168.1.100
  }
}
```

Večina nastavitev ima dovolj opisno že ime, zato jih ni potrebno posebej opisovati. Omenimo samo naslednje:

- **lvs_sync_daemon_interface** določa, preko katere omrežne povezave naj se pošilja promet VRRP za to instanco;
- **virtual_router_id** določa identiteto instance VRRP. Je izmišljena, vendar mora biti v omrežju unikatna. Tudi naši dve instanci imata nastavljeni unikatni identiteti;
- **advert_int** določa, kako pogosto naj se pošiljajo paketi VRRP.

Dodali bomo še del, ki skrbi, da bosta zunanja in notranja instanca vedno na istem strežniku. Brez teh nastavitev bi bili omrežni povezavi neodvisni in bi se lahko ena nastavila na enem strežniku, druga pa na drugem. Spodnji del to prepreči:

```
vrrp_sync_group Trgovina {
  group {
    zunanja
    notranja
  }
}
```

S tem smo splošno nastavitev izenačevalcev obremenitve zaključili. Ostane nam samo še nastavitev za našo stran in strežnike, ki jo bodo stregli.

Za naše nastavitve in dva spletna strežnika bomo zapisali spodnjo konfiguracijo:

```
virtual_server 192.168.2.100 80 {
  delay_loop 10
  lb_algo wlc
  lb_kind NAT
  protocol TCP
  real_server 192.168.1.10 80 {
    weight 10
    HTTP_GET {
      url {
        path /
        digest 60b725f10c9c85c70d97880dfe8191b3
      }
      connect_port 80
      connect_timeout 5
      nb_get_retry 2
      delay_before_retry 2
    }
  }
  real_server 192.168.1.11 80 {
    weight 10
    HTTP_GET {
      url {
        path /
        digest 60b725f10c9c85c70d97880dfe8191b3
      }
      connect_port 80
      connect_timeout 5
      nb_get_retry 2
      delay_before_retry 2
    }
  }
}

virtual_server 192.168.2.100 443 {
  delay_loop 10
  lb_algo wlc
  lb_kind NAT
```

```
  protocol TCP
  real_server 192.168.1.10 443 {
    weight 10
    SSL_GET {
      url {
        path /
        digest 30ca33d61d76ee3238e640b918c2e747
      }
      connect_port 443
      connect_timeout 5
      nb_get_retry 2
      delay_before_retry 2
    }
  }
  real_server 192.168.1.11 443 {
    weight 10
    SSL_GET {
      url {
        path /
        digest 30ca33d61d76ee3238e640b918c2e747
      }
      connect_port 443
      connect_timeout 5
      nb_get_retry 2
      delay_before_retry 2
    }
  }
}
```

virtual_server je zunanji omrežni naslov naše strani. Zanj je nastavljeno preverjanje delovanja spletnih strežnikov preverjalo vsakih 10 sekund. Algoritem za razporejanje povezav je wlc (angl. Weighted Least-Connections), kar pomeni, da bo nova povezava poslana strežniku z najmanj odprtimi povezavami v odvisnosti od njegove uteži. Obstajajo tudi drugi algoritmi, vendar se je na produkcijskem okolju ta izkazal za najbolj uravnoteženega, ki najbolje deluje tudi z različno zmožnimi strežniki. Sledijo definicije posameznih spletnih strežnikov (imenovanih *real_server*). Njim se določi utež, način preverjanja, če strežniki delujejo in časovne omejitve ter količina preverjanj. V zgornjem primeru imajo vsi strežniki enako utež, kar pomeni, da bodo dobivali enako količino povezav. Delovanje se bo preverjalo na korenskem naslovu, kjer

mora biti zgoščena funkcija md5 odgovora enaka nastavljeni v konfiguracijski datoteki. V nasprotnem primeru bo stran označena kot nedelujoča. Prav tako bo označena kot nedelujoča, če dvakrat zapovrstjo v petih sekundah ne bo odgovorila na zahtevo izenačevalca obremenitve. Na enak način kot http so nastavljene tudi https strani.

Ker imata izenačevalca obremenitve že v nastavitvah določeno, kateri je glavni in kateri nadomestni, bomo na nadomestnem v konfiguracijski datoteki spremenili tri nastavitve. Te tri mu bodo povedale, da je nadomestni strežnik in naj ne poganja ničesar, dokler glavni strežnik deluje.

```
lvs_id trgbackup
state BACKUP
priority 50
```

Keepalived vsebuje orodje za administracijo z imenom *ipvsadm*, s katerim lahko spremljamo stanje strani in delovanja strežnikov. Lahko tudi dodajamo in brišemo spletne strežnike ter jim spreminjamo uteži. Za našo konfiguracijo na treh spletnih strežnikih je izpis tak:

```
TCP  192.168.2.100:80 wlc
  -> 192.168.1.10:80              Masq    10     0          1
  -> 192.168.1.11:80              Masq    10     0          1
TCP  192.168.2.100:443 wlc
  -> 192.168.1.10:443             Masq    10     0          1
  -> 192.168.1.11:443             Masq    10     0          1
```

Izpis nam pove, da za stran na omrežnem naslovu 192.168.2.100 (za protokol http in https) delujeta oba spletna strežnika, oba imata utež 10, nič aktivnih povezav in eno neaktivno.

Z nastavitvijo izenačevalcev obremenitve je potrebno prenastaviti tudi preslikave zunanjih omrežnih naslovov v notranje na način, da preslikava za zunanji naslov ne kaže več neposredno na spletni strežnik (192.168.1.10), temveč na zunanji omrežni naslov na izenačevalcu obremenitev (192.168.2.100). S to spremembo bomo uporabili dve omrežni kartici za ločitev zunanjega in notranjega omrežja na izenačevalec obremenitev. Na strežnikih z manj kot štirimi omrežnimi karticami bomo s tem onemogočili združevanje povezav in poslabšali zanesljivost strežnika. Rešitvi sta dve:

- uporaba strežnikov s štirimi omrežnimi karticami
- uporaba VLAN-ov

V našem primeru bomo uporabili VLAN (angl. Virtual LAN), ki ga bomo zaključili na izenačevalcu obremenitev. Na ta način bo promet za obe omrežji tekel preko iste omrežne kartice, zaradi česar bomo lahko tudi zgolj dve omrežni kartici združili v redundančno povezavo.

Za to delo smo predpostavili, da bodo statično vsebino stregli strežniki Apache. To ni nujno, saj lahko na spletnih strežnikih poleg strežnika Apache namestimo tudi strežnik Lighttpd ali Nginx. Za statično vsebino lahko ustvarimo novo domeno, ki jo usmerimo na novi spletni strežnik. Tega nastavimo na tak način, da streže vsebino iz direktorija /var/www/html/podatki, kamor smo priključili vsebino datotečnega strežnika. Ker je ta vsebina na razpolago vsem spletnim strežnikom, lahko na izenačevalcu obremenitve naredimo novo obličje in zaledje ter zahteve za statično vsebino prav tako razporejamo na vse spletne strežnike. Novi spletni strežnik bo statično vsebino stregel bistveno bolj učinkovito kot strežnik Apache.

3.3 Podatkovni strežnik

Za nastavitev podatkovnega strežnika bomo določili naslednje fiktivne podatke:

```
192.168.1.20 omrežni naslov, na katerem bo dosegljiva baza
192.168.1.21 (baza1) prvi podatkovni strežnik
192.168.1.22 (baza2) nadomestni podatkovni strežnik
/dev/vg1/lv_baza replicirana particija na nadomestni strežnik
/dev/drbd0 naprava za dostop do DRBD particije
/var/lib/pgsql lokacija za priklop naprave /dev/drbd0
```

V idealnem primeru bi podatkovne strežnike postavili v grozd enakovrednih strežnikov, vendar strežnik Postresql tega ne podpira. Podpira le replikacijo baz na drug strežnik. S pomočjo replikacije bi lahko ustvarili grozd, v katerem bi imeli eno bazo za pisanje replicirano na več baz za branje. Ker je največ ravno bralnih transakcij, bi te razporedili na baze za branje, transakcije, ki pišejo podatke, pa bi morali vse usmeriti na prvo bazo. Prednost take arhitekture bi bila uporabljena funkcionalnost strežnika Postgres, ki je gotovo zanesljiva, vendar bi imela taka rešitev več slabosti kot prednosti. Tudi baza za pisanje bi morala imeti svojo redundanco, kar pomeni, da bi za grozd ene baze potrebovali vsaj štiri strežnike. Poleg tega bi bilo potrebno za to rešitev našo aplikacijo spremeniti tako, da bi na pravih korakih uporabljala prave podatkovne strežnike.

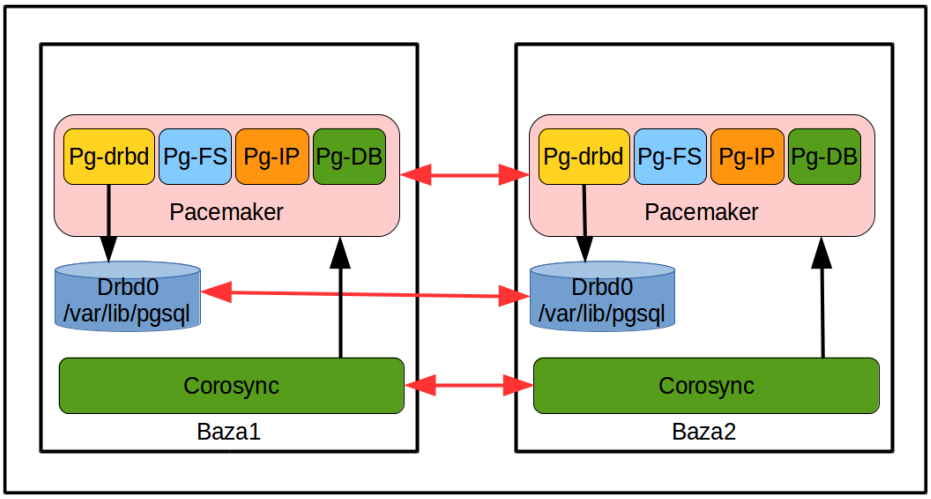
Boljša možnost bi bil grozd baz z uporabo aplikacije *Pgpool*[24]. Pgpool je aplikacija, ki jo postavimo pred podatkovni strežnik. Odjemalci pošiljajo zahteve aplikaciji Pgpool, ta pa naprej na vse baze. Aplikacija zna sama vzdrževati zgoraj opisan grozd tako, da bralne ukaze razporeja po najmanj zasedenih strežnikih, pisalne pošlje pa vsem. Na ta način zagotavlja, da so vse baze identične. Dodatna prednosti take rešitve je, da naše aplikacije ne bi bilo potrebno spreminjati. Ostaja pa težava velikega števila strežnikov, kajti tudi strežnik, na katerem bi tekel Pgpool, bi bilo potrebno podvojiti.

Kljub temu, da imamo na produkcijskem okolju več spletnih strežnikov, uporablja aplikacija toliko predpomnjenja, da je podatkovni strežnik zelo malo obremenjen. Zaradi tega smo se v primeru baz omejili samo na razpoložljivost, ne pa tudi na povečanje hitrosti. Kot najbolj primerna in cenovno ugodna rešitev se je tako izkazala uporaba dodatnega strežnika v obliki vroče rezerve. Z odprtokodnimi programi smo rešitev našli v kombinaciji programov *DRBD*[25], *Corosync*[26] in *Pacemaker*[27]. Vsak od teh programov teče na vsakem strežniku in prevzame svojo vlogo v grozdu.

Program DRBD skrbi za replikacijo podatkov med dvema ali več strežniki. Vsak strežnik mora imeti namensko particijo, ki jo bo DRBD v realnem času repliciral na drug strežnik. Replikacija deluje na nivoju blokov na disku, zato je vseeno, katere vrste podatki so zapisani na particiji. Particija je lahko v dveh stanjih, primarnem in sekundarnem. Tok podatkov vedno teče od primarnega k sekundarnemu. V kolikor ni drugače določeno, je particija v primarnem stanju zgolj na enem strežniku in le na tem tudi v uporabi.

Program Corosync nadzoruje stanje strežnikov v grozdu. Nameščen mora biti na obeh strežnikih v grozdu in deluje tako, da si instanci na določen časovni interval pošiljata sporočila, imenovana srčni utripi (angl. heartbeat). V primeru, da en strežnik neha delovati, instanca na delujočem strežniku to sporoči programu Pacemaker.

Program Pacemaker je tisti, ki poganja in ustavlja vse aplikacije, priključuje particije in spreminja vlogo DRBD particij. Prav tako kot program Corosync mora biti nameščen na obeh strežnikih in je edini opisani program, ki se mora pognat ob zagonu strežnika. Ostali morajo biti onemogočeni, saj jih zažene program Pacemaker. Njegova konfiguracija je v xml formatu, nastavlja pa se jo s temu namenjenim orodjem, ki ga bomo opisali v nadaljevanju. V konfiguraciji definiramo vire (angl. resource) in njihove lastnosti ter pravila, ki te vire obravnavajo.



Slika 3.4: Slika prikazuje redundančno postavitev podatkovnega strežnika z uporabo programov DRBD, Corosync in Pacemaker. Program DRBD poskrbi, da se podatki replicirajo z glavnega na nadomestni strežnik. V primeru odpovedi glavnega strežnika program Corosync obvesti program Pacemaker, ki poskrbi, da se vsi programi v ustreznem vrstnem redu poženejo na nadomestnem strežniku.

Pravila obsegajo med drugim:

- vrstni red nalaganja virov;
- odvisnosti virov med sabo;
- odvisnosti virov od stanja ostalih virov;
- združevanja virov v skupine.

3.3.1 DRBD

Program DRBD komunicira preko vrat 7789, ki morajo biti na požarnem zidu odprta za sosednji strežnik v grozdu. Osnovna konfiguracija DRBD je

v /etc/drbd.d/global_common.conf. Tam bomo nastavili vse splošne nastavitve, kot so algoritmi za prenos in preverjanje podatkov, način potrjevanja, hitrost prenosa, procedure ob napakah, količine medpomnilnikov in podobno. Nastavitve so specifične za vsak sistem in odvisne od mnogih podrobnosti, kot je hitrost procesorja in količina pomnilnika v strežniku, hitrost omrežne opreme in povezav, skrbnikovih zahtev, zato dokaj obsežne datoteke ne bomo izpisovali. Tudi točnih ukazov za začetno vzpostavitev particij ne bomo omenjali posebej, ker so dobro dokumentirani v navodilih za DRBD. Poleg osnovnih nastavitev bomo nastavili tudi sam vir, ki mu bomo določili tudi namensko particijo. Datoteke za vire bomo ustvarili v istem direktoriju in jih poimenovali s končnico ".res". Ker bodo kasnejše nastavitve temeljile na konfiguracijski datoteki, bomo kar takoj ustvarili datoteko /etc/drbd.d/baza.res:

```
resource baza {
  on baza1 {
    device    /dev/drbd0;
    disk      /dev/vg1/lv_baza;
    address   192.168.1.21:7789;
    meta-disk internal;
  }
  on baza2 {
    device    /dev/drbd0;
    disk      /dev/vg1/lv_baza;
    address   192.168.1.22:7789;
    meta-disk internal;
  }
}
```

Iz konfiguracije je razvidno, da bosta v grozdu dva strežnika, *baza1* in *baza2*. Sinhronizacijo bosta en z drugim vzpostavila preko vrat 7789. Oba bosta uporabljala particijo */dev/vg1/lv_ baza*, do katere bosta dostopala preko naprave */dev/drbd0*.

Za nastavitev particij DRBD bomo uporabili orodje *drbdadm*[28] in na izbrano particijo najprej zapisali metapodatke. Zatem do particije ne bomo dostopali več neposredno, temveč preko naprave (angl. device) /dev/drbd0. Na napravo /dev/drbd0 bomo naprej zapisali datotečni sistem, jo priključili na izbran direktorij ter nanjo zapisali ustrezne podatke. Ko bosta particiji DRBD pravilno nastavljeni na obeh strežnikih, bodo vsi zapisani podatki avtomatično preneseni na nadomestni strežnik. Ob prvi uspešni povezavi obeh particij se bodo podatki med njima avtomatično sinhronizirali.

Po uspešni vzpostavitvi sinhronizacije obeh strežnikov, si lahko trenutno stanje DRBD particij ogledamo z ukazom *drbd-overview*:

```
0:baza/0  Connected Primary/Secondary UpToDate/UpToDate
/var/lib/pgsql ext4 99G 38G 56G 41%
```

Izpis nam pove, da je vir z imenom baza uspešno povezan z nadomestnim strežnikom, da je na tem strežniku v primarnem in na drugem strežniku v sekundarnem stanju. Obe particiji sta sinhronizirani. Particija je na tem strežniku priključena pod /var/lib/pgsql. Datotečni sistem je ext4, velika je 100 gigabajtov, od česar je 38 gigabajtov zasedenih (41 odstotkov).

Za priključitev iste particije na drugem strežniku moramo storiti naslednje:

- odklopimo particijo na prvem strežniku (angl. umount);
- nastavimo stanje particije kot sekundarno;
- nastavimo stanje particije na drugem strežniku kot primarno;
- priključimo particijo na drugem strežniku (angl. mount).

Replikacija se avtomatično ustavi pri drugi in obrne smer pri tretji točki. Na ta način lahko particiji poljubno menjamo in do istih podatkov dostopamo na kateremkoli strežniku. Da bi to počeli ročno, je preveč zamudno, zato bomo to delo v nadaljevanju prepustili programu Pacemaker, ki bo za preklop baze na nadomestni strežnik opravil isti postopek, s tem, da bo poskrbel tudi za premik plavajočega omrežnega naslova in zagon baze Postgresql.

Orodje drbdadm podpira tudi nekaj drugih skrbniških ukazov. Med bolj uporabnimi so naslednji:

- **Verify** - ponovna sinhronizacija podatkov med dvema viroma.
- **Resize** - povečevanje particije. Particijo povečamo po povečanju spodnje ležeče particije z ukazom *lvextend* (če smo naredili particijo tipa LVM) in pred povečanjem zgornje ležečega datotečnega sistema z ukazom *resize2fs*. Particijo lahko tudi zmanjšamo, vendar je postopek zahtevnejši.
- **Up/Down** - omogočimo ali onemogočimo vir. Vire moramo onemogočiti pri vzdrževalnih delih.
- **Attach/Detach** - viru priklopimo ali odklopimo particijo. V tem primeru ostane vir brez particije (angl. diskless).

- **Connect/Disconnect** - povežemo ali prekinemo povezavo virov med seboj. Vira postaneta samostojna (angl. Standalone). Dokler ju ponovno ne povežemo, se podatki ne prenašajo na drug strežnik.
- **Primary/Secondary** - preklop particije v primarno ali sekundarno stanje.
- **Invalidate/Invalidate-remote** - ukažemo, naj se particiji sinhronizirata na tak način, da se bodo prepisali podatki na tem ali drugem strežniku (na tem, kjer smo pognali ukaz, ali na drugem). To je potrebno storiti v primeru stanja, v katerem ni jasno, kateri strežnik ima zadnje podatke. Do omenjenega stanja, imenovanega razcepljena osebnost (angl. split-brain), pride, če se prekine medsebojna povezava med viri in se particiji na obeh strežnikih preklopita v primarno stanje. V tem primeru mora težavo rešiti skrbnik.

3.3.2 Corosync

Kot je bilo že omenjeno, programa Corosync na obeh strežnikih komunicirata med sabo in v primeru izpada enega delujoči to sporoči svojemu programu Pacemaker, ki sproži ustrezne procedure za popravo stanja, v kolikor je to potrebno.

Za nastavitev bomo z ukazom *corosync-keygen* najprej ustvarili ključ za avtentikacijo, katerega bomo zapisali na oba strežnika v */etc/corosync/authkey*. V to datoteko se zapiše ključ, s katerim se bosta obe instanci programa Corosync avtenticirali v grozd. Konfiguracija programa se nahaja v datoteki */etc/corosync/corosync.conf*. Spodaj bomo omenili samo pomembnejše nastavitve privzete konfiguracijske datoteke:

```
totem {
        version: 2

        crypto_cipher: none
        crypto_hash: none

        interface {
                ringnumber: 0
                bindnetaddr: 10.16.35.0
                mcastport: 5405
                ttl: 1
```

```
        }
        transport: udpu
}
```

Najprej je določeno, za katero verzijo programa Corosync je oblika konfiguracije. V našem primeru je napisana za drugo verzijo. Privzeta konfiguracija ima izključeno šifriranje in avtenticiranje sporočil. Če nista onemogočena, je privzet algoritem za šifriranje *aes256* in zgoščevalna funkcija za podpisovanje *sha1*. Opcija *interface* ni obvezna, vsebuje pa nastavitve za komuniciranje med dvema strežnikoma. Komunikacija bo potekala preko vrat 5405, ki morajo biti odprta v požarnem zidu na obeh strežnikih. Za naše potrebe smo nastavili protokol UDP. Če v konfiguraciji fiksno nastavimo oba strežnika, lahko komunikacijo nastavimo samo med dvema strežnikoma (angl. unicast), namesto večvrstnega oddajanja (angl. multicast). V tem primeru za *transport* nastavimo udpu (angl. unicast UDP). Za udpu protokol moramo določiti še oba strežnika:

```
nodelist {
    node {
        ring0_addr: baza1
        nodeid: 1
    }

    node {
        ring0_addr: baza2
        nodeid: 2
    }
}
```

Vsakemu strežniku smo določili naslov (v našem primeru ime) in identiteto strežnika, ki sta poljubna, vendar morata biti unikatna. Po tej konfiguraciji bosta strežnika začela medsebojno komunicirati in uspešno postavila grozd.

Razloga, da smo se odločili za obveščanje neposredno med strežnikoma in ne za večvrstno oddajanje, sta dva; prvi je omejevanje nepotrebnega prometa po omrežju. V tem primeru sicer ne zaradi količine prometa, temveč zaradi možnosti težav v prihodnosti, ko bi lahko promet motil druge naprave, priključene v isto omrežje. Drugi razlog je ta, da smo strežnika grozda nastavili že v konfiguraciji. Tako ni možnosti, da bi se recimo nespretno postavljen testni strežnik priključil v produkcijski grozd in ogrozil njegovo delovanje.

3.3.3 Pacemaker

Program Pacemaker nima svoje konfiguracijske datoteke, temveč se ga nastavlja preko orodja *pcs* (ali *crm* na distribuciji Ubuntu). Svoje nastavitve, do katerih naj ne bi dostopali neposredno, hrani v formatu xml. Ker se konfiguracija v realnem času sinhronizira med obema strežnikoma, lahko grozd nadzorujemo s katerega koli strežnika.

Za našo arhitekturo bomo nastavili naslednjo konfiguracijo:

```
pcs property set stonith-enabled=false
pcs property set no-quorum-policy=ignore
pcs resource create pg-drbd \
ocf:linbit:drbd drbd_resource="pgsql" op monitor interval=30s
pcs resource master ms-pg-drbd pg-drbd master-max=1 \
master-node-max=1 clone-max=2 clone-node-max=1 notify=true
pcs resource create pg-FS Filesystem device="/dev/drbd0" \
directory="/var/lib/pgsql" fstype="ext4"
pcs resource create pg-IP ocf:heartbeat:IPaddr2 params \
ip=192.168.1.20 cidr_netmask=32 op monitor interval=10s
pcs resource create pg-DB ocf:heartbeat:pgsql op start \
interval=0 timeout=60s op stop interval=0 timeout=60s
pcs resource group add pg-group pg-FS pg-IP pg-DB
pcs constraint colocation add pg-group ms-pg-drbd INFINITY \
with-rsc-role=Master
pcs constraint order promote ms-pg-drbd then start pg-group
```

S *pcs property* nastavimo neko splošno lastnost grozda, ki določa njegovo obnašanje. V našem primeru smo onemogočili funkcijo STONITH, ki po potrebi onemogoči nepravilno delujoče strežnike. Ker imamo samo dva strežnika, ne moremo uporabljati sklepčnosti (angl. quorum), zato nastavimo ignoriranje nesklepčnosti.

S *pcs resource* nastavljamo posamezne vire. V našem primeru smo nastavili:

- **DRBD vir - pg-drbd**. To je vir, ki smo ga nastavili v konfiguraciji za DRBD. Ta vir bo program Pacemaker upravljal tako, da bo samo na enem strežniku v primarnem načinu.
- **Datotečni sistem - pg-FS**. Na strežniku z virom DRBD v primarnem načinu bo Pacemaker tudi priključil particijo v ustrezni direktorij. S tem korakom bo postal datotečni sistem za replicirano particijo dosegljiv za

branje in pisanje. Da ta korak uspe, mora biti DRBD vir v primarnem stanju.

- **Plavajoči naslov IP - pg-IP**. Plavajoči omrežni naslov je tisti, na katerem bo dosegljiva baza, zato mora biti nastavljen še pred zagonom baze.

- **Bazo - pg-DB**. Baza je bistvo našega strežnika in vse konfiguracije. Edini pogoj za bazo je njeno neprestano delovanje. V nasprotnem primeru se z upoštevanjem pravil in ostalih virov poskrbi, da se lahko spet požene. V kolikor baza ugasne zaradi odpovedi strežnika, bodo vsi ostali viri prestavljeni na nadomestni strežnik. Ta vir je odvisen od datotečnega sistema in plavajočega naslova IP.

- **Skupino vseh virov - pg-group**. Skupino vseh virov ustvarimo z namenom določitve vrstnega reda poganjanja virov in lažjega upravljanja. Namesto, da bi za vsak vir določali pravila odvisnosti od ostalih virov in vrstnega reda (kar bi drugače morali), lahko to naredimo s skupino. Viri se poganjajo po vrsti, od leve proti desni, in skupina poskrbi, da sta recimo datotečni sistem in plavajoči IP naslov vzpostavljena, preden se požene baza. Vrstni red se upošteva tudi pri ugašanju virov, in sicer v obratnem vrstnem redu.

S *pcs constraint* nastavljamo medsebojne zahteve in omejitve za vire. Prvo pravilo pravi, da se lahko skupina požene samo na strežniku, na katerem je particija DRBD v primarnem stanju. To je seveda nujno, saj jo lahko edino tam priključimo na strežnik in dostopamo do datotečnega sistema. Pri drugem pravilu pa je določen vrsti red, in sicer je potrebno particijo najprej spremeniti v primarno, šele nato pognati vse vire iz skupine. Brez zadnjega pravila bi se namreč viri lahko začeli poganjati še preden bi bil DRBD pripravljen, kar bi vrnilo zgolj napako, ne pa delujoče baze.

Omejitvam lahko nastavljamo tudi prioritete in s tem dopustimo, da pomembnejša pravila prevladajo nad manj pomembnimi. Omejitve sodijo v tri sklope:

- **Lokacija**. S tem pravilom vir ali skupino fiksiramo na določen strežnik. Prekrši se le v primeru, če pride do situacije, ko prevlada pravilo z večjo oceno.

- **Vrstni red**. S tem pravilom določamo vrstni red poganjanja virov. Pri ustavljanju velja obratni vrsti red. V našem primeru tega pravila nimamo, saj ga vsebuje že skupina.

- **Kolokacija**. Pravilo določa sobivanje virov. Določeni viri so medsebojno odvisni in ti se morajo vedno pognati na istem strežniku.

Naš primer je zelo enostaven. Možno je nastaviti tudi zahtevnejše rešitve, recimo grozd dveh baz na treh strežnikih, ki ima samo en nadomestni strežnik. V tem primeru je potrebno nastaviti tudi odvisnosti med njimi; recimo, da lahko vsaka baza teče le na svojem osnovnem ali nadomestnem strežniku, ne pa tudi na osnovnem strežniku druge baze. Za boljši izkoristek lahko določimo tudi, da bazi ne smeta teči na istem strežniku (recimo obe na nadomestnem). Prvo pravilo so seveda delujoči viri, zato se dodatne omejitve ignorirajo, če bi njihovo upoštevanje pomenilo, da bazi ne bi mogli delovati. Če bi recimo odpovedala oba osnovna strežnika, bi se obe bazi pognali na nadomestnem, kljub pravilu, da ne smeta teči na istem.

Za pregled stanja grozda lahko uporabimo ukaz *pcs status*, ki izpiše vse aktivne strežnike v grozdu, seznam virov in na katerem strežniku tečejo. Zelo podoben izpis ima ukaz *crm_ mon*, ki za povrhu stanje še osvežuje. Na primer: pri migraciji baze na rezervni strežnik lahko vidimo, kako se viri na osnovnem ustavljajo in na rezervnem prižigajo. Primer izpisa ukaza crm_mon je (zaradi jasnosti izpisa smo nekaj vrstic izpustili):

```
Last updated: Thu Mar  3 14:58:44 2016 Last change:
Wed Feb 17 23:39:50 2016
Stack: corosync
Current DC: baza2 (version 1.1.13-10.el7_2.2-44eb2dd)
 - partition with quorum
2 nodes and 6 resources configured

Online: [ baza1 baza2 ]

 Master/Slave Set: ms-pg-drbd [pg-drbd]
     Masters: [ baza1 ]
     Slaves: [ baza2 ]
 Resource Group: pg-group
     pg-FS (ocf::heartbeat:Filesystem):    Started baza1
     pg-IP (ocf::heartbeat:IPaddr2): Started baza1
     pg-DB (ocf::heartbeat:pgsql): Started baza1
```

3.4 Datotečni strežnik

V naši arhitekturi je ostal neredundančen samo še datotečni strežnik. Po nastavitvi visoko razpoložljive baze se nam enaka rešitev kar sama ponuja tudi za datotečni strežnik. Program Pacemaker veliko datotečnih protokolov (SMB, CIFS, NFS) že privzeto podpira, zato rešitev ne bi bila težka in veliko drugačna od grozda baz.

Vendar pa za produkcijsko arhitekturo nismo izbrali tega pristopa. Taka rešitev bi namreč imela za nas nekaj pomanjkljivosti; datotečna strežnika sta bila visoka eno enoto (angl. 1U) in sta imela prostora za zgolj dva trda diska. Glede na to, da SAS diski niso zelo veliki, smo bili že nekaj časa omejeni s prostorom. Nakup večjih diskov bi težavo rešil le za kratek čas.

Na dveh diskih tudi dostop do podatkov ni hiter. Za čim hitrejši dostop smo se držali tega, da so bili na datotečnih strežnikih samo produkcijski podatki, vendar to ni bila dolgoročna rešitev. Varnostne kopije podatkov smo imeli takrat na tretjem strežniku, ki je bil ravno tako že prezaseden.

Za razliko od preklopa baz bi imeli pri datotečnem strežniku nekaj več težav, saj povezava spletnih strežnikov na datotečni strežnik ni samo ena od TCP povezav, ki jih naredi aplikacija (če ena povezava ne uspe, poizkusi z naslednjo), temveč jo vzpostavi operacijski sistem ob zagonu strežnika, potem pa ne več. Zagotoviti bi bilo potrebno, da bi imele povezave med spletnimi in datotečnim strežnikom daljši čas omejitve (angl. timeout), kot bi trajal preklop osnovnega na nadomestni datotečni strežnik ali pa da bi jo znali spletni strežniki ponovno vzpostaviti tudi med delovanjem.

Za tako rešitev bi morali torej kupiti dva nova strežnika višine 2U in v vsakega vstaviti po šest diskov. Zaradi visoke cene takega nakupa smo se namesto za strežnike odločili za diskovni sistem. V sistem smo namestili 14 diskov SATA in jih nastavili kot redundančno polje diskov. Diskovni sistem ima podvojeno napajanje in dva krmilnika, ki sta popolnoma redundančna in v primeru odpovedi enega vse delo prevzame drugi. Tudi v omrežje smo oba povezali redundančno. V istem koraku smo tako za datotečni strežnik zagotovili visoko razpoložljivost, znatno povečanje prostora in tudi hitrosti. Poleg skupnih podatkov za spletne strežnike smo na diskovni sistem prenesli tudi varnostne kopije in vse podatke centralizirali. Diskovni sistem nudi ostalin strežnikom podatke preko protokola NFS za statično spletno vsebino in varnostne kopije ter iSCSI za slike virtualnih strežnikov.

3.5 Druge vrste odpovedi

V prejšnjem poglavju smo pogledali, kako se lahko izognemo različnim vrstam odpovedi našega sistema. Najprej smo naredili arhitekturo odporno na izpad električne energije. Odpove lahko ena cela veja za dlje časa ali celo obe za manj, odvisno od zmogljivosti naprav za neprekinjeno napajanje. Odpove lahko tudi polovica omrežne opreme, po en napajalnik v vsakem strežniku, polovica omrežnih kartic na strežnik, lahko odpove celo polovica pravih strežnikov, pa bo naša aplikacija še vedno delovala. V grozdu spletnih strežnikov jih lahko odpove celo več kot pol. S strojnega stališča smo za redundanco dobro poskrbeli.

Težave pa lahko nastanejo tudi s programsko opremo. Pri nadgradnjah bodisi aplikacije, bodisi strežnikov, je zato nujno spremembe najprej preveriti na enem ali nekaj strežnikih, šele nato nadgraditi vse. Ker hočemo imeti v končni fazi vse strežnike posodobljene, je vseeno, če so redundančni, če bomo iste napake vnesli na vse. Zaradi tega je dobro narediti tudi redundanco okolja, v katerem teče aplikacija. Ni nujno, da so vsa okolja enako kompleksna, saj gre zgolj za testiranje združljivosti med recimo spletnim strežnikom, verzijo jezika PHP in baze Postgresql ter za preverjanje same funkcionalnosti aplikacije. Za produkcijsko aplikacijo uporabljamo štiri okolja:

- **prvo testno**, kjer preverimo, da je aplikacija združljiva z vsemi komponentami arhitekture;
- **drugo testno**, kjer preverimo, da aplikacija deluje kot je potrebno z uporabniškega stališča;
- **obremenitveno**, kjer preverimo, da se hitrost delovanja ni bistveno spremenila;
- **produkcijsko**, kjer aplikacija dejansko prodaja vstopnice za uporabnike.

Kot zadnjega izpostavimo zelo pogost razlog za izpad delovanja, in sicer človeško napako. Težava človeške napake je, da imamo konfiguracijo grozdov po navadi avtomatizirano, kar pomeni, da napake repliciramo na vse strežnike hkrati. Zato je potrebno avtomatizacijo prilagoditi tako, da naenkrat spreminja le del strežnikov v grozdu. Če so spremembe uspešne, se jih aplicira še na ostale strežnike. Enako velja tudi za ročno konfiguracijo, ki mora biti najprej opravljena na enem strežniku, ko tam preverjeno deluje, pa se jo izvede še na preostalih strežnikih.

Poglavje 4

Obvladovanje velikega števila uporabnikov

V prejšnjem poglavju smo naredili naš sistem za prodajo vstopnic visoko razpoložljiv. Poleg strojne redundance smo se poizkusili čimbolj izogniti tudi izpadom zaradi programskih nezdružljivosti, napak v izvorni kodi aplikacije in človeških napak pri konfiguraciji strežnikov ter aplikacije same. Z izjemo napovedanih izpadov bi morala biti aplikacija zdaj uporabnikom večino časa na razpolago. Izkaže se, če aplikacije ne onemogoči odpoved strojne opreme ali človeška napaka, jo lahko tudi uporabniki sami. V tem poglavju si bomo zato pogledali, kako lahko pri aplikaciji dosežemo večjo robustnost in odpornost na veliko količino hkratnih uporabnikov, ki jih naš sistem sicer ne bi zmogel obvladati.

Pri promociji dogodkov je postala zelo priljubljena praksa, da je prodaja vstopnic vnaprej napovedana za določen dan in uro. To še najbolj drži za zelo znane izvajalce, kot so Rolling Stones, U2, Robbie Williams ali za odmevne dogodke, kot je *Eurovizija*[29]. Ker pridejo uporabniki po vstopnice naenkrat, smo v takih primerih soočeni s količino uporabnikov, ki je strežniki ne uspejo obvladati.

Na produkcijskem okolju smo za največje prireditve v preteklosti uporabljali storitve podjetja Akamai[30]. To je podjetje, ki ima po svetu veliko število strežnikov in se ukvarja ravno z obvladovanjem velike količine uporabnikov in prometa. Za čas prodaje vstopnic smo domeno usmerili na njihove strežnike, pri čemer smo uporabljali čakalno vrsto za uporabnike in jih počasi spuščali na naše strežnike. Spremljajoč obremenitev smo količino uporabnikov povečevali in zmanjševali, dokler vstopnice niso bile prodane ali je zanimanje dovolj upadlo, da smo vse uporabnike obvladali sami. Storitev je bila draga,

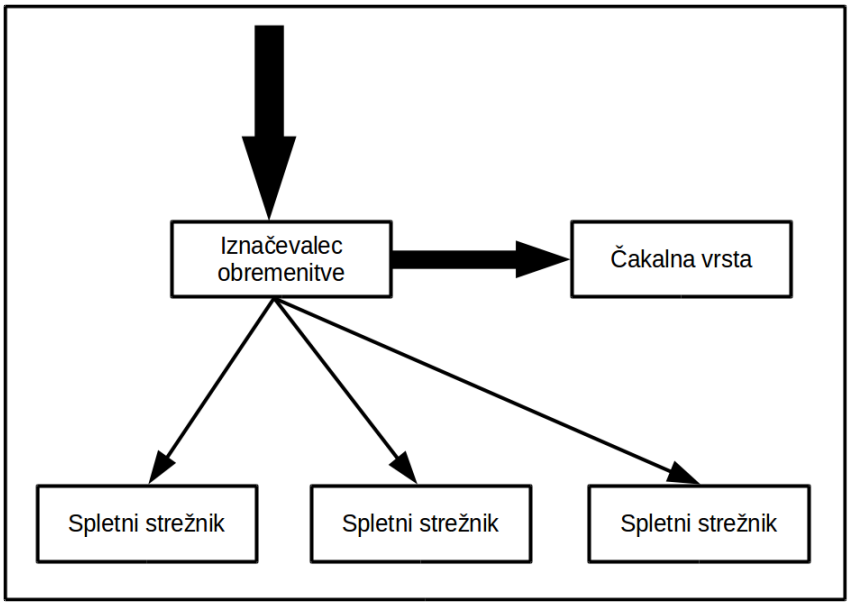
ocena števila uporabnikov pa težka, zato se je velikokrat zgodilo, da je bila prodaja napačno ocenjena in preusmeritve sploh ne bi potrebovali. Zaradi tega smo se odločili najti svojo rešitev, ki bi jo lahko uporabili za vse razen največjih dogodkov.

Po dolgem iskanju in testiranju smo rešitev našli v novem izenačevalcu obremenitve Haproxy[31], ki deluje na aplikacijskem nivoju. Prva bistvena razlika med programoma Keepalived in Haproxy je ta, da program Keepalived zgolj posreduje povezave, Haproxy deluje pa kot proksi (angl. proxy). To pomeni, da vsako uporabniško povezavo zaključi in do spletnega strežnika ustvari novo. Slednje nam daje možnost vpogleda in spreminjanja zahtev na spletne strežnike, kar je druga bistvena razlika med njima. Zaradi tega lahko napišemo dokaj kompleksna pravila obravnavanja zahtev, ki vključujejo vsa polja v glavi zahteve (angl. header) in nekatere parametre trenutne povezave. Seznam parametrov, ki jih Haproxy spremlja in hrani, je dolg, med bolj uporabnimi so pa naslednji:

- skupno število povezav na obličju (angl. frontend) in zaledju (angl. backend);
- število povezav na sekundo na obličju in zaledju;
- število prostih povezav in dolžina vrste na obličju in zaledju;
- število delujočih zalednih strežnikov;
- vse vrednosti iz paketov na TCP in IP nivoju (IP naslovi, vrata);
- vse vrednosti iz glave HTTP zahteve.

Slabost programa Haproxy je, da v verziji 1.4 ne podpira protokola https, zato bomo za https promet in postavitev naslovov IP še vedno uporabili program Keepalived, na Haproxy pa bomo prenesli le http. Tudi preklop na rezervni izenačevalec obremenitve bo še vedno nadzoroval Keepalived, ki bo spremljal tudi delovanje programa Haproxy. V primeru, da bo Haproxy odpovedal, bo ravno tako sprožil preklop na rezervni izenačevalec obremenitve.

Za delovanje moramo programu Haproxy nastaviti obličje, ki sprejema povezave in zaledje, na katerega so sprejete povezave posredovane. V konfiguraciji bomo nastavili, koliko povezav bo lahko hkrati odprtih na zaledje, preostale povezave bodo preusmerjene pa v čakalno vrsto (angl. waiting room, sorry server). Za čakalno vrsto bomo uporabili namenski strežnik, na katerem bo



Slika 4.1: Slika prikazuje preusmerjanje uporabnikov nad določeno mejo na strežnik za čakalno vrsto. Ko uporabniki uspešno zaključijo nakupni proces na spletnih strežnikih, se količina povezav na zaledju zmanjša in jih lahko nadomesti nekaj uporabnikov iz čakalne vrste. To se ponavlja toliko časa, dokler količina uporabnikov ne pade pod mejo, ko so lahko vsi usmerjeni na spletne strežnike.

strežnik Lighttpd vedno vračal kratko obvestilo, da je uporabnik v čakalni vrsti.

Konfiguracija za Haproxy je v datoteki */etc/haproxy/haproxy.conf*. Poglejmo nekaj osnovnih nastavitev:

```
global
        daemon
        user haproxy
        group haproxy
        log /dev/log local4
        pidfile /var/run/haproxy.pid
        stats socket /var/run/haproxy.stat mode 600
        stats maxconn 5
```

```
defaults
        log global
        option httplog
        option httpclose
        option forwardfor header Uporabniski-IP
        mode http
        balance leastconn
```

Nastavitve v globalnem delu (angl. global) niso povezane z obravnavanjem povezav, zato jih bomo spustili. V privzetem delu (angl. default) smo nastavili beleženje povezav. Določimo tudi, naj se povezave po odgovoru spletnega strežnika prekinejo. Ker vse povezave do spletnih strežnikov odpira izenačevalec obremenitve, so izvorni naslovi IP napačni, zato smo v glavo zahtevam na spletne strežnike dodali uporabniški naslov IP (Uporabniski-IP). Na spletnih strežnikih moramo v ta namen uporabiti modul *remoteip*[32], ki bo izvorni naslov dobil iz glave zahteve namesto iz polja v TCP paketu. Način dela smo nastavili na http (obstaja tudi tcp, ki deluje na četrtem nivoju). Algoritem za razporejanje povezav po zalednjih strežnikih smo nastavili na *leastconn*, kot pri programu Keepalived.

Za aplikacijo bomo najprej nastavili dve zaledji; strežnike, ki bodo prikazovali stran, in strežnik za čakalno vrsto:

```
backend trgovina
  option httpchk OPTIONS * HTTP/1.1\r\nHost:\ www
  server web1 192.168.1.10 check port 80 inter 5s maxconn 100
  server web2 192.168.1.11 check port 80 inter 5s maxconn 100

backend vrsta
  option httpchk GET / HTTP/1.1\r\nHost:\ www
  server vrsta 192.168.1.19 check port 80 maxconn 2000
```

Najprej smo nastavili način preverjanja, če zaledni strežniki delujejo, za kar smo uporabili ukaz OPTIONS. Za tem smo našteli vse zaledne strežnike. V našem primeru smo določili, da se bo delovanje strežnikov preverjalo vsakih pet sekund (angl. inter) na vratih 80. Zaledni strežnik bo označen kot nedelujoč, če bodo tri zaporedna preverjanja neuspešna, in kot delujoč, če bosta dve zaporedni preverjanji uspešni. Vrednosti lahko spremenimo z opcijama *fall* in *rise*. Obema strežnikoma smo omejili količino hkratnih povezav na 100. Ta nastavitev je mišljena le za obvarovanje strežnikov pred preveliko količino

povezav in posledično odpovedjo. Omejitev količine uporabnikov za čakalno vrsto bomo nastavili kasneje.

Sledi konfiguracija strežnika za čakalno vrsto. Strežnik Lighttpd bomo nastavili tako, da bo poslušal na istih vratih kot stran, vse zahteve bomo pa prepisali v korensko pot (/). To bomo storili zato, ker bodo uporabniki preusmerjeni v čakalno vrsto na različnih straneh, ki na tem strežniku ne bodo obstajale. Zaradi omenjenega potrebujemo mehanizem, ki bo za vsako zahtevo pokazal isti odgovor. V Lighttpd to naredimo tako:

```
$SERVER["socket"] == "0.0.0.0:80" {
  server.document-root = "/var/www/html"
  url.rewrite-once = (
    ".*" => "/"
  )
}
```

V /var/www/html/index.html bomo napisali tisto, kar hočemo, da bo videl v čakalno vrsto preusmerjen uporabnik. Za boljšo uporabniško izkušnjo lahko stran periodično avtomatično osvežujemo in dodamo tudi odštevanje do naslednjega osveževanja. Ob tem moramo poudariti, da mora biti ob osvežitvi uporabljena prejšnja zahteva, sicer uporabnik ne bo usmerjen na pravo stran.

Zdaj nastavimo še obličje:

```
frontend www.trgovina.tld
        bind 192.168.2.100:80
        maxconn 200
        acl poznan hdr_sub(Cookie) -i piskot
        acl zaseden fe_conn ge 50
        acl poln fe_conn ge 100
        use_backend vrsta if zaseden !poznan or poln
        default_backend trgovina
```

Zgornji omejitvi (*zaseden* in *poln*) sta nastavljeni začasno in sta odvisni od zmožnosti spletnih strežnikov ter zahtevnosti aplikacije. V našem primeru smo spodnjo vrednost nastavili na 50 in zgornjo na 100 sočasnih povezav. Omenimo, da vrednosti veljata za obličje, ne posamezen spletni strežnik.

Z našo implementacijo čakalne vrste smo želeli doseči, da bi uporabnik ob prihodu v nakupni proces tega tudi nemoteno zaključil. Če ima nakupni proces na primer pet korakov (izbira dogodka, izbira vstopnice, metoda plačila, metoda dostave, nakup), bi bila uporabniška izkušnja izredno slaba, če

bi se uporabnikom vmes prikazovala čakalna vrsta. Zaradi tega mora aplikacija vsakemu uporabniku nastaviti piškotek (angl. cookie) z imenom "piskot". Na ta način Haproxy spozna uporabnike, ki so že začeli z nakupnim procesom in jim daje prednost pred tistimi, ki še niso. Ko količina vzporednih povezav preseže spodnjo mejo (*zaseden*), začnemo uporabnike brez piškotka preusmerjati v čakalno vrsto. Uporabnikov, ki imajo nastavljen piškotek, ne preusmerjamo, dokler število povezav ne preseže zgornje vrednosti (*poln*). Ko se število vzporednih povezav bliža zgornji vrednosti, se količina novih uporabnikov, ki pridejo v sistem, zmanjšuje, s čimer se vire prepusti uporabnikom, ki z nakupom zaključujejo. Zgornjo mejo nastavimo zgolj z namenom preprečitve popolne odpovedi sistema ob preveliki obremenitvi, zato mora biti pametno nastavljena, da uporabnikov ne začne omejevati prekmalu, hkrati pa omogoča še nekaj rezerve za nepredvidene dogodke. Obe meji sta odvisni od veliko dejavnikov, zato ju je potrebno določiti empirično. Da uporabniki nemoteno zaključijo nakupni proces, smo dosegli tudi z omejevanjem zgolj nešifriranega prometa. Ko pride uporabnik na šifrirano stran, ne more biti več preusmerjen v čakalno vrsto.

Ob tem je treba povedati, da smo s samo enim strežnikom za čakalno vrsto prekršili pravilo, da mora biti vsaka naprava visoko razpoložljiva. Glavni razlog je, da strežnik za čakalno vrsto nima funkcije v povprečnem dnevu, temveč le med večjimi prodajami. Odpoved strežnika za čakalno vrsto v večini primerov torej nima negativnih posledic za sistem, verjetnost, da bo odpovedal ravno v času prodaje, je pa izredno majhna. Zaradi tega lahko za redundanco poskrbimo tako, da čakalno vrsto nastavimo na spletnih strežnikih in jo lahko uporabimo v nujnih primerih. Čeprav rešitev ni idealna, ker so med prodajami spletni strežniki že tako obremenjeni, je strežba majhne statične strani s strežnikom Lighttpd toliko varčnejša od interpretiranja strani PHP s strežnikom Apache, da velike razlike v obremenjenosti strežnikov ne bi smeli opaziti.

Poglavje 5

Ovrednotenje izboljšav sistema

5.1 Redundanca in zanesljivost sistema

Ker sta končna in produkcijska arhitektura enaki, bomo končno arhitekturo ovrednotili kar na podlagi testov, opravljenih na produkcijski arhitekturi.

Testi so pokazali, da je sistem resnično visoko razpoložljiv in veliko izpadov uporabnik sploh ne zazna. Največje odpovedi se sicer opazijo, vendar se same od sebe popravijo, preden dobi skrbnik prvo sporočilo SMS od nadzornega sistema.

Redundanco **požarnih zidov** smo testirali z odklopom aktivnega iz električnega napajanja. V tem primeru je rezervni požarni zid prevzel delo v roku ene sekunde. Pri sočasnem pinganju se je izgubil en ping. Brskanje po strani je bilo večinoma nemoteno, v izrednem primeru, ko je bila zahteva poslana ravno tisto sekundo, ko požarni zid še ni deloval, pa se je zahteva izvajala dlje, saj jo je moral brskalnik ponoviti.

Stikala smo testirali na enak način in ugotovili, da so bili tu izpadi daljši, vendar je bil preklop kljub temu večinoma opravljen v roku nekaj sekund. V primeru odpovedi stikala, ki skrbi za kopico, se ta čas podaljša na pol minute, saj morajo stikala znova glasovati, kdo bo postal glavni v kopici. Tu smo opazili neželen stranki učinek, in sicer da se lahko v tem času prekinejo notranje povezave, ki so potrebne za delovanje strežnikov, kot recimo NFS ali CIFS za datotečni strežnik ali iSCSI za virtualne strežnike. Zato je nujno časovne omejitve teh povezav prilagoditi največji dolžini izpada omrežja, kajti ne pomaga veliko, če omrežje v roku pol minute po izpadu ponovno deluje, če so odpovedali vsi virtualni strežniki, ki do svojih datotečnih sistemov dostopajo preko omrežja.

Odpoved ene veje **napajanja** bodisi zaradi izpada električne energije bodisi

odpovedi napajalnika ali ene **omrežne kartice** prav tako ne povzroči izpada. Brez enega napajanja strežnik nemoteno deluje. Tako kratke prekinitve, kot se zgodijo pri odklopu ene omrežne kartice, pa zakrijejo protokoli na višjih nivojih s ponovitvami. V praksi smo izmenično odklapljali kabla iz omrežnih kartic, a se niso prekinile niti obstoječe povezave na strežnik.

Po postavitvi grozda **spletnih strežnikov** izpada aplikacije zaradi strojne odpovedi spletnih strežnikov praktično ne more biti več. Za test smo izklopili posamezen spletni strežnik in tudi izenačevalec obremenitve. Povprečen čas, ko izenačevalec obremenitev zazna nedelujoč spletni strežnik in ga tako tudi označi, traja med 15 in 20 sekund. V tem času je $\frac{1}{n}$ (kjer je n število spletnih strežnikov) uporabnikov usmerjenih na nedelujoč strežnik, zato jim poteče časovna omejitev za zahtevo. Pri naslednji zahtevi je strežnik zelo verjetno že označen kot nedelujoč. Seveda to velja za zgornje nastavitve. Preverjanje spletnih strežnikov se da nastaviti bolj strogo in s tem še zmanjšati čase, ko nedelujoč strežnik dobiva zahteve. Preklop z glavnega na rezervni izenačevalec obremenitev je še hitrejši. Obvestila se pošiljajo vsako sekundo, tako da se v primeru izpada glavnega vsi naslovi IP in konfiguracija prenesejo na nadomestni strežnik v roku treh do petih sekund. Dodatna prednost grozda je enostavna razširljivost. Dodajanje novega spletnega strežnika v grozd je enostavna in časovno nepotratna naloga.

Z grozdom spletnih strežnikov smo dobili še eno izvrstno funkcionalnost, in sicer možnost nadgradnje spletnih strežnikov brez izpada. Če na izenačevalcu obremenitve spremenimo določenemu spletnemu strežniku utež na nič, bo nehal dobivati zahteve. Ko število aktivnih zahtev pade na nič, strežnik brez škode nadgradimo. Po ponovnem zagonu ga bosta izenačevalca obremenitev avtomatično zaznala in dodala v seznam delujočih strežnikov. Če smo pripravljeni na dva izpada po nekaj sekund, lahko nadgradimo tudi oba izenačevalca obremenitev.

Tudi grozd **podatkovih strežnikov** smo testirali z izklopom aktivnega iz napajanja. Pri naši konfiguraciji traja preklop baze na drugi strežnik približno pet sekund. V tem času baza ni dosegljiva. To pa ni nujno, da je ves izpad. Ugotovili smo namreč, da po preklopu na rezervni strežnik ostale naprave še vedno uporabljajo stari naslov MAC za plavajoč naslov IP. Dokler ta ne poteče, je baza, kljub temu delovanju za aplikacijo nedosegljiva. V našem primeru je to trajalo približno 30 sekund. Ena od možnih rešitev je nastavitev krajšega časa poteka tabel ARP na vseh omrežnih napravah, ob čemer hkrati povečamo količino prometa ARP po omrežju. Boljša rešitev bi bila pa objava novega MAC naslova na omrežju po preklopu na rezervni strežnik.

Za take grozde je zelo priporočljivo nastaviti mehanizem *STONITH* (Shoot

the other node in the head). Ta mehanizem zna nepravilno delujoč strežnik ali vir v celoti onemogočiti. Zgodi se namreč, da zaradi napake v komunikaciji oba strežnika predvidevata, da je drugi odpovedal in začneta oba delovati kot primarna. V tem primeru se začnejo na vsakega zapisovati drugi podatki, kar privede do nedosledne kopije podatkov. Za potrebe te naloge STONITH ni nujen, zato smo ga izpustili. Največje tveganje za stanje rezcepljene osebnosti je nezanesljiva povezava med samima strežnikoma. Ker smo arhitekturo že zasnovali s podvojenimi omrežnimi napravami in tudi podvojenimi omrežnimi povezavami med strežniki in omrežno opremo, je potreba po taki tehnologiji toliko manjša.

Kot pri spletnih strežnikih nam je grozd podatkovnih strežnikov poleg visoke razpoložljivosti omogočil tudi nadgranjo strežnikov z zelo malo izpada. Najprej nadgradimo nadomestni strežnik, nato z orodjem pcs bazo prestavimo nanj in nadgradimo še osnovnega. Drug izpad ni več potreben, saj je vseeno, na katerem teče baza. Razlog, zakaj bi jo hoteli imeti na konkretnem strežniku, je recimo slabši nadomestni strežnik. Nikjer ni rečeno, da morata biti strežnika enaka, zato lahko, s tem da se za čas izpada primarnega zadovoljimo z nižjo hitrostjo, nekaj privarčujemo pri strojni opremi.

Kot slabost te rešitve vidim predvsem ceno, cena instance se namreč podvoji. Če izberemo slabšo strojno opremo za nadomestni strežnik, lahko sicer nekaj prihranimo, vendar razlike ne morejo biti bistvene, ker tudi nadomestni strežnik ne sme biti bistveno slabši. Več lahko prihranimo z uporabo enega ali dveh nadomestnih strežnikov za več baz. Če za recimo štiri ločene baze uporabimo samo en nadomestni strežnik, instanco podražimo samo za četrtino, hkrati pa dobimo redundanco za vse baze. Če je to dovolj redundančno, je potrebno presoditi za vsako aplikacijo posebej.

Na **diskovnem sistemu** smo simulirali odpoved enega krmilnika, tako da smo enemu ukazali preklop vseh diskov na drugega, kar se zgodi tudi ob odpovedi enega krmilnika. Preklop je bil narejen hitreje kot v sekundi in ostali strežniki preklopa niso niti opazili, vse povezave do ostalih strežnikov pa so ostale odprte.

Razpoložljivost produkcijskega okolja merimo z zelo enostavnim orodjem *Pingdom*[35]. Orodju smo nastavili seznam spletnih strani, na katere se vsakih pet minut poveže in za vsako zabeleži, če je bila dosegljiva in v kolikšnem času. Na podlagi teh vrednosti izračuna tedensko in mesečno razpoložljivost. S produkcijsko arhitekturo, ki je nastavljena zelo podobno, kot smo opisali v tem delu, smo v letu 2015 dosegli 99,98-odstotno razpoložljivost, kar pomeni, da smo imeli v celem letu manj kot dve uri izpada. Kar se tiče hitrosti produkcijskega okolja pa lahko povemo, da lahko aplikacija razproda približno 16000

sedežev stadiona Stožice v manj kot desetih minutah.

5.2 Obvladovanje velike količine uporabnikov

S težavami pri obisku velikega števila uporabnikov v kratkem času smo se ukvarjali dolgo časa. Rešitev ni enostavna in vsaka ideja je imela svoje slabe strani. Uporabljena rešitev ima to prednost, da je zastonj in je hkrati drastično povečala količino uporabnikov, ki smo jo sposobni obvladati brez izpada. V istem času smo naredili tudi velik korak pri optimizaciji same aplikacije in hkrati povečali količino uporabnikov, ki jo je zmožna obvladati brez čakalne vrste. Rezultat je bil ta, da smo sposobni sami prodajati veliko večino dogodkov. Le za največje dogodke, v povprečju približno enega na leto, moramo najeti zunanje storitve.

Testiranje čakalne vrste je zahtevno, ker je težko simulirati veliko število uporabnikov. Za naše potrebe smo uporabili osem virtualnih strežnikov z orodjem *Jmeter*[33]. Jmeter je orodje, ki ga sprogramiramo tako, da namesto nas odpira strani, v polja vnaša vrednosti in jih potrjuje. Tako smo naredili program, ki gre skozi nakupni proces in simulira enega uporabnika. Na vsakem virtualnem strežniku smo pognali 50 hkratnih uporabnikov, na izenačevalcu obremenitve pa nastavili dovolj nizke vrednosti, da so uporabniki dobivali tudi čakalno vrsto. Potrebno se je zavedati, da so taki umetni testi zgolj približek realnega uporabnika, zato so tudi rezultati približni. Analiza dnevniških datotek je pokazala, da uporabniki počnejo včasih zelo nerazumljive stvari, kot so večkratno dodajanje ter odstranjevanje vstopnic iz košarice, skakanje na osnovno stran in nazaj na košarico ali zapiranje brskalnika sredi nakupnega procesa. Posledično je tudi število povezav na strežnike in njihova obremenitev pri realnih uporabnikih drugačna kot pri umetnih testih. Istočasno smo spremljali tudi obremenitev strežnikov. Za spremljanje količine uporabnikov na strani smo uporabili Google Analytics[34]. Z obremenitvenimi testi smo se seznanili z osnovnimi zmožnosti sistema, bolje pa smo ga spoznali pri sami prodaji vstopnic znanih izvajalcev. Izkazalo se je, da do približno 6000 sočasnih uporabnikov ne potrebujemo čakalne vrste, pri več uporabnikih pa obremenitev spletnih strežnikov že toliko naraste, da strežba brez omejitev ni več varna. Zgornje količine uporabnikov, ki jo zmore obvladati sistem, ne poznamo, saj tolikšne količine uporabnikov še nismo bili deležni.

Prvotno je bila sposobnost obladovanja velike količine uporabnikov pomembna za obstoj sistema, z uporabo čakalne vrste pa postane količina uporabnikov, ki so jo strežniki zmožni obdelati, manj pomembna. Ker z upo-

rabo čakalne vrste na spletne strežnike spustimo le toliko uporabnikov, kot jih zmorejo obdelati, je hitrost prodajanja vstopnic pomembna samo še za dobro uporabniško izkušnjo.

Čeprav je glavni pokazatelj obremenitve sistema število sočasnih povezav na strežnike (ki je tudi glavni kriterij za preusmeritev uporabnikov v čakalno vrsto), pa pride do zanimive konverzije virov in je za končno število uporabnikov, ki smo jih sposobni sočasno obvladati, v resnici pomembno število zahtev, ki jih strežnik za čakalno vrsto opravi na časovno enoto. V resnici noben spletni strežnik ne zmore toliko sočasnih povezav. Gre za to, da strežnik za čakalno vrsto opravi nekaj tisoč povezav na sekundo in ker uporabniki ne pošiljajo zahtev istočasno, je večina strani vrnjenih, še preden jo zahteva naslednji uporabnik. Kljub temu, da je morda aplikacijo obiskalo tisoč uporabnikov, je lahko hkrati na strežnik za čakalno vrsto odprtih samo petdeset povezav. Če upoštevamo tudi, da uporabniki ne klikajo vsako sekundo (čeprav velja, da so uporabniki v čakalni vrsti bolj neučakani kot tisti v nakupnem procesu), se sposobnost sistema še poveča. Zato je pomembno, da stran za čakalno vrsto zasede malo prostora in je čim hitreje poslana uporabniku. V nasprotnem primeru vire, ki jih lahko kompenziramo s časom (na primer procesorsko moč, kjer strežnik še vedno opravi svoje, vendar malenkost kasneje), nadomeščamo z viri, ki so absolutno omenjeni (na primer količina spomina).

Ob tem je potrebno poudariti, da je z uporabo čakalne vrste povečanje sposobnosti sistema zgolj navidezno. Konverzija sistema je še vedno enaka, saj ne moremo prodati več vstopnic, kot je tega sposoben sistem. Razlika se pokaže pri tistih uporabnikih, ki so padli v čakalno vrsto. Ti niso naredili ničesar uporabnega, lahko pa predstavljajo večino. Razlike v količini uporabnikov, ki jih je sistem sposoben obdelati, so v resnici velike. Povečanje števila uporabnikov za faktor deset tu s hitrim strežnikom za čakalno vrsto ni težko. Oglaševalcem bi se ob taki vrednosti verjetno zasvetile oči, kot sem pa že omenil, se je pa potrebno zavedati ozadja in uporabnosti take "pohitritve".

Med testiranjem smo ugotovili, da se lahko v daljšem obdobju poveča količina uporabnikov s piškotkom, zaradi katerih naraste število sočasnih povezav na spletne strežnike in začne izenačevalec obremenitve tudi te uporabnike preusmerjati v čakalno vrsto. Kljub temu, da omejujemo uporabnike brez piškotka, število povezav niha v odvisnosti od aktivnosti uporabnikov. Čeprav jih je lahko v splošnem že dovolj, se na vsake toliko najde trenutek, ko je število povezav pod mejo omejevanja in takrat bo nov uporabnik spuščen na stran. V daljšem časovnem obdobju se lahko teh uporabnikov nabere več kot bi želeli. Glavni razlog so po navadi tehnične težave, zaradi katerih uporabniki ne morejo zaključiti nakupnega procesa. S povečevanjem števila uporabnikov

s piškotkom se težava manjša, zato se do neke mere popravlja sama. Dokler lahko uporabniki uspešno zaključujejo nakupni proces in prodaja ne traja zelo dolgo, je težava zanemarljiva. V izrednem primeru, ko vse odpove, je potrebno po odpravi težav zamenjati ime piškotka in uporabnike znova spustiti v nakupni proces.

Poglavje 6

Sklepne ugotovitve

V prvem poglavju smo predstavili teorijo razpoložljivost in ugotovili, da so visoko razpoložljive storitve dandanes nuja. Nerazpoložljivost spletnih storitev podjetja veliko stane, zato morajo biti storitve uporabnikom neprestano na voljo. Raziskave kažejo, da je pridobitev nove stranke približno desetkrat dražja od ohranitve obstoječe[1].

V drugem poglavju smo predstavili osnovne komponente sistema za prodajo vstopnic in namestili osnovno arhitekturo. Čeprav smo na njej že lahko prodajali vstopnice, je bila njena največja pomanjkljivost nezanesljivost, saj bi vsaka strojna ali programska odpoved pomenila izpad celotne aplikacije. Zanesljivost posameznih komponent in pričakovana razpoložljivost aplikacije sta bili zelo slabi.

Zaradi navedenega smo v tretjem poglavju preučili možnosti povečanja razpoložljivosti naše osnovne arhitekture. Mrežno opremo smo podvojili in nastavili tako, da v primeru izpada ene naprave, druga avtomatično prevzame njeno delo. Spletni strežnik smo nadomestili z grozdom spletnih strežnikov in dvema izenačevalcema obremenitve. Pri odpovedi prvega bo drugi prevzel njegovo delo. V primeru odpovedi spletnega strežnika bo ta izločen iz grozda, na zahteve uporabnikov pa bodo odgovarjali preostali spletni strežniki. Tudi pri odpovedi prvega podatkovnega strežnika ali krmilnika v diskovnem sistemu bo delo samodejno prevzel drugi. Vsi izpadi se zdaj rešijo samodejno in potrebe po ročnem reševanju težav ni več, kar prihrani večino časa. Povprečen čas izpada se meri v sekundah, kar pomeni, da bo večina izpadov rešena, preden bi jih zaznali uporabniki. S temi koraki smo razpoložljivost aplikacije znatno povečali. Poleg strojnih odpovedi smo se dotaknili tudi človeških napak in nevarnosti pri konfiguraciji več strežnikov istočasno ter ugotovili, da predstavljajo znatno količino razlogov za izpade in da jih je potrebno obravnavati z

enako proriteto kot strojno zanesljivost.

V četrtem poglavju smo sistem izboljšali še na področju obvladovanja velikega števila uporabnikov. Tudi visoko razpoložljiva arhitektura bo odpovedala, če pride na stran preveliko število uporabnikov. Zaradi tega smo sistemu dodali čakalno vrsto, kamor preusmerjamo uporabnike, če jih je več od nastavljene omejitve. Na ta način smo aplikacijo rešili pred popolno odpovedjo v primeru velikega števila uporabnikov, hkrati pa uporabnikom omogočili lepšo izkušnjo pri nakupu vstopnic.

Slike

Literatura

[1] Evan Markus, Hal Stern, "Blueprints for High Availability, Second Edition" Wiley Publishing, Inc., poglavje 2.

[2] Piranha, "http://www.linuxvirtualserver.org/docs/ha/piranha.html", dostopano 4. 4. 2016.

[3] The Apache Software Foundation, "http://www.apache.org/", dostopano 4. 4. 2016.

[4] TCP Wrappers, "https://en.wikipedia.org/wiki/TCP_Wrapper", dostopano 4. 4. 2016.

[5] Nagios, "https://www.nagios.org/", dostopano 4. 4. 2016.

[6] Linux CentOS, "https://www.centos.org/", dostopano 4. 4. 2016.

[7] Linux Ubuntu, "http://www.ubuntu.com/", dostopano 4. 4. 2016.

[8] Php, "http://www.php.net/", dostopano 4. 4. 2016.

[9] NFS, "http://nfs.sourceforge.net/", dostopano 4. 4. 2016.

[10] LVM, "https://en.wikipedia.org/wiki/Logical_Volume_Manager_(Linux)", dostopano 4. 4. 2016.

[11] Lighttpd, "https://www.lighttpd.net/", dostopano 4. 4. 2016.

[12] SMB, "https://en.wikipedia.org/wiki/Server_Message_Block", dostopano 4. 4. 2016.

[13] Postgresql, "http://www.postgresql.org/", dostopano 4. 4. 2016.

[14] Certificate Authority (CA), "https://en.wikipedia.org/wiki/Certificate_authority", dostopano 4. 4. 2016.

[15] Privacy Enhanced Mail (PEM), "https://en.wikipedia.org/wiki/Privacy-enhanced_Electronic_Mail", dostopano 4. 4. 2016.

[16] Nginx, "http://nginx.org/", dostopano 4. 4. 2016.

[17] PHP-FPM, "http://php-fpm.org/", dostopano 4. 4. 2016.

[18] Secure Shell (SSH), "http://www.openssh.com/", dostopano 4. 4. 2016.

[19] Prevajalnik omrežnih naslovov (angl. Network Address Translation), "https://en.wikipedia.org/wiki/Network_address_translation", dostopano 4. 4. 2016.

[20] Cisco ASA, "http://www.cisco.com/c/en/us/products/security/adaptive-security-appliance-asa-software/index.html", dostopano 4. 4. 2016.

[21] Keepalived, "http://www.keepalived.org", dostopano 4. 4. 2016.

[22] Microsoft Windows, "(http://www.microsoft.com/", dostopano 17. 4. 2016.

[23] Samba, "https://www.samba.org/", dostopano 4. 4. 2016.

[24] Pgpool, "http://www.pgpool.net/", dostopano 4. 4. 2016.

[25] Distributed Replicated Block Device, "http://drbd.linbit.com/", dostopano 4. 4. 2016.

[26] Corosync, "http://corosync.github.io/corosync", dostopano 4. 4. 2016.

[27] Pacemaker, "http://clusterlabs.org", dostopano 4. 4. 2016.

[28] Drbdadm, "https://www.drbd.org/en/doc/users-guide-83/re-drbdadm", dostopano 4. 4. 2016.

[29] Eurovision Song Contest, "http://www.eurovision.tv/", dostopano 4. 4. 2016.

[30] Akamai, "https://www.akamai.com/", dostopano 4. 4. 2016.

[31] Haproxy, "http://www.haproxy.org/", dostopano 4. 4. 2016.

[32] Remoteip, "https://httpd.apache.org/docs/trunk/mod/mod_remoteip.html", dostopano 4. 4. 2016.

[33] Jmeter, “http://jmeter.apache.org/”, dostopano 14. 4. 2016.

[34] Google Analytics, “http://www.google.com/analytics/”, dostopano 14. 4. 2016.

[35] Pingdom, “https://www.pingdom.com/”, dostopano 16. 4. 2016.