UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Žan Kafol

VISOKO RAZPOLOŽLJIV SISTEM GEOGRAFSKO RAZPRŠENIH SPLETNIH STREŽNIKOV

DIPLOMSKO DELO

NA VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

Mentor: doc. dr. Rok Rupnik

Ljubljana, 2011

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00111/2011
Datum: 05.04.2011

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ŽAN KAFOL**

Naslov: **VISOKO RAZPOLOŽLJIV SISTEM GEOGRAFSKO RAZPRŠENIH SPLETNIH STREŽNIKOV**

HIGHLY AVAILABLE GEOGRAPHICALLY DISTRIBUTED WEB SERVERS SYSTEM

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Zasnujte in izdelajte načrt za sistem spletnih strežnikov, ki bodo postavljeni na geografsko ločenih lokacijah. Pri tem upoštevajte omejitev, da je na posamezni lokaciji le en strežnik in ne lokalna skupina strežnikov. Sistem naj omogoča, da se v primeru izpada enega izmed strežnikov v sistemu vsi uporabniki povežejo na kateregakoli drugi dosegljiv strežnik brez izgube globalne seje sistema.

Mentor:
doc. dr. Rok Rupnik

Fakulteta za računalništvo in informatiko
UNIVERZA V LJUBLJANI
2

Dekan:
prof. dr. Nikolaj Zimic

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani **Žan Kafol**,
z vpisno številko **63070040**,

sem avtor diplomskega dela z naslovom:

**VISOKO RAZPOLOŽLJIV SISTEM
GEOGRAFSKO RAZPRŠENIH SPLETNIH STREŽNIKOV**

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom
  doc. dr. Roka Rupnika

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 5. 7. 2011 Podpis avtorja: _____

Zahvala

Za pomoč pri izdelavi diplomske naloge se zahvaljujem mentorju doc. dr. Roku Rupniku.

Zahvaljujem se mami Nadji, očetu Dušanu in sestri Tjaši za lektoriranje ter spodbudo in pomoč v času študija.

Zahvaljujem se tudi prijatelju Andreju za pomoč pri testiranju, Sebastjanu in Alešu za vse predloge in Ani za vso pomoč, podporo in dobro voljo.

KAZALO

POVZETEK

Cilj diplomskega dela je predstaviti izdelavo sistema visoke dosegljivosti strežnikov (v nadaljevanju HA, angl. *High Availability*), ki so postavljeni na geografsko ločenih lokacijah.

Postopki opisani v diplomskem delu prikazujejo vzpostavitev sistema v nizkem cenovnem rangu in lastnim gostovanjem strežnikov.

Zaradi ekonomskih in prostorskih razlogov je na posamezni lokaciji samo en strežnik in ne lokalna skupina strežnikov (angl. *local cluster*). Zaradi nepredvidljivih okoliščin (relativno pogosti izpadi električnega napajanja ali internetne povezave, hitro pokvarljiva strojna oprema), se v diplomskem delu osredotočam na okrevanje po katastrofi (angl. *Disaster Recovery*) strežnika na posamezni lokaciji in doseganje HA strežnikov.

Izdelani sistem omogoča transparenco, tj. ob pravilnem delovanju vsi povezani strežniki delujejo kot eden, ter distribucijo podatkov med ostalimi povezanimi strežniki. V primeru izpada enega izmed strežnikov v sistemu se njegovi uporabniki povežejo na kateregakoli drugega dosegljivega, brez izgube globalne seje sistema.

Ključne besede

Okrevanje po katastrofi, visoka dosegljivost, MySQL, PHP, memcache

ABSTRACT

This diploma paper describes the making of a high availability (HA) system of servers, which are set up on geographically separated locations.

Methods described in this paper are suitable for low-price configurations and home server hosting.

Due to financial and economical reasons only one server is hosted on each location and not a local cluster of servers. Home server hosting configurations have a higher probability of power supply failures, internet connectivity failures and hardware failures. For this reason my work is focusing on disaster recovery and achieving high availability of servers.

The designed HA system is transparent to the user, which means that in normal circumstances all of the connected servers appear to be working as one logical system, distributing their data across all servers. In case of failure, clients connected to the failing server are reconnected to a working server without losing their global session on the system.

Keywords

Disaster Recovery, High Availabilty, MySQL, PHP, memcache

1. UVOD

V okviru diplomske naloge sta bila v HA sistemu postavljena dva strežnika s podobno strojno in programsko opremo. Strežnika se nahajata na geografsko ločenih točkah, A (Andrej) in Ž (Žan).

Na obeh strežnikih tekoče aplikacije komunicirajo med seboj preko prostranega omrežja (v nadaljevanju WAN, angl. *Wide Area Network*). Strežnika med seboj replicirata podatke iz podatkovne baze, uporabniške seje ter predpomnilnika. Cilj replikacije je razdeljeno branje podatkov, neprestana pripravljenost na okrevanje po katastrofi in nadaljevanje uporabniške seje ob izpadu, kot je vidna iz celotnega sistema strežnikov.

HA sistem je bil testiran z lastno izdelano spletno aplikacijo ob normalnem delovanju, popolnem izpadu enega izmed strežnikov (izguba WAN povezave, izpad električne energije) ter izgubi povezave med strežnikoma. Ocenjena je bila stabilnost sistema kot celota in hitrost okrevanja sistema ob izpadu strežnika.

1.1. Uporabljene tehnologije

Posamezni strežnik v HA sistemu ima dva SATA diska enake velikosti, povezana v RAID-1 polje s pomočjo *FakeRAID* kontrolerja. Oba strežnika sta priključena na UPS, kar zagotavlja manjšo verjetnost izpada strežnikov ob izpadu električne energije.

Strežnik je postavljen na LAMP platformi (*Linux, Apache, MySQL, PHP*). Za operacijski sistem *Linux* je izbrana distribucija *Ubuntu*. Za predpomnjenje podatkov se uporablja *repcache*, ki je predelan *memcache* z možnostjo replikacije.

Spletna aplikacija, uporabljena za testiranje HA sistema je napisana v programskih jezikih *Flash* / *ActionScript*, *JavaScript*, PHP, HTML in CSS.

1.2. Opis problema

Z željo po lastnem gostovanju strežnikov, sem v okviru diplomskega dela raziskoval področja HA sistemov ter okrevanja po katastrofi. V današnjem času je raznovrstna strojna oprema ekonomsko lažje dostopna, zato je relativno enostavno postaviti strežnik in spletno aplikacijo z lastnim gostovanjem.

Podjetja, ki nudijo profesionalno gostovanje, imajo strežnike z namensko strojno opremo. V večini primerov so ti strežniki postavljeni v lokalno skupino zaradi porazdelitve obremenitve ter nadomestnega načina delovanja (angl. *failover*) ob izpadu strežnika. Za gostovanje

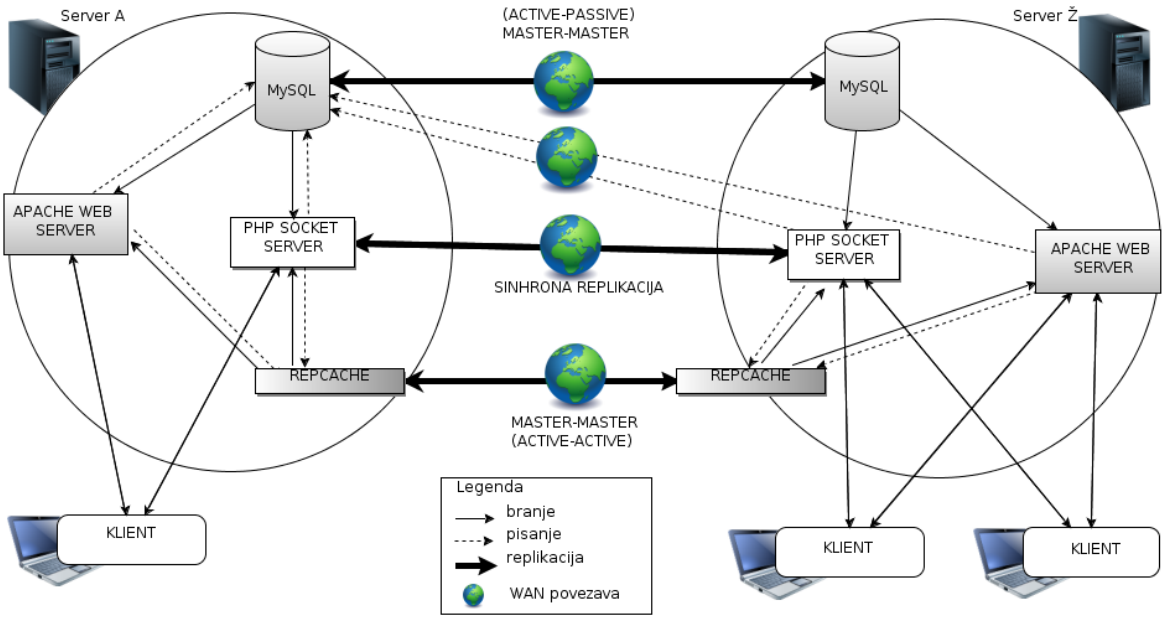
strežnikov so pripravljena posebna okolja, da podjetja zagotavljajo maksimalno varnost in kvaliteto storitve. Stroški vzdrževanja takšnih sistemov so izjemno dragi, zato se za najeto gostovanje odločijo podjetja ali posamezniki, ki potrebujejo zanesljivo in stabilno rešitev.

V času srednješolskega in študijskega izobraževanja sem izdeloval različne spletne aplikacije, ki so se stregle na lastnem strežniku. Lastni strežnik sem večinoma uporabljal za testiranje in razvijanje aplikacij. Iz finančnega vidika se mi gostovanja ne bi izplačalo najeti. Sčasoma je z naraščanjem števila uporabnikov nekaterih aplikacij prišla želja po visoki dosegljivosti strežnika, kajti z gostovanjem strežnika v neidealnem okolju in neprofesionalno strojno opremo je prihajalo do relativno pogostih izpadov, opaženi najpogostejši razlogi za to pa so bili izpadi WAN povezave, izpadi električne energije ter okvare strojne opreme.

Lokalna skupina je primerna za porazdelitev obremenitve in nadomestni način delovanja, vendar postane ob izgubi WAN povezave ali električnega napajanja celotna storitev nedosegljiva. Zaradi pričakovane večje verjetnosti takšnih izpadov v tem primeru takšna konfiguracija ni primerna. Pomožni strežniki morajo imeti drugi vir napajanja ter drugega ponudnika internetnih storitev, zato da so strežniki v medsebojnem delovanju čim bolj neodvisni od okoliščin. Vzdrževanje lokalne skupine zahteva večjo porabo energije ter prostora, zato tudi zaradi ekonomskih in finančnih razlogov ni bila postavljena lokalna skupina strežnikov na posamezni lokaciji.

Torej, z iskanjem rešitve po visoki dosegljivosti in nadomestnem načinu delovanja ob izpadu strežnika sem začel razmišljati o geografsko ločenih strežnikih, ki delujejo pod popolnoma neodvisnimi okoliščinami, zato da lahko sistem kot celota okreva po katastrofi (angl. *Disaster Recovery*). Kot katastrofa se označuje popolni izpad sistema na posamezni lokaciji.

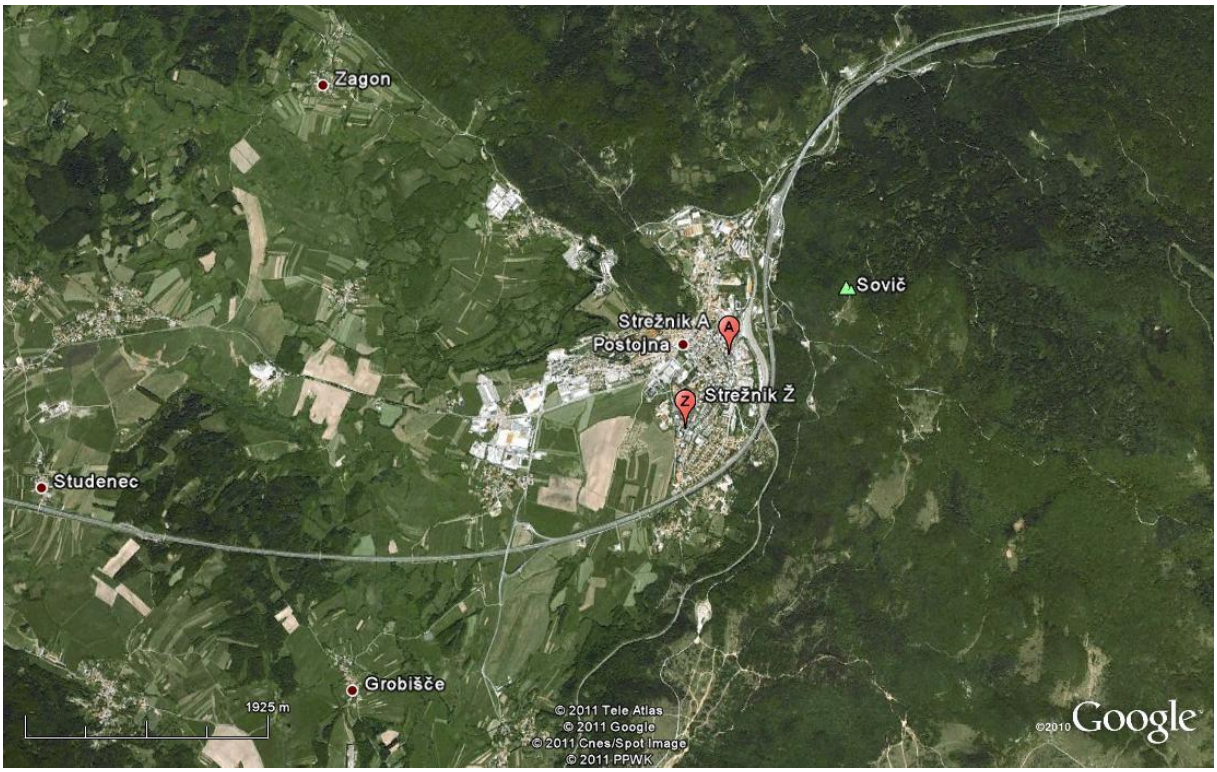
Izdelana rešitev vsebuje dva strežnika na geografsko ločenih lokacijah, ki med seboj asinhrono replicirata podatkovno bazo ter predpomnilnik. Strežnik Ž je povezan v optično omrežje, ki ga omogoča ponudnik internetnih storitev SiOL, strežnik A pa v vDSL omrežje (*very-high-bitrate Digital Subscriber Line*) pri ponudniku T-2. Na posameznem strežniku teče spletna aplikacija na spletnem strežniku *Apache* ter vtičnem strežniku, sprogramiranem v PHP. Vtični strežnik in spletni strežnik shranjujeta podatke v lokalni predpomnilnik, katerega vsebina se replicira na drugi strežnik. Podatke bereta iz lokalne podatkovne baze, pišeta pa samo v eno, zaradi ohranjanja konsistence podatkov, ki se lahko izgubi, če se pri asinhroni replikaciji podatki pišejo v različne strežnike. Vtična strežnika na posameznem strežniku sta med seboj sinhrono povezana in replicirata podatke ter uporabniško sejo.



Slika 1: Shema izdelanega sistema strežnikov in povezav med klienti in aplikacijami.

Ko uporabnik želi uporabljati spletno aplikacijo, njegov klient naredi poizvedbo na domenski strežnik, ki mu vrne dva naslova v naključnem vrstnem redu [21]. Tako se klient poveže na naključni strežnik. Sistem se obnaša kot eden, če sta oba strežnika delujoča. Ob izpadu, se uporabniku prekine povezava in uporabniški klient se takoj poskusi povezati na drugi strežnik. Ker je uporabniška seja replicirana, se lahko seja nadaljuje brez ponastavitve. Aplikacije, ki se povezujejo na podatkovni strežnik, lociran na drugem naslovu, tudi zaznajo izpad in začnejo uporabljati lokalni podatkovni strežnik. Zaradi replikacije podatkovnih baz, bo izpadli strežnik prejel podatke takoj, ko bo spet dosegljiv.

S takšnim modelom sistema sem se želel čim bolj približati stabilnosti in robustnosti profesionalnih rešitev.



Slika 2: Lokacije strežnikov A in Ž na zemljevidu.

1.3. Možni zapleti

Programski jezik PHP nima niti (angl. *threads*), vendar ker vtični strežnik ne izvaja potratnih računskih operacij in ker je vtičnik konfiguriran tako, da ne blokira izvajanja programa, dokler ne prejme podatkov za branje, niti za posameznega uporabnika niso potrebne.

Ker je model zasnovan tako, da lahko pride do primera, ko vsaka aplikacija piše v svoj podatkovni strežnik, je velika verjetnost nekonsistentnosti podatkov (angl. *split-brain*), če se podatki z obeh strežnikov hkrati pišejo na isto mesto. Problem moramo reševati na aplikacijskem nivoju, da aplikacije same operirajo na tak način, da minimizirajo možnosti napak.

Problemi lahko nastanejo tudi pri sinhroni komunikaciji med vtičnima strežnikoma zaradi latence. Zaradi obremenjene WAN povezave ali drugih motenj v povezljivosti je lahko vtični strežnik pogosto neodziven. Med strežnikoma je potrebno minimizirati frekvenco pošiljanja in količino poslanih podatkov.

2. PREDPRIPRAVA STREŽNIKOV

2.1. Izbira in konfiguracija strojne opreme

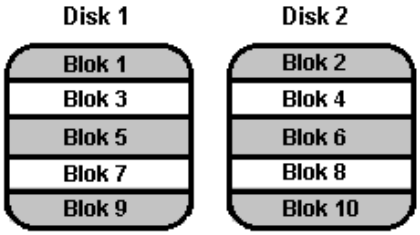
Minimalne strojne zahteve strežnika so odvisne od pričakovane obremenitve. Ocenjena minimalna velikost trdega diska je 100 GB, velikost pomnilnika 2 GB ter minimalna hitrost procesorja 1 GHz z enim jedrom. Oba strežnika in modema sta priklopljena na brezprekinitveno napajanje (UPS, angl. *Uninterrupted Power Supply*) s proti-napetostno zaščito.

Vsak strežnik ima 2 diska enake velikosti, ki sestavljata RAID-1 polje. RAID je angleška kratica za *Redundant Array of Inexpensive Disks* in omogoča povezovanje več diskov v en logični disk [1].

Možnih konfiguracij RAID polja je več, dosežemo pa lahko identično kopijo diskov, da se zavarujemo pred izgubo podatkov. Namen RAID polj je tudi razdelitev vhodno-izhodnih operacij po posameznih enotah v polju, s tem pa dosegamo hitrejše branje in pisanje diska ter večjo velikost diska z združitvijo diskov v polju v eno logično enoto.

2.1.1. Konfiguracija RAID polja

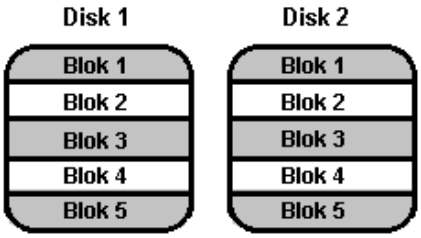
Za postavitev RAID polja potrebujemo minimalno 2 diska, ta pa lahko povežemo v RAID-0 ali RAID-1 polje. RAID-0 omogoča zaporedno pisanje podatkovnih blokov po posameznih diskih (angl. *striping*), zato lahko dosega visoke hitrosti pri pisanju podatkov. Tak sistem pa ne omogoča redundance in se ob izgubi ene enote v polju izgubi celotno polje.



Slika 3: Shema RAID-0 polja za dve enoti.

Prirejeno po viru: 1.

RAID-1 zapiše ob vsakem zapisu podatkovnega bloka isti blok še na redundančni disk (angl. *mirroring*), zato se čas pisanja poveča, ob izgubi ene enote v polju pa redundančni disk prevzame zahteve branja in pisanja [2].



Slika 4: Shrema RAID-1 polja za dve enoti.

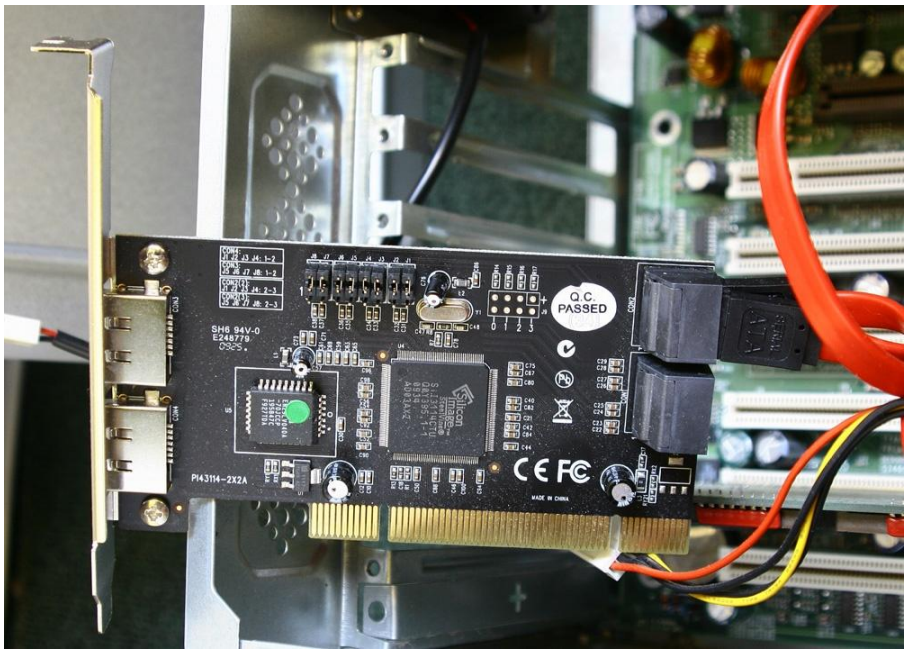
Prirejeno po viru: 2.

RAID polja stopnje 3, 4, 5 in višje uporabljajo kombinacije tehnik *striping* z različnimi preverjanji paritet ter s takimi konfiguracijami omogočajo redundanco in razdeljene vzhodno-izhodne operacije hkrati.

Zaradi poudarka na redundanci in nižjih stroških je RAID-1 z dvema diskoma izbrana konfiguracija za posamezni strežnik.

Implementacija RAID polja je lahko strojna (angl. *Hardware-RAID*), programska (angl. *Software-RAID*) ali programsko-strojna (angl. *HostRAID*, tudi *FakeRAID*) [3]. Strojni RAID sestavlja posebni kontroler, ki je neodvisen od sistema in ima svojo centralno procesorsko enoto (v nadaljevanju CPU, angl. *Central Processing Unit*) [3] in pomnilnik. Zato je strojni RAID hitrejši [3] in bolj zanesljiv, vendar so ti sistemi dražji. Operacijski sistem ali programska oprema v operacijskem sistemu upravlja z programskim RAID-om [3]. Operacijski sistem zazna vse diske posebej, z njimi pa izvaja operacije RAID polja na programskem nivoju. Programska oprema izvaja dvojno pisanje ali razdeljeno pisanje in branje.

Izbrana implementacija RAID polja na posameznem strežniku je *FakeRAID*. To je kontroler, ki je priklopljen na PCI vodilo na matični plošči. Na kontroler se priklopijo diski preko IDE (*Integrated Drive Electronics*) ali SATA (*Serial Advanced Technology Attachment*) vodila. Prednost te implementacije v tem primeru je nizka cena kontrolerja ter enostavnost. Čip na kontrolerju računa RAID operacije na strežnikovem CPU, izvaja jih pa kontroler, tako da operacijski sistem preko kontrolerja zazna RAID polje kot eno logično enoto.



Slika 5: FakeRAID kontroler na strežniku Ž s SATA priključki.

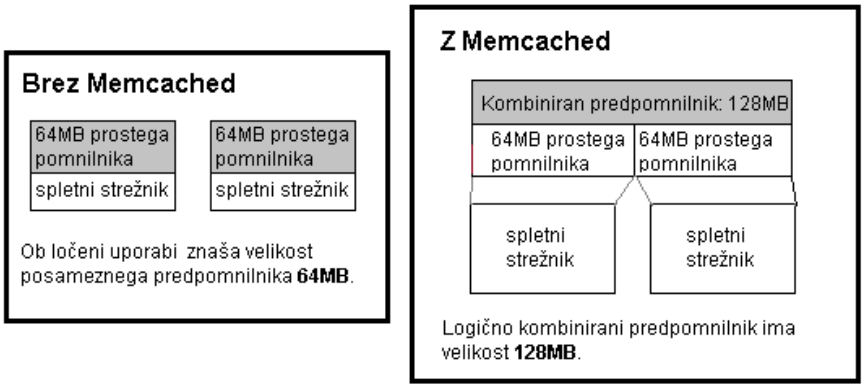
Avtor fotografije: Žan Kafol.

2.2. Namestitev programske opreme in konfiguracija

Namestitev operacijskega sistema Linux, distribucije Ubuntu je enostavna. Iz uradne spletne strani Ubuntu [24] je možno prenesti slikovno datoteko, ki jo nato zapišemo na CD ali USB ključ. Ubuntu verzije od 10.04 naprej same prepoznajo RAID naprave [3], tako da med instalacijo samo izberemo želeno konfiguracijo.

Privzeta namestitev distribucije Ubuntu je že opremljena z LAMP platformo, potrebno je namestiti še *memcached* in *repcached*.

Memcached je programska oprema, ki omogoča shrambo podatkov neposredno v pomnilnik [4] za namene predpomnjenja. Primerno je za shranjevanje dinamičnih podatkov, ki se sicer večkrat brez potrebe izračunavajo. S primernim predpomnjenjem podatkov se v veliki meri zmanjša obremenitev strežnika, predpomnjenje neposredno v pomnilnik pa še dodatno skrajša čas izvajanja aplikacij. Podatki v *Memcached* so v razpršeni tabeli, aplikacija pa se lahko poveže na več *memcached* strežnikov za dostop in shrambo podatkov. V primeru dveh ali več *memcached* strežnikov se podatki razdeljeno shranjujejo po posameznih strežnikih, zato si lahko *memcached* z dvema ali več strežniki predstavljamo kot RAID-0 polje. Redundance ni, lahko pa sestavimo eno logično enoto s kombinirano velikostjo.



Slika 6: Shema predpomnilnika brez in z *memcached*.

Prirejeno po viru: 6.

Za potrebe redundance je boljša rešitev *repcached*, ki je osnovan na *memcached* in omogoča repliciranje podatkov med strežniki [5]. Repliciranje podatkov lahko nastavimo po modelu *master-slave*, pri katerem *master* strežniki izvajajo klientove ukaze pisanja in branja podatkov, *slave* strežniki pa samo ukaze branja, ukaze pisanja pa prejemajo od *master* strežnikov.

Replikacija *master-master* je lahko aktivno-pasivna (angl. *active-passive*) ali aktivno-aktivna (angl. *active-active*). Pri aktivno-pasivni replikaciji je načelo takšno, da aplikacija lahko podatke bere iz katerega koli strežnika (aktivnega ali pasivnega), piše pa v samo aktivni strežnik. Pri aktivno-aktivni replikaciji se lahko bere in piše v katerikoli strežnik. Tak princip se uporablja pri vseh vrstah replikacije in o teh možnostih je govora tudi pri *MySQL* replikaciji.

Replikacija *master-slave* je vedno aktivno-pasivna, drugače lahko pride do nekonsistentnih podatkov.

V tem primeru z dvema strežnikoma, s poudarkom na redundanci in visoki dosegljivosti je primerna izbira *master-master* replikacija *repcached* strežnikov. Vsak zapis se izvede še na drugem strežniku, podatki so na obeh *repcached* strežnikih enaki.

Ker podatki v predpomnilniku niso trajni, smo lahko tolerantni do nekonsistentnih podatkov med dvema *repcached* strežnikoma. Če podatkov na nekem strežniku ni, ker do replikacije še ni prišlo, bo aplikacija iskala podatke iz podatkovnega strežnika. Poudarek je tudi na hitrosti, zato želimo, da se operacije branja in pisanja med aplikacijo in predpomnilnikom izvajajo lokalno, zato je zaradi teh dveh razlogov primernejša izbira aktivno-aktivna replikacija.

2.2.1. Namestitev in konfiguracija *memcached* in *repcached*

Memcached lahko prenesemo z upravljalnikom paketov APT, *repcached* pa prenesemo iz uradne strani.

```
sudo apt-get install memcached
lynx http://sourceforge.net/projects/repcached/files/repcached/2.2-1.2.8/memcached-1.2.8-repcached-2.2.tar.gz/download
tar xvf memcached-1.2.8-repcached-2.2.tar
cd memcached-1.2.8-repcached-2.2/
./configure --enable-replication
make
make install
```

Na tej točki imamo dve instalaciji programa *memcached*. *Memcached* iz APT paketa je nameščen v `/usr/bin/memcached`, *repcached* je pa nameščen v `/usr/local/bin/memcached`. Kljub temu, da ima enako ime, je originalni *memcached* nespremenjen.

Konfiguracijo originalnega *memcached* lahko uporabimo kot vzorec za *repcached*, tako da konfiguracijsko datoteko kopiramo.

```
cp /etc/memcached.conf /etc/repcached.conf
vi /etc/repcached.conf
```

Konfiguracija *repcached* se od *memcached* razlikuje v samo dveh argumentih, ki jih je potrebno dodati, in sicer vrata za replikacijo ter naslov *master* strežnika [8].

```
# Port for server replication. Default is 11212
-X 11212

# IP for repcached master server
-x 88.255.244.22
```

Repcached nastavimo, da se zažene ob zagonu sistema (angl. *boot*) tako, da kopiramo že obstoječe *init.d* skripte programa *memcached* in jih preuredimo.

Init, krajše za *initialization*, je proces v operacijskih sistemih (Linux, BSD), ki so bazirani na UNIX operacijskem sistemu. *Init* je prvi proces, ki se zažene ob zagonu operacijskega sistema in je edini proces brez starševskega procesa [7]. *Init* ob zagonu sistema požene skripte, ki se nahajajo v mapi `/etc/init.d` [7].

V mapi `/etc/default` se nahajajo konfiguracijske datoteke za nekatere *init.d* skripte. V teh konfiguracijskih datotekah so navedene spremenljivke, ki vplivajo na okolje in delovanje *init.d* skript. *Memcached* ima v `/etc/default` konfiguracijsko datoteko s spremenljivko `ENABLE_MEMCACHED`. S to spremenljivko lahko na enostaven in hiter način določimo, ali naj se *memcached* zažene ob zagonu sistema.

Za *repcached* lahko storimo enako, ustvarimo datoteko `repcached` v mapi `/etc/default` in v njej deklariramo spremenljivko `ENABLE_REPCACHED`.

```
# Set this to no to disable repcached.
ENABLE_REPCACHED=yes
```

Skopiramo še *init.d* skripto programa *memcached*.

```
cd /etc/init.d
cp memcached repcached
```

Dejanski zagon programa *memcached* se izvede v drugi skripti, ki jo pokliče *init.d* skripta programa *memcached*, zato je potrebno v `repcached` *init.d* skripti spremeniti pot do programa *repcached*, spremenljivko `ENABLE_REPCACHED` ter pot do zagonske skripte programa *repcached*.

init.d skripta za `repcached`:

```
PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
DAEMON=/usr/local/bin/memcached
DAEMONBOOTSTRAP=/usr/share/memcached/scripts/start-repcached
NAME=repcached
DESC=repcached
PIDFILE=/var/run/$NAME.pid

test -x $DAEMON || exit 0
test -x $DAEMONBOOTSTRAP || exit 0

set -e

. /lib/lsb/init-functions

ENABLE_REPCACHED=no
test -r /etc/default/repcached && . /etc/default/repcached

case "$1" in
  start)
        echo -n "Starting $DESC: "
  if [ $ENABLE_REPCACHED = yes ]; then
        start-stop-daemon --start --quiet --exec $DAEMONBOOTSTRAP
        echo "$NAME."
        else
                echo "$NAME disabled in /etc/default/repcached."
        fi
        ;;
  stop)
        echo -n "Stopping $DESC: "
        start-stop-daemon --stop --quiet --oknodo --pidfile $PIDFILE --exec
$DAEMON
        echo "$NAME."
        rm -f $PIDFILE
        ;;

  restart|force-reload)
        echo -n "Restarting $DESC: "
```

```
        start-stop-daemon --stop --quiet --oknodo --pidfile $PIDFILE
        rm -f $PIDFILE
        sleep 1
        start-stop-daemon --start --quiet --exec $DAEMONBOOTSTRAP
        echo "$NAME."
        ;;
  status)
        status_of_proc $DAEMON $NAME
        ;;
  *)
        N=/etc/init.d/$NAME
        echo "Usage: $N {start|stop|restart|force-reload|status}" >&2
        exit 1
        ;;
esac

exit 0
```

Pot do zagonske skripte *repcached* je v *init.d* skripti določena v drugi lokaciji, zato kopiramo zagonsko skripto *memcached* v to lokacijo.

```
cp /usr/share/memcached/scripts/start-memcached \
   /usr/share/memcached/scripts/start-repcached
```

V zagonski skripti *repcached* ravno tako spremenimo določene parametre, da ne bi *init.d* in zagonske skripte obeh nameščenih programov *memcached* in *repcached*, uporabljale istih spremenljivk in datotek in ju lahko uporabljamo povsem neodvisno.

Ko so vse skripte pripravljene, jim je potrebno nastaviti pravice za izvršitev ter jih namestiti z ukazom `update-rc.d` [9].

```
chmod +x /usr/share/memcached/scripts/start-repcached
chmod +x /etc/init.d/repcached
update-rc.d repcached defaults
```

Po tem postopku se bo *repcached* zagnal ob zagonu operacijskega sistema, prav tako pa je mogoče enostavno vklapljanje in izklapljanje programa z ukazom `service`.

```
service repcached start
service repcached stop
```

2.2.2. Konfiguracija sistema za upravljanje podatkovnih baz MySQL

MySQL je najpopularnejši odprtokodni sistem za upravljanje podatkovnih baz (v nadaljevanju SUPB) [26]. Poizvedbe v *MySQL* so osnovane na programskem jeziku SQL (angl. *Structured Query Language*). S poizvedbami lahko iz SUPB beremo, pišemo, urejamo podatke v podatkovni bazi.

Visoko dosegljivost lahko v *MySQL* dosežemo na več načinov.

2.2.2.1. MySQL cluster

MySQL cluster je sinhrona rešitev, kjer si več *MySQL* strežniških instanc deli informacije. Za razliko replikacije je v lokalni skupini možno pisati in brati iz katerekoli strežniške instance, podatki se distribuirajo na ostale instance in konsistenca je zagotovljena [10]. Prednost te rešitve je tudi avtomatičen nadomestni način delovanja (angl. *failover*) [10]. Zaradi sinhronosti je rešitev primerna na lokalni mreži (v nadaljevanju LAN, *Local Area Network*) na gigabitnih povezavah, za geografsko ločene instance je pa rešitev neuporabna. *MySQL cluster* je podprt samo na določenih platformah [10].

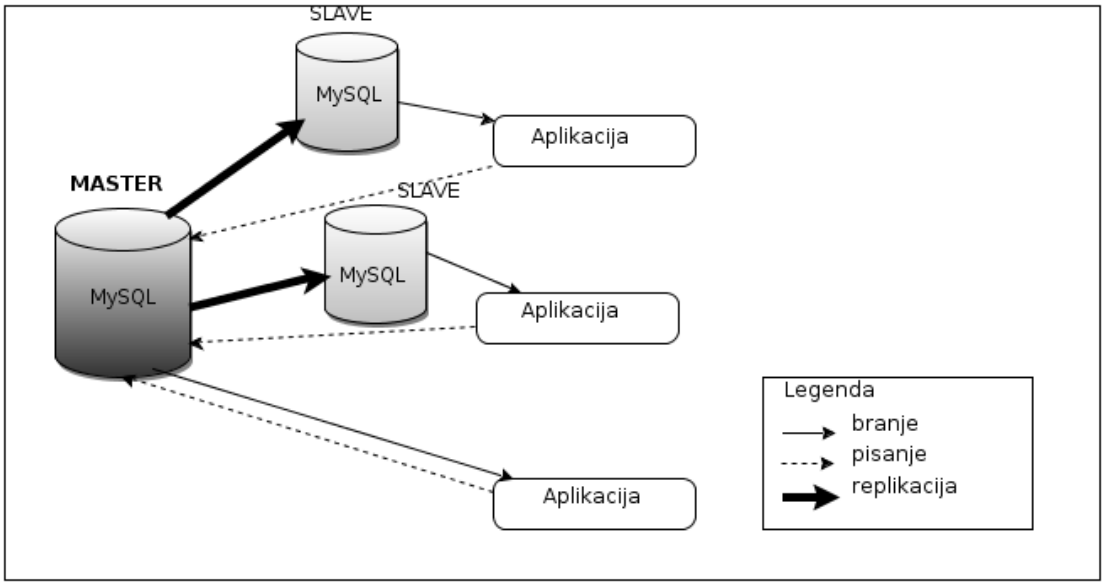
2.2.2.2. DRBD

DRBD (Distributed Replicated Block Device) je rešitev, ki je trenutno podprta samo na operacijskem sistemu Linux [10]. DRBD shranjuje podatkovno bazo na virtualni podatkovni sistem, ki se nato replicira na drugi strežnik. Ker replikacija poteka na nivoju podatkovnega sistema, so podatki bolj zanesljivi kot pri replikacijah, ki potekajo na podatkovnem nivoju [10]. Prednost rešitve je torej integriteta kopij podatkov v primeru izpada. Ker sekundarna vozlišča ne morejo uporabljati podatkovnega sistema, medtem ko se ta replicira, rešitev ni primerna za simultano aktivnost z drugimi strežniškimi instancami [10]. Aktivna je samo ena strežniška instanca, ostale so pasivne in prejemajo replicirane podatke. Ob primeru izpada aktivne instance se druga aktivira in prevzame delo.

Edina rešitev za replikacijo podatkov med geografsko ločenimi strežniškimi instancami je torej *MySQL* replikacija. Replikacija poteka po modelu *master-slave*, vsaka strežniška instanca ima lahko v nekem trenutku definiran samo en *master* strežnik, medtem ko je lahko na en *master* strežnik povezanih več *slave* strežnikov. *Master* strežnik se lahko tekoči strežniški instanci spremeni z ukazom `CHANGE MASTER TO`.

2.2.2.3. MySQL replikacija

MySQL replikacija omogoča asinhrono replikacijo MySQL stavkov in podatkov med dvema strežniškima instancama [10]. Podatki se replicirajo iz enega *MySQL master* strežnika v več *MySQL slave* strežnikov [10]. Pojma *master* in *slave* v tem kontekstu imata podobno vlogo kot pri *memcached*. Prednosti *MySQL* replikacije so podprtost na vseh platformah, ki jih podpira *MySQL*, zaradi asinhronosti je možno replikacijo začasno ustaviti ter jo nadaljevati kasneje. Replikacija je primerna pri aplikacijah z malo *master* strežniki in več *slave* strežniki, kjer aplikacije izvršujejo več branja iz podatkovne baze in manj pisanja [10]. Slabost asinhronosti pri replikaciji je nezmožnost zagotavljanja konsistentnosti podatkov v določenem trenutku [10].



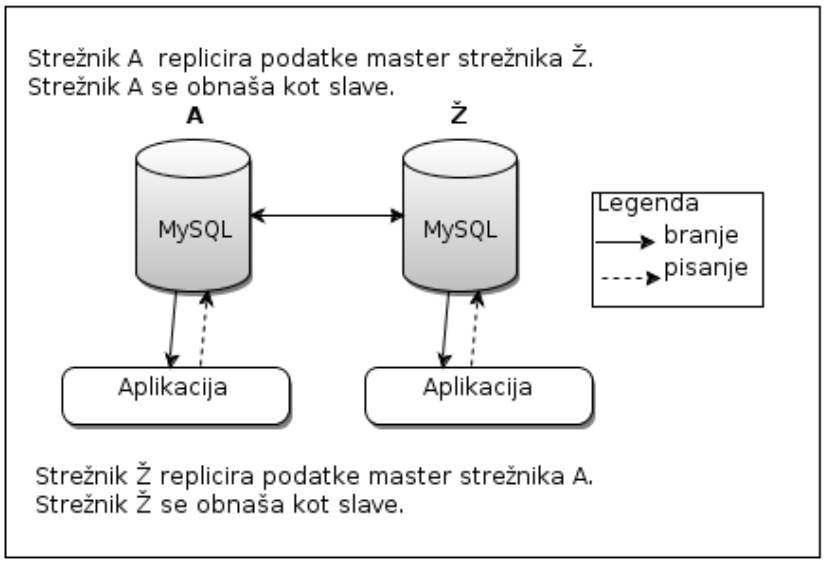
Slika 7: Model MySQL replikacije master-slave.

Da se ohranja konsistentnost podatkov na vseh vozliščih, je potrebno izvajati pisanje samo na *master* strežniku, slave strežniki pa prejemajo replicirane ukaze. Podatke lahko beremo iz katerekoli instance.

2.2.2.3.1. Aktivno-aktivna *Master-master* replikacija

V primeru dveh strežniških *MySQL* instanc lahko posameznemu *master* strežniku tudi navedemo *master* strežnik in tako omogočimo pisanje v katerokoli od instanc. Takšno konfiguracijo imenujemo *master-master* replikacija. Obe instanci opravljata vlogo *master* in *slave*, vloga je pa odvisna od tega, katera instanca prejme ukaz za zapis. Potrebno je seveda obravnavati ukaz zapisa pri replikaciji drugače, kot ukaz zapisa, ki ga pošlje *MySQL* klient.

Ob spremembi podatkov v podatkovni bazi se sprememba piše v binarnem formatu v datoteko `bin-log` [11]. Poleg replikacije se `bin-log` uporablja tudi za rekonstrukcijo podatkov ob izpadu. Celotna vsebina od določene oz. zadnje prejete pozicije dalje se nato iz *master* strežnika pošlje slave strežnikom, ki celotno vsebino datoteke `bin-log` izvršijo pri sebi. Izvršeni ukazi se pa ne zapišejo v `bin-log` temveč v `relay-log` [12]. V nasprotnem primeru bi se podatki v *master-master* konfiguraciji neprestano replicirali.



Slika 8: Model MySQL replikacije master-master z dvema aktivnima strežnikoma.

Zaradi asinhronosti je takoj očiten problem, ko želita oba strežnika zapisati informacijo na isto mesto istočasno. Naj bo to nov zapis v tabeli (`INSERT`, `REPLACE`) ali sprememba obstoječega podatka v podatkovni tabeli (`UPDATE`). Najbolj pogost problem je pri *auto-increment* stolpcih. To so primarni ključi v tabeli, katerih vrednost se avtomatično nastavi ob vpisu nove vrstice v tabelo. Denimo, da ima tabela pet vrstic, oba strežnika pa bi rada istočasno zapisala šesto vrstico v tabelo. *Auto-increment* stolpec v vrstici se bo pri obeh strežnikih zapisal kot 6. Ker je to tudi primarni ključ, se bo replikacija na obeh mestih ustavila, ker je primarni ključ 6 unikaten in je že v tabeli. Vse nadaljnje spremembe se ne bodo replicirale, dokler ta problem ne bo odpravljen, zato imata strežnika od tega trenutka dalje nekonsistentne podatke. Problem je težko rešljiv, ker je težko določiti, kateri izmed podatkov naj se ohrani, ali tisti, na prvem strežniku, ali tisti, na drugem. Problem se pa lahko zaobide tako, da preskočimo [14] izvršitev tega ukaza in nadaljujemo z replikacijo, vendar podatki ostanejo nekonsistentni.

```
mysql> SET GLOBAL SQL_SLAVE_SKIP_COUNTER = 1;
mysql> START SLAVE;
```

Stavki `INSERT … ON DUPLICATE KEY UPDATE` so ravno tako problematični, saj se replikacija izvaja na binarnem nivoju. Če se na strežniku, ki je prejel ukaz izvede `INSERT`, se bo v `bin-log` zapisal `INSERT`, v nasprotnem primeru se zapiše `UPDATE`. V `bin-log`, torej tudi v replikaciji, pa se ne zapiše pogoj, saj tako zopet ne moremo zagotavljati konsistentnosti. Replikacija ne sme dopuščati možnosti, da se na enem strežniku izvede `INSERT`, na drugem pa `UPDATE`.

Za vzpostavitev replikacije, uredimo konfiguracijsko datoteko *MySQL* `my.cnf` v mapi `/etc/mysql`. Pri replikaciji mora imeti vsaka strežniška instanca nastavljen unikatni `server-id`. *Master* strežniki morajo imeti vklopljeno beleženje binarnega dnevnika. Ta se posreduje *slave* strežnikom. Nastavimo tudi IP naslov *master* strežnika ter uporabniško ime in geslo za povezavo. Če želimo, da replikacija poteka samo za določeno podatkovno bazo ali tabelo, lahko to nastavimo pri spremenljivkah `replicate-do-db` in `replicate-wild-do-table`.

Da rešimo problem *auto-increment* stolpcev, lahko nastavimo, da pri *master-master* replikaciji en strežnik vstavlja sode *auto-increment* ključe, drugi pa lihe [15]. Rešitev je možno na podoben način prilagoditi tudi za več *master* strežnikov v topologiji. Pri spremenljivki `auto_increment_increment` navedemo število *master* strežnikov v topologiji. Pri spremenljivki `auto_increment_offset` pri posameznem *master* strežniku vnesemo unikatno celo število na intervalu [1 .. število *master* strežnikov].

Primer konfiguracije za *master-master* replikacijo pri strežniku Ž:

```
server-id                       = 1
log_bin                         = /var/log/mysql/mysql-bin.log
master-host                     = 64.255.221.73
master-user                     = slave
master-password                 = slavepass

log-bin                         = mysql-bin

replicate-do-db                 = kafol
replicate-wild-do-table         = kafol.%

auto_increment_offset           = 2
auto_increment_increment        = 2
```

Problem unikatnih ključev pri *master-master* replikaciji se lahko pojavi tudi pri stolpcih, ki niso *auto-increment*. Recimo, da imata v tabeli oba *master* strežnika enake podatke, na kar se v nekem trenutku v tabelo z unikatnim stolpcem na enem od strežniku zapiše nek podatek. Replikacija se sicer zgodi takoj, ko se lahko, vendar se lahko zaradi več dejavnikov (zasedeni resursi, počasna povezava med strežniki) zakasni. Zakasnitev nima zgornje meje, lahko traja tudi več ur. Dokler drugi strežnik ne prejme repliciranih podatkov, so podatki nekonsistentni. V času zakasnitve lahko drugi strežnik v isto tabelo zapiše drugačno vrstico, z enakim unikatnim ključem, ki se še ni repliciral iz prvega strežnika. Replikacijo lahko konfiguriramo tako, da se kljub dupliciranem unikatnem ključu replikacija ne ustavi. Napako lahko ignorira

in nadaljuje z replikacijo, podatki pa ostanejo nekonsistentni. V spremenljivki `slave-skip-errors` lahko zapišemo z vejico ločene šifre napak, ki naj jih *slave* strežnik pri replikaciji ignorira.

2.2.2.3.2. Aktivno-pasivna *master-master* replikacija

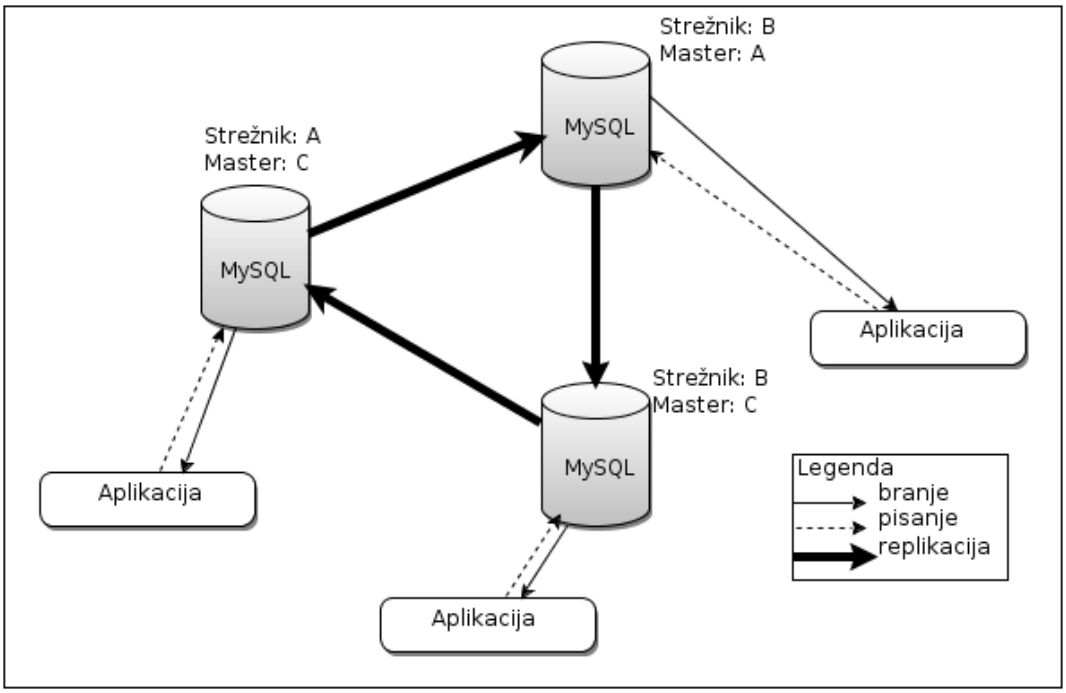
Pri *master-master* replikaciji tako razen *auto-increment* stolpcev ne moremo imeti unikatnih ključev, če sta oba *master* strežnika aktivna. Boljša rešitev je imeti en aktiven in en pasiven *master* strežnik. Pasiven strežnik prejema replikacijo, iz njega se lahko bere, piše pa se samo na aktivni strežnik. Ob primeru izpada morajo aplikacije izbrati drug *master* strežnik za pisanje in ta postane aktiven. Takšna rešitev sicer ne reši problema v celoti, vendar zelo zmanjša verjetnost, da pride do napake.

Problem nekonsistentnosti pri posodabljanju že obstoječih podatkov v tabeli je potrebno reševati na višjem, aplikacijskem nivoju. Če sta oba *master* strežnika aktivna, morajo aplikacije same poskrbeti, da oba *master* strežnika ne pišeta istočasno po istih kosih podatkov.

Za aktivno-pasivne strežnike obstaja več programskih rešitev, ki preverjajo dosegljivost posameznega strežnika in ležijo na vmesnem nivoju med aplikacijo in podatkovno bazo in izvajajo preklope ob izpadu. Ena izmed rešitev je *LinuxHA*, ki je bolj primerna za lokalne skupine, saj ne reši problema nekonsistentnosti [16]. Ena izmed enostavnejših rešitev, ki bi se lahko izkazala za uporabno je *Multi-Master Replication Manager for MySQL (MySQL-mmm)*. To je nabor skript, ki izvajajo preklop aktivnega *master* strežnika v primeru izpada [17]. V dnevniku sprememb *MySQL-mmm* [18] je opaziti, da se razvijalci še vedno ukvarjajo s stabilnostjo skript, zato ta rešitev ni bila preizkušena. Upravljalnik preklopov ne sme biti manj zanesljiv od samega sistema.

2.2.2.3.3. Krožna konfiguracija *master-master* replikacije z več strežniki

V primeru treh ali več *master* strežnikov je potrebno zaradi ohranjanja konsistentnosti podatkov vzpostaviti krožno konfiguracijo (angl. *ring configuration*), ker ima lahko posamezna strežniška *MySQL* instanca določen samo en *master* strežnik.



Slika 9: Krožna konfiguracija replikacije s tremi aktivnimi MySQL master strežniki.

Ker se `relay-log` ne uporablja za replikacijo, je potrebno nastaviti konfiguracijski spremenljivki `log-slave-updates` in `replicate-same-server-id`. Prva spremenljivka spremeni privzeto obnašanje zapisovanja sprememb, in sicer če je spremenljivka nastavljena, se bo vsebina `relay-log` datoteke zapisala tudi v `bin-log`, ki pa se uporabi v replikaciji ter pošlje še drugemu strežniku v topologiji. Ker ima vsak strežnik svojo šifro (ID), s spremenljivko `replicate-same-server-id` preprečimo ciklanje replikacije v topologiji, tako da ji nastavimo vrednost šifre trenutnega strežnika [13].

Iz slike krožne konfiguracije treh strežnikov je razvidno, da se ob izpadu ene strežniške instance podre replikacija. Ob primeru izpada strežnika C, strežnik A ne bo prejemal repliciranih podatkov, ki so se zapisali v strežnik B. Strežnik B bo dobival replicirane podatke strežnika A, ker je nanj neposredno povezan kot *slave* strežnik.

Krožne konfiguracije zato zmanjšajo visoko dosegljivost celotnega sistema podatkovne baze. V primeru treh strežnikov je bolj primerno imeti dva *master* strežnika in en *slave* strežnik. Tako imamo tri strežnike za branje ter dva za pisanje.

Izbrana topologija v prototipu je *master-master* replikacija dveh strežnikov, pri katerem je en aktiven, drugi pasiven. Kateri strežnik je aktiven, kateri pasiven se odloča aplikacija sama, brez uporabe rediteljev (angl. *monitor*) kot so *LinuxHA* ali *MySQL-mmm*.

2.2.3. Namestitev DNS strežnikov in zapisov

DNS (angl. *Domain Name System*) je hierarhičen sistem strežnikov, ki prevede absolutno domeno (v nadaljevanju *FQDN*, angl. *Fully Qualified Domain Name*) v IP (angl. *Internet Protocol*) naslov [19]. Programski paket *bind* opravlja funkcijo DNS strežnika na operacijskem sistemu *Linux*. Prenesemo ga lahko z upravljalnikom paketov APT.

```
sudo apt-get install bind9
```

Strežnik *bind* deluje hierarhično po modelu *master-slave*, tako da lahko na sekundarnem strežniku namestimo *bind* kot *slave* strežnik, ki ima replicirane zapise *master* strežnika. Konfiguracija strežnika *bind* se nahaja v datoteki `/etc/named.conf`. V tej datoteki je navedena vloga strežnika (*master* ali *slave*) ter poti datotek z DNS zapisi [20].

Primer konfiguracije pri *master* strežniku:

```
options {
      directory "/var/named";
      allow-transfer { 88.255.244.22; }; // naslovi slave streznikov
};

zone  "primer" {
      type master;
      file "primer.dns";
};
```

V direktoriju `/var/named` se nahaja datoteka `primer.dns` z DNS zapisi domene. *Slave* strežnik ima v konfiguraciji naveden IP naslov *master* strežnika.

```
zone "primer" in {
      type slave;
      file " primer.dns.slave";
      masters { 193.255.66.55; };
};
```

Najpogostejši tipi DNS zapisov coni so:

- SOA (*angl. Start of Authority*), ki je obvezen zapis in drugim DNS strežnikom označuje, da je ta DNS strežnik najboljši vir informacij za podatke v tej coni [33]. Parametri v SOA zapisu so izvirni naslov strežnika, elektronski naslov administratorja cone, serijska številka (ali revizija) cone, čas ponovne osvežitve sekundarnih strežnikov, čas za ponovni poskus prenosa domene ob spodletelem poskusu, interval veljavnosti cone in interval veljavnosti za zapise v coni (TTL, angl. *Time To Live*)
- A (angl. *Address record*), to je zapis za 32-bitni IPv4 naslov
- AAAA, zapis za 64-bitni IPv6 naslov
- CNAME (angl. *Canonical name record*) je alias oziroma drugo ime za drugi DNS zapis

- NS (angl. *Name Server record*) zapis določa, kateri DNS strežniki lahko upravljajo z domeno ali poddomenami
- MX (angl. *Mail Exhange record*) zapis navaja prioriteto in naslove strežnikov, ki upravljajo elektronsko pošto na domeni ali poddomeni.

Cona je domena ali poddomena s katero DNS strežnik upravlja in njenimi FQDN zapisi.

Primer DNS zapisa za cono:

```
; nastavitve domene
@ IN  SOA   example.com. hostmaster.example.com. (
      100013117 ; serijska stevilka
      3H    ; refresh
      15M   ; retry
      1W    ; expiry
      1D    ; minimum
)

; navedeni domenski strezniki za domeno
IN    NS    ns1.example.com.
IN    NS    ns2.example.com.

; dns zapisi
ns1   IN    A     193.255.66.55 ; ns1.example.com ima A zapis ta IP naslov
ns2   IN    A     88.255.244.22 ; ns2.example.com ima A zapis ta IP naslov
www   IN    CNAME ns1 ; www.example.com je alias za ns1.example.com
www   IN    CNAME ns2 ; www.example.com je alias za ns2.example.com
      IN    CNAME www ; example.com je alias za www.example.com
```

Za domeno sta tako nastavljena dva domenska strežnika, primarni in sekundarni. Prav tako sta za domeno in poddomeno *www* nastavljena dva zapisa, kar nam omogoča dve vstopni točki. V primeru več zapisov za en FQDN, se bodo zapisi prikazovali v naključnem vrstnem redu [21]. Posledica za to je, da bo ob izpadu enega od strežnikov 50% klientov od DNS strežnika dobilo IP naslov delujočega strežnika, druga polovica pa IP naslov nedelujočega strežnika. Z večanjem števila vstopnih točk višamo visoko dosegljivost sistema, kot če bi imeli samo en zapis za FQDN in bi bil ob izpadu strežnika, navedenega v zapisu, cel sistem nedosegljiv.

Pri MX zapisih (angl. *Mail Exchange*) se poleg FQDN in poštnega strežnika navede tudi prioriteta, ki jo poštni strežniki (v nadaljevanju MDA, angl. *Mail Delivery Agent*) uporabljajo, če povezava do strežnika z višjo prioriteto spodleti [22].

Ker se DNS zapisi redko spreminjajo, ni pomembno kateri strežnik je *master* in kateri *slave*. DNS strežnikov ni možno nastaviti na *master-master* konfiguracijo.

3. PROGRAMIRANJE PROTOTIPA APLIKACIJE V PHP

PHP je rekurzivni akronim za *PHP Hypertext Preprocessor* (včasih *Personal Home Page*) in je eden izmed najbolj popularnih programskih jezikov za programiranje spletnih aplikacij [23]. PHP se v večini primerov uporablja za programiranje dinamičnih spletnih strani, njegova obsežnost pa omogoča izdelovanje kakršnihkoli konzolnih aplikacij.

Prototip aplikacije na pripravljeni programski opremi je vmesnik za povezavo do podatkovnih strežnikov *MySQL* z možnostjo predpomnjenja podatkov v *memcache* ter generični vtični strežnik (angl. *socket server*) za XML (angl. *Extensible Markup Language*) komunikacijo preko TCP (angl. *Tranismission Control Protocol*). Tak strežnik se pri spletnih aplikacijah lahko uporablja za živo komunikacijo med klienti in strežnikom ter ohranjanjem žive povezave, kar HTTP sam ne omogoča.

3.1. Programiranje objekta za dostop do podatkovne baze

Objekt, ki upravlja s podatkovno bazo ima shranjen seznam podatkovnih strežnikov in glede na vrsto poizvedbe izbere strežnik, na kateremu se bo poizvedba izvedla. Poizvedbe se delijo glede na to, ali poizvedba izvaja pisanje ali branje. Kljub temu, da imamo *master-master* replikacijo, želimo, da aplikacija iz kateregakoli strežnika piše na isti *master* strežnik. Če ta strežnik ni dosegljiv, se izbere drugi. Aplikacija vedno bere iz tistega strežnika, ki je bližji, običajno `localhost`, razen če ta ni dosegljiv. V tem primeru se spet izbere naslednji strežnik iz seznama.

Spodnji primer kode prikazuje izbiro podatkovnega strežnika iz tipa poizvedbe za izvajanje operacij. Funkcija `ready()` preveri, ali je povezava s podatkovnim strežnikom vzpostavljena ter če je povezan preferenčni strežnik. Če ni, poskusi povezavo ponovno vzpostaviti, če povezava spodleti, uporabi naslednji strežnik v seznamu.

```
function query($q) {
	$regex = '/[\s\(]*(select|show)/is';
	$db = preg_match($regex,$q) ? $this->db_read : $this->db_write;
	if($db->ready()) {
		$result = mysql_query($q, $db->link);
		if(!$result) {
			$error = $db->get_error();
			if($error) trigger_error("SQL ERROR: $error");
		}
		return $result;
	} else return false;
}
```

Objekt ima kazalca na objekta `db_read` in `db_write`, ki izvajata operacije na podatkovnem strežniku.

```
function ready() {
	if($this->link && mysql_ping($this->link)) {
		//preveri ce je povezan v preferencni streznik
		if($this->host == $this->servers[0]['host']) {
			return $this->link;
		}
	}
	return $this->connect();
}

function connect() {
	// v tabeli $this->servers so strezniki urejeni po prioriteti
	foreach($this->servers as $server) {
		$this->user = $server['user'];
		$this->pass = $server['pass'];
		$this->host = $server['host'];
		$this->disconnect();
		@$this->link = mysql_connect(
			$this->host,
			$this->user,
			$this->pass
		);
		if(!$this->link) continue;
		$this->database = mysql_select_db($this->db, $this->link);
		if(!$this->database) continue;
		if(!$this->set_charset()) continue;
		return true;
	}
	$this->link = false;
	return false;
}
```

3.2. Programiranje vtičnega strežnika

Vtični strežnik je pri spletnih aplikacijah uporaben pri dogodkih, ki se dogajajo hitro in pogosto. Primeren je za aplikacije, ki omogočajo hipno sporočanje med uporabniki, ali kot strežnik spletnih iger.

Do sedaj imamo pripravljeno replikacijo podatkov na dveh nivojih – replikacija podatkovne baze ter replikacija predpomnilnika. Oba načina sta asinhrona. Vtični strežnik je primeren tudi za tiste dele spletne aplikacije, kjer potrebujemo sinhrono replikacijo podatkov med strežniki.

Izdelani prototip vtičnega strežnika je tako kot nižje-nivojska programska oprema distributiven, podatki se replicirajo med ostalimi strežniki, tako da so ob izpadu enega izmed strežnikov ostali pripravljeni, da prevzamejo njegovo delo.

Prototip strežnika posluša na dveh TCP vratih, ena so uporabljena za komunikacijo med klienti, druga za komunikacijo med ostalimi strežniki. Vtični strežnik ob inicializaciji ustvari

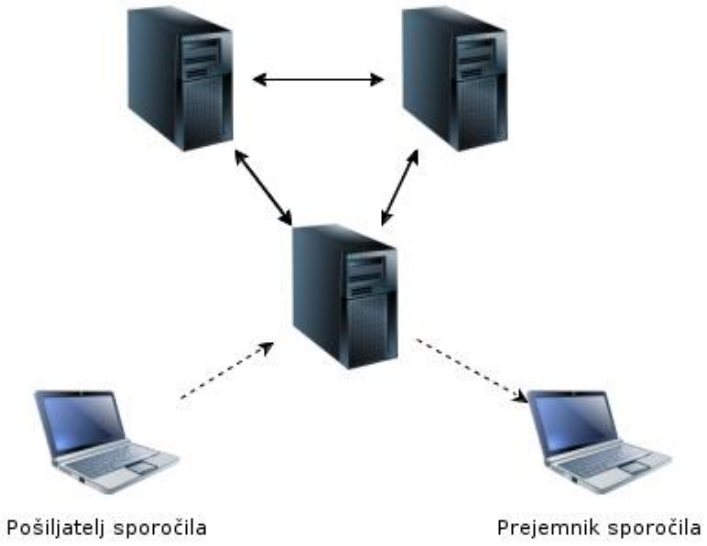
vtičnik z ukazom `socket_create()`, nato ga poveže z vrati z ukazom `socket_bind()` ter posluša na njih, z ukazom `socket_listen()`. Vtičniku je potrebno še nastaviti opcijo `socket_set_nonblock()`, zato da strežnik ne ustavi izvrševanja, dokler ne dobi odgovora iz vtičnika [25]. Ker PHP nima niti (ang. *threads*), da bi za vsako povezavo imel svojo časovno linijo za izvrševanje, je ta opcija nujna, zato da lahko strežnik nadaljuje z izvrševanjem ostalih operacij, tudi če vtičnik katerega od klientov ne vrača podatkov. Strežnik hrani tabelo klientov, ki je potrebna za ohranjanje seje. Izvrševanje se po inicializaciji odvija v neskončni zanki. V zanki se najprej pokliče funkcija `socket_select()`, ki iz tabele klientov vrne tiste vtičnike, v katere je klient nekaj zapisal in je iz njih možno brati. Sledi zanka, ki se sprehodi čez vtičnike, ter iz njih bere z ukazom `socket_recv()`. Če je v tabeli vtičnikov tudi vtičnik strežnika, ki posluša na TCP vratih, pomeni, da se na strežnik želi povezati nov klient. Vtičnik novega klienta sprejmemo s klicem funkcije `socket_accept()` ter ga shranimo v tabelo klientov. Prebrane ukaze ostalih klientov pretvorimo v XML objekt s konstruktorjem `new SimpleXMLElement()`, nato jih pa obdelujemo.

Klicu funkcije `socket_recv()` je potrebno še navesti velikost medpomnilnika (angl. *buffer*). Ker je lahko velikost prebranih podatkov večja od velikosti medpomnilnika, je potrebno implementirati svojo verzijo medpomnilnika v PHP. Ker PHP omogoča dinamično velikost spremenljivke, je dovolj ustvariti tabelo, v katero se za vsakega klienta na konec celice dodajajo prebrani podatki. Po vsakem dodajanju se celoten niz podatkov razdeli po poljubnem ločilniku, ki se venomer uporablja v lastnem komunikacijskem protokolu. Ločilnik uporabljamo za ločevanje posameznih XML sporočil. Če je število razdeljenih podatkov večje od 1, imamo vsaj en celovit kos podatkov, ki jih lahko obdelamo. Ta način medpomnjenja je potreben, če pričakujemo podatke v obliki XML. Preden podatke pošljemo v XML prevajalnik, morajo ti biti sintaktično pravilni in s tem posledično tudi celoviti.

```
function buffer_add($sock,$data) {
      if(!isset($this->buffer[$sock])) {
            $this->buffer[$sock]['data'] = '';
      }

      $this->buffer[$sock]['data'] .= $data;
      $this->buffer[$sock]['ts'] = time();
}

function buffer_get($sock) {
      // split buffer by the end of string
      $lines = preg_split('/\0/',$this->buffer[$sock]['data']);

      // reset buffer to the last line of input
      // if data was sent completely, the last line of input should be
      // an empty string
      $this->buffer[$sock]['data'] = trim($lines[count($lines)-1]);

      // remove the last line of input (incomplete data)
      // parse any complete data
      unset($lines[count($lines)-1]);
```

```
      // return only the fully sent data
      return $lines;
}
```

V isti neskončni zanki poteka komunikacija na podoben način za vtičnik, namenjen za replikacijo podatkov med ostalimi strežniki. Vsak strežnik ima seznam vseh ostalih strežnikov. Ob inicializaciji se strežnik poskuša povezati z vsemi. Neposredne obojestranske povezave med strežniki zagotavljajo najhitrejšo storitev, vendar je predvideno, da zaradi nastavitev požarnih zidov obojestranska povezava ne bo v vseh primerih mogoča. V takšnih primerih se sporočila prenašajo posredno. V topologiji treh strežnikov, pri katerem dva nista neposredno povezana, je vmesni strežnik uporabljen kot rele.



Slika 10: Neposredna povezava v topologiji treh strežnikov.

Da preprečimo kroženje sporočil pri obojestranskih povezavah, se poleg sporočila pošlje še seznam strežnikov, ki so ta ukaz že prejeli. Ko strežnik opravlja funkcijo releja, s pomočjo tega seznama izloči strežnike, ki jim je potrebno sporočilo posredovati, posredovanemu sporočilu pa doda dopolnjen seznam strežnikov.

```
/*
 * Sends message all connected nodes
 * they need to propagate message to other nodes
 * include an ignore list
 */
function servers_send($command,$data,$ignored_servers=null) {
      $connected_servers = $this->get_connected_servers();

      foreach($connected_servers as $server_id) {
```

```
            if(is_array($ignored_servers)) {
                  if(in_array($server_id,$ignored_servers)) {
                        continue;
                  }
            }

            $server = $this->get_server($server_id);
            $this->send(
                  $this->servers[$server]['sock'],
                  $command,
                  array_merge($data,array(
                        'ignore' => implode(',',array_merge(
                              (array) $ignored_servers,
                              (array) $connected_servers,
                              (array) $this->id
                        ))
                  )
            ));
      }
}
```

Ob vzpostavitvi povezave oba strežnika drug drugemu pošljeta seznam klientov ter drugih podatkov, ki jih želimo replicirati. Prejete podatke strežnika združita. Ko se tretji strežnik poveže, prejme združen seznam podatkov, ostala dva strežnika pa k svojim dodata še podatke tretjega. Vsi podatki se torej neprestano replicirajo, ob izpadu strežnika pa se klient poskusi povezati na drug strežnik. Ta strežnik že vsebuje podatke o klientu, klient ob vzpostavitvi povezave pošlje šifro seje, s katero lahko strežnik identificira klienta in nadaljuje z njegovo sejo. Strežnik obstoječemu klientu zamenja resurs novega vtičnika.

```
case 'update-client':
      if($this->is_connected($i)) {
            $object = $this->decode($xml->data);
            merge_clients(array(
                  $object->id => $object
            ));

            $this->servers_send('update-client',array(
                  'data' => $xml->data,
            ),explode(',',strval($xml->ignore)));
      }
break;
```

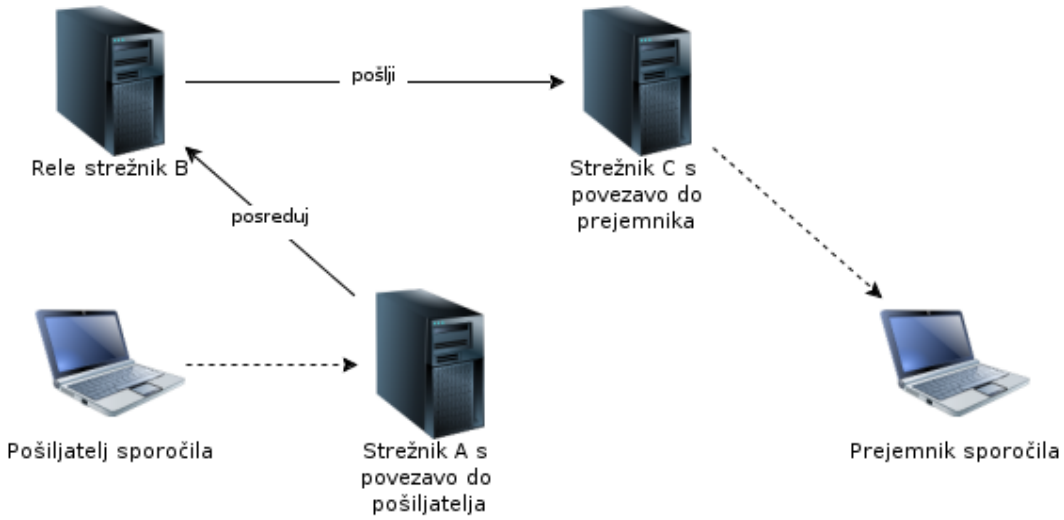
```
function merge_clients($arr = null) {
      global $clients;

      //save existing socket resources so
      //they don't get overwritten by unserialize()
      //resources are not preserved in serialization,
      //so existing connections get destroyed
      $sockets = array();
      foreach($clients as $id=>$client) {
            $sockets[$id] = $client->socket;
      }
```

```
	// merge sent clients with existing ones
	$clients = array_merge(
		(array) $clients,
		(array) $arr
	);

	// restore existing and active socket resources,
	// so that clients don't get disconnected
	foreach($sockets as $id=>$sock) {
		$clients[$id]->socket = $sock;
	}

	// reset pings, so clients don't get disconnected
	if(is_array($arr)) {
		foreach($arr as $i=>$client) {
			$clients[$i]->ping = time();
			$clients[$i]->pong = $clients[$i]->ping;
		}
	}

	return $clients;
}
```

Klienti, povezani na različne strežnike se vidijo med seboj, sporočilo enega klienta drugemu, pa posredujejo strežniki med seboj. Ko strežnik prejme sporočilo klienta, najprej preveri, če ima resurs do vtičnika prejemnika sporočila in če ga ima, pošlje sporočilo neposredno klientu. V nasprotnem primeru strežnik najprej preveri, ali ima povezavo do drugega strežnika z resursom vtičnika tega klienta. Če povezava do strežnika s klientom obstaja, mu strežnik pošlje ukaz, naj sporočilo posreduje temu klientu. Če oba načina spodletita, strežnik vsem ostalim povezanim strežnikom pošlje ukaz, naj si to sporočilo posredujejo med seboj, dokler sporočilo ne pride do strežnika, na katerega je povezan klient. Pri posredovanju sporočil se dodaja seznam strežnikov, katerim je bilo posredovano sporočilo že poslano, da se izognemo krožnemu posredovanju sporočila.

```
/*
 * Propagate messages that were sent to clients not connected to this node
 */
function propagate(user &$client,&$message,$ignored_servers = null) {
	if($client->server == $this->id) {
		// this node has the connection to the client
		return $this->write($client->socket,$message);
	} else {
		$server = $this->get_server($client->server);
		if($server !== false) {
			// node has connection to the server with this client
			$this->send(
				$this->servers[$server]['sock'],
				'relay',
				array(
					'client_id' => $client->id,
					'message'   => $this->encode($message)
				)
```

```
                    );
              } else {
                    // node is not connected to any server with this client
                    // propagate message to all nodes with ignore list
                    $this->servers_send('propagate',array(
                          'client_id' => $client->id,
                          'message'   => $this->encode($message),
                    ),$ignored_servers);
              }
        }
        return false;
}
```

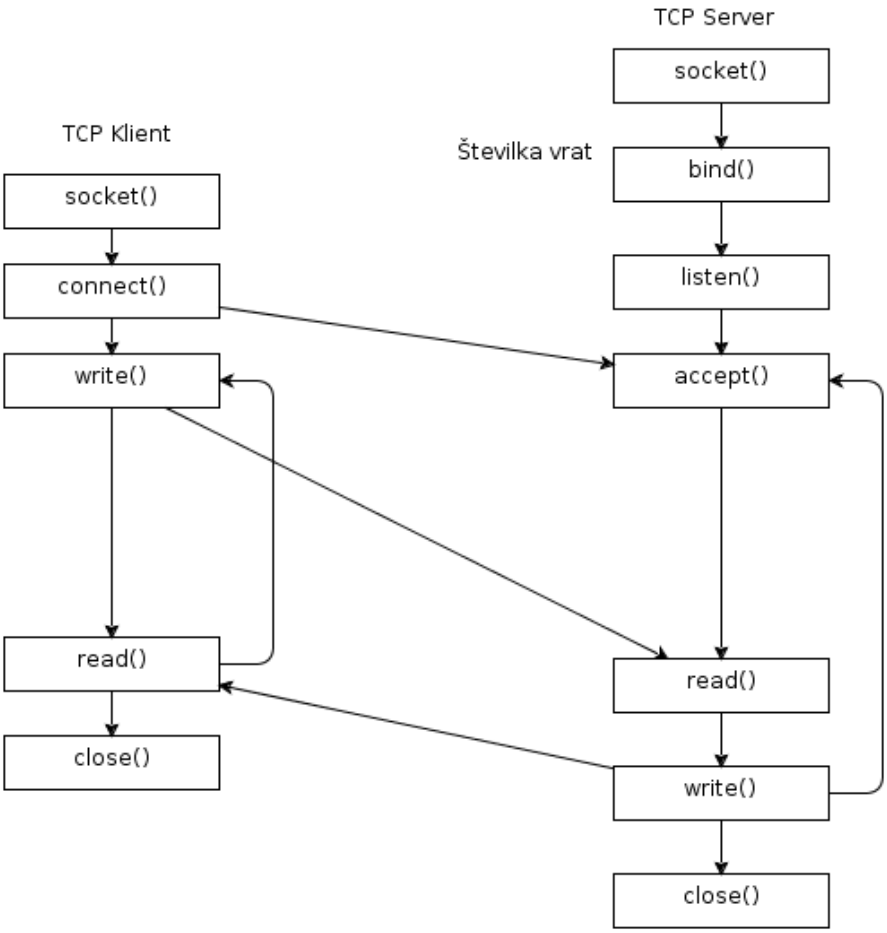


Slika 11: Posredovanje sporočila med strežniki.

Komunikacija med posredno povezanima strežnikoma A in C poteka preko rele strežnika B.

Sporočila XML se v izdelanem strežniku pred pošiljanjem stisnejo v formatu *gzip*, da se minimizira velikost poslanih in prejetih podatkov in s tem veča hitrost komunikacije. PHP-jeve funkcije `gzinflate()` in `gzdeflate()` so hitrejše in stisnjeni podatki so krajši od `gzcompress()` in `gzuncompress()`, ker ne vključujejo glave *gzip* formata [28].

Za vsakega klienta, naj bo to uporabnik ali drug strežnik, se periodično preverja aktivnost. Ker se v določenih primerih ne zazna prekinitev povezave med klientom in strežnikom, se na podlagi tega preverjanja aktivnosti strežnik odloči, ali naj povezavo prekine. Aktivnost se preverja na tak način, da strežnik pošlje ukaz z določenim parametrom klientu, klient pa mora odgovoriti na ta ukaz z istim parametrom. Na določen interval, na primer 30 sekund, strežnik preveri, če je poslani parameter enak prejetemu. Če ni, strežnik sklepa, da je klient vsaj 30 sekund v zaostanku z aktivno sejo in povezavo prekine. Tak mehanizem preverjanja aktivnosti običajno v angleščini imenujemo kot *pinging*.



Slika 12: Model klient-strežnik.

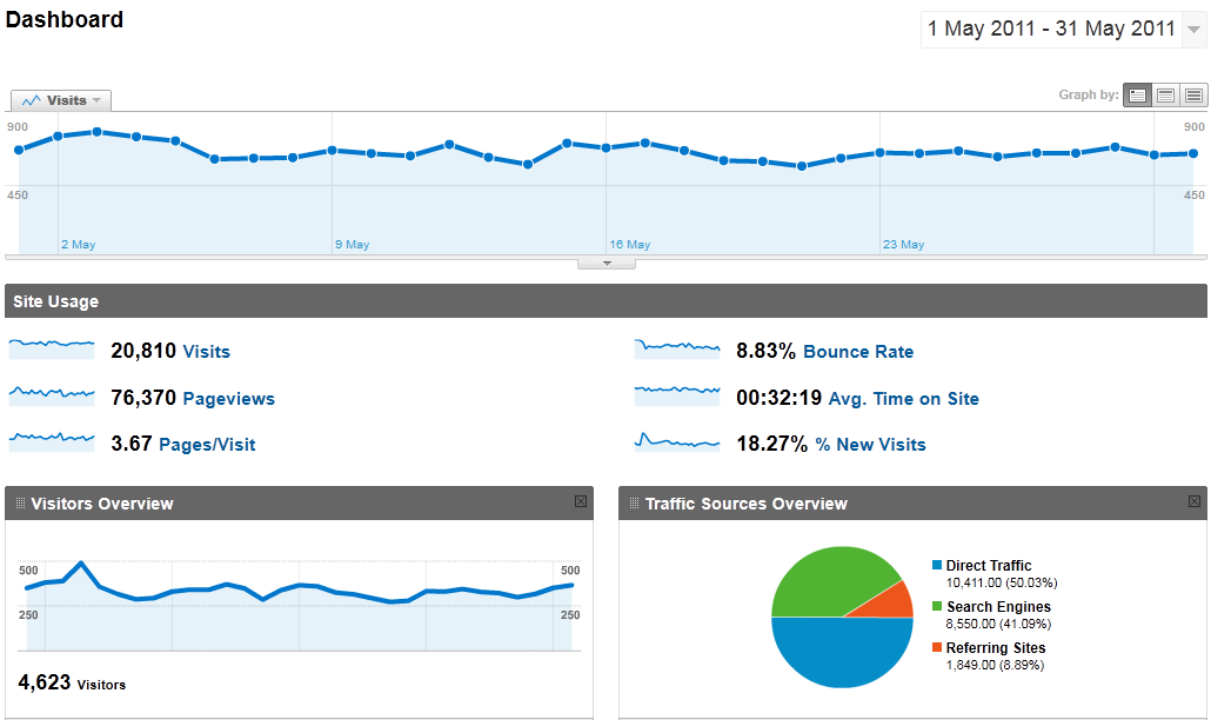
Prirejeno po viru: 27.

4. TESTIRANJE

Izdelan sistem sem preizkusil s predelavo obstoječe aplikacije – spletne igre s kartami – v izdelani prototip. Uporabniški vmesnik je razdeljen na dva dela – spletna stran in *Flash* vmesnik, ki je integriran v spletno stran. Uporabnik ima dve individualni povezavi do aplikacije – stalna povezava do *Flash* vmesnika, ki je povezan na vtični strežnik ter povezava do HTTP strežnika, za izvajanje AJAX (angl. *Asynchronous JavaScript and XML*) klicev ter serviranje spletne strani in *Flash* vmesnika. AJAX je množica programskih tehnik, ki omogočajo dinamično nalaganje vsebin ali posameznih delov vsebin na spletni strani brez ponovne naložitve celotne spletne strani [29]. Shema komunikacij je prikazana v sliki 1.

Individualni povezavi lahko kažeta na različna strežnika, na primer, če se klientu servira spletna stran iz strežnika Ž, se lahko *Flash* vmesnik poveže na strežnik A, AJAX klice pa še vedno izvaja na strežniku Ž.

Sistem je bil testiran v mesecu maju 2011 na predelani obstoječi spletni aplikaciji Briskula.si, dostopni na naslovu [31]. Briskula.si je spletni portal za igranje iger s kartami (briškole, tršeta in kifameno), ki sem ga sprogramiral avgusta 2008. V mesecu maju je bilo zabeleženih v povprečju 650 obiskov dnevno, od teh dnevno v povprečju 320 unikatnih obiskovalcev, večinoma iz Hrvaške (9.891 obiskov), Slovenije (8.117 obiskov) in Srbije (1.364 obiskov).



Slika 13: Statistika obiskov portala Briskula.si izmerjenega z orodjem Google Analytics.

4.1. Testiranje RAID-1 polja

RAID-1 polje sem testiral tako, da sem izklopil enega izmed diskov in nato strežnik ponovno prižgal ter preveril, ali so vsi podatki še vedno prisotni. Ne glede na to, kateri disk sem izklopil, so bili podatki identični in sistem je deloval brezhibno. Vsakič, ko sem disk priklopil nazaj, se je vsebina drugega diska zaradi konsistentnosti avtomatično v celoti ponovno zapisala na zaostali disk. Ponovni zapis je trajal približno eno uro.

4.2. Testiranje predpomnilnika *repcached*

Po vzpostaviti *repcached* se je vsebina predpomnilnika normalno replicirala med strežnikoma. Ob dodajanju, brisanju ali spreminjanju vrednosti v predpomnilnik na enem *repcached* strežniku se je ukaz repliciral še na drugem. Težav ali napak pri *repcached* nisem opazil.

Na strežniku A:

```
root@s2:/etc# telnet localhost 11211
Trying ::1...
Connected to localhost.
Escape character is '^]'.
set foo 0 60 3
bar
STORED
get foo
VALUE foo 0 3
bar
END
```

Na strežniku Ž

```
root@s1:/etc# telnet localhost 11211
Trying ::1...
Connected to localhost.
Escape character is '^]'.
get foo
VALUE foo 0 3
bar
END
```

4.3. Testiranje DNS zapisov

Dodal sem dve poddomeni za *briskula.si* – *s1*, ki je kazala na naslov strežnika Ž ter *s2*, ki je kazala na naslov strežnika A. Domena *briskula.si* je imela dva CNAME zapisa – *s1.briskula.si* in *s2.briskula.si*, obstoječe poddomene *www*, *en*, *hr*, *sr*, *pl*, *ba* pa CNAME zapis na *briskula.si*. Sistem je deloval po pričakovanju. Nekateri uporabniki, ki so obiskali portal,

so se povezovali na strežnik A (*s2*), drugi pa na strežnik Ž (*s1*). Izbira naslova je bila avtomatična in naključna.

4.4. Testiranje replikacije podatkovne baze

Testiranje je sprva potekalo na aktivno-aktivni *master-master* replikaciji podatkovne baze. Aplikacije so lahko pisale in brale podatke iz kateregakoli podatkovnega strežnika. Ker je večina unikatnih ključev v relacijski shemi *auto-increment* polj, sem ocenil majhno verjetnost, da lahko pride do kolizije ključev. Unikatni ključ sem imel še pri tabeli uporabnikov pri polju za uporabniško ime.

Po treh tednih normalnega delovanja pa je ravno tu prišlo do kolizije. Po analizi izpisov *Apache* spletnega strežnika sem ugotovil, da se je uporabnik registriral dvakrat, na vsakem strežniku posebej. Dvojna registracija se je zgodila preden se je izvedla replikacija, ki naj bi drugo registracijo zavrnila, ker bi bil uporabnik tudi na drugem strežniku že v tabeli.

Na portalu sem dodal možnost registracije z uporabniškim računom socialnega omrežja *Facebook*. Ko uporabnik aplikaciji dovoli, da dostopa do javnih podatkov, aplikacija registrira (doda podatke o uporabniku v tabelo) novega uporabnika avtomatično. Ko je uporabnik ponovno naložil stran, je po vsej verjetnosti iz DNS strežnika dobil drugače urejen seznam strežnikov, zato se je povezal na drugi strežnik. Drugi strežnik še ni prejel replikacije, zato ga je ponovno avtomatično registriral, ker je lahko dostopal do uporabniških podatkov iz *Facebook*-a.

Objekta za dostop do baze sem preuredil na tak način, da sta vedno zapisovala podatke v isti strežnik, če sta bila oba dosegljiva. S tem sem zmanjšal verjetnost kolizij, ki so najpogostejše napake pri replikaciji.

V času testiranja se je replikacija ustavila zaradi še ene napake. Na enemu izmed strežnikov se je zaradi preobremenitve sesul *MySQL* strežnik. Predno se je sesul, je drugemu strežniku poslal novo pozicijo za repliciranje, ni mu pa poslal vsebine `bin-log`. *MySQL* po privzetem načinu delovanja ne shranjuje sprememb v datoteko `bin-log` takoj, ampak jih nekaj časa hrani v pomnilniku in jih občasno zapiše na disk [32], da se minimizira količina vzhodno-izhodnih operacij na disku, ki so ene izmed najpočasnejših operacij. Ko se je *MySQL* strežnik ponovno zagnal, je drugi *MySQL* strežnik zahteval neveljavno pozicijo iz `bin-log`, ki je prvi strežnik ni imel zapisane. Najbolj varna in najpočasnejša rešitev tega problema je opcija `sync_binlog` v konfiguracijski datoteki *MySQL*. Z nastavitvijo na vrednost 1, se po vsaki spremembi zapiše podatek tudi v `bin-log` [32].

4.5. Testiranje vtičnega strežnika

Pri vtičnem strežniku – strežniku za igranje kart – na katerega so se povezovali *Flash* vmesniki, se je v času testiranja pojavilo nekoliko več napak. Ker ure med strežnikoma niso bile sinhronizirane, sta strežnika napačno določala čas neaktivnosti klientov, zato je strežnik pogosto prekinil povezavo do klienta. Podobni problemi so se pojavljali še pri drugih delih strežnika, kjer so bile določene časovne omejitve.

Pri prekinitvi povezave med klientom in strežnikom se je klient v nekaj sekundah avtomatično povezal na drugi strežnik in normalno nadaljeval z uporabniško sejo. Napak ni bilo opaziti.

Pri prekinitvi povezave med obema strežnikoma (a ne tudi med njunimi klienti) sta po pričakovanju nastali dve ločeni omrežji (angl. *netsplit*). Delovanje sistema ni bilo okrnjeno, vsak izmed strežnikov je po določenem času odstranil kliente, do katerih ni imel neposrednih povezav.

4.5.1. Periodična sinhronizacija ur na strežnikih s pomočjo *crontab* tabele

Problem s sinhronizacijo ur sem rešil tako, da sem v *crontab* nastavil periodično sinhronizacijo ur na obeh strežnikih preko NTP protokola (angl. *Network Time Protocol*) s strežnikom *ntp1.arnes.si*. *Crontab* je tabela nalog, ki jih operacijski sistem izvaja po navedenem urniku [30].

Tabela *crontab* za uporabnika *root* na strežniku Ž:

```
root@s1:~# crontab -l
*/20     *       *       *       *       rdate -s ntp1.arnes.si
```

Prvih pet številk označuje periodo (minuta, ura, dan v mesecu, mesec, dan v tednu), sledi jim ukaz. Zvezdica (*) pomeni, da je pogoj izpolnjen za katerokoli vrednost datuma in ure posameznega stolpca. Številka pomeni, da se bo ukaz izvedel, ko bo datum ali ura enaka vrednosti tistega stolpca. Periodo lahko označimo tudi z zvezdico in poševnico, tako da vrednost na primer */20 pri stolpcu za minute pomeni, da se bo ukaz izvedel vsakih 20 minut. Vrednost na primer 20 pri minutah pomeni, da se bo ukaz izvedel vsakih 20 minut čez polno uro, torej ob 0:20, 1:20, 2:20, 3:20, 4:20 in tako naprej. Če želimo, da se ukaz izvede ob polnoči vsako novo leto, v tabelo *crontab* vnesemo takšne vrednosti:

```
0     0       1       12       *       /usr/bin/newyear
```

Napake pri vtičnem strežniku so se pojavljale tudi pri združitvi podatkov o povezanih klientih. Izkazalo se je, da PHP-jeva funkcija `serialize()`, ki zakodira kakršenkoli

podatek v niz (angl. *string*), ne ohranja prenosljivosti podatkov za resurse, kot so vtičniki, kazalci na datoteke, itd. Pri združitvi klientov je potrebno začasno shraniti resurse vtičnikov do klientov ter jih po združitvi ponovno nastaviti, sicer se izgubi povezava do klienta, čeprav je poslani resurs isti. Resursi so v PHP posebni podatkovni tipi, ki jih ni mogoče uvažati ali izvažati izven tekoče skripte.

Pri vtičnem strežniku so se pri združitvi podatkov o povezanih klientih in sobah včasih podatki odrezali, ker so bili daljši od velikosti medpomnilnika. Problem sem rešil z implementacijo dinamičnega medpomnilnika v PHP, opisanega v poglavju 3.2.

Slika 14: Portal Briskula.si.

5. ZAKLJUČEK

Ob zaključku diplomske naloge se oziram na delovanje izdelanega sistema kot celoto, grobo oceno stabilnosti, konsistentnost in varnost podatkov ob izpadu enega izmed strežnikov.

Iz pridobljenih rezultatov testiranja lahko sklepam, kje ima sistem dodatne možnosti za izboljšave ter ali je primeren za uporabo in pod kakšnimi pogoji.

5.1. Rezultati testiranja

Med izdelavo diplomske naloge sem na internetu večkrat na naletel na nasvet, naj se ne piše na oba *master* strežnika pri asinhroni replikaciji, ki se ga sprva nisem držal, ker sem menil, da sem aplikacijo pripravil na tak način, da sem izločil vse možnosti napak, ki bi se lahko pojavile pri replikaciji. Nasvet se je izkazal za zelo dobrega, saj je največ napak pri celotnem projektu nastalo ravno zaradi kolizije podatkov pri replikaciji podatkovnih baz.

Zaradi kolizije je prišlo do nekonsistentnih podatkov med strežnikoma, kakor tudi do izgube podatkov, takrat ko je ena poizvedba prepisala drugo.

Z aktivno-pasivno replikacijo ni prihajalo do kolizij (vendar jih tak način v celoti ne izključuje), aplikacija pa je ob izpadu izvedla preklop na drugi strežnik v nekaj sekundah. Preklop s strani uporabnika se je ravno tako izvedel relativno hitro, s preklopom povezave v ozadju pa je bilo poskrbljeno za transparentnost sistema in nemoteno uporabo storitve kljub izpadu enega strežnika.

Polovica uporabnikov lahko občuti daljši čas izpada, ko DNS strežnika kažeta na delujoči in izpadli strežnik hkrati. Do tega primera pride, ko je potrebna ročna intervencija za popraviti strežnik ali za odstraniti DNS zapis, ki kaže na nedosegljiv strežnik. Tudi ob hitri odstranitvi DNS zapisa, tako da zapis kaže samo na naslov delujočega strežnika, se zaradi načina delovanja DNS strežnikov zapis še nekaj časa ohrani, dokler rok zapisa ne poteče. Rok oziroma interval veljavnosti določa SOA zapis domene.

Pri RAID polju in predpomnilniku *repcached* nisem opazil napak.

5.2. Sklep

Sistem se je za moje zahteve izkazal za primernega. Cilj je bil doseči varnostno kopijo podatkov in redundanco ter visoko dosegljivost strežnikov. Podatki so bili replicirani med strežniki ter še podvojeni v RAID-1 polju na posameznem strežniku. Dosežena je bila torej štirikratna kopija podatkov. Seveda pa redundanca ni istega pomena kot varnostna kopija,

kajti podatkov RAID polje ne arhivira. Za varnostne kopije moramo poskrbeti sami s periodičnim arhiviranjem. Z živimi podatki se je preklop ob izpadu hitro izvedel, zato v večini primerov uporabniki niso občutili izpada, če pa so ga, so ga le za krajši čas, tako da uporaba storitve v večji meri ni bila okrnjena.

Seveda takšen sistem za profesionalne rešitve, kjer je velik poudarek na konsistentnosti podatkov in si napak ne moremo privoščiti, ni priporočljiv, saj lahko pride tudi do izgube podatkov.

Sistem je primeren za srednje obsežne sisteme s toleranco do manjših napak pri repliciranju. Rešitev menim, da je primerna za začetna podjetja, ki za najeto gostovanje nimajo dovolj začetnih sredstev, ukvarjajo pa se z izdelavo spletnih portalov.

5.3. Možnosti za izboljšave

Najbolj očitna pomanjkljivost v celotnem sistemu je seveda zagotavljanje visoke dosegljivosti in porazdelitev obremenitve na lokalnem nivoju, kar pa je v nasprotju z začetnimi pogoji in ekonomskimi omejitvami.

Zaradi mehanike DNS strežnikov se lahko pojavi navidezno daljši čas izpada, zato bi lahko ta del sistema izboljšali tako, da se zmanjša interval veljavnosti za zapise in periodično avtomatično preverjanje dosegljivosti naslovov, ki so navedeni v DNS zapisih. Če strežnik ni dosegljiv, se njegov naslov ne sme vračati v DNS poizvedbah.

PHP seje se po privzeti konfiguraciji shranjujejo v datoteke v mapi `/tmp`. Z vzpostavitvijo repliciranega predpomnilnika bi lahko PHP konfigurirali tako, da se seje shranjujejo v predpomnilnik in bi bile tako dostopne na vseh strežnikih. Sejo v izdelanem sistemu sicer obnavljam s sistemom piškotkov (angl. *cookies*). Piškotki so manjši deli informacij, ki si jih uporabnikov brskalnik shrani in na zahtevo posreduje strežniku. Uporaba piškotkov za nekatere primere ni najbolj varna rešitev, ker jih lahko uporabniki spreminjajo. Kjer uporaba piškotkov za ponovno inicializacijo seje (na primer ohranjanje vsebine košarice v spletni trgovini) na drugem strežniku ni možna, je rešitev replicirana PHP seja v predpomnilniku.

Konfiguracijska datoteka PHP se nahaja v `/etc/php5/apache2/php.ini`, v kateri lahko spremenimo vrednosti spremenljivk `session.save_handler` in `session.save_path`.

```
session.save_handler = memcache
session.save_path = "tcp://10.0.0.1:11311, tcp://10.0.0.2:11311"
```

Razdeljeno pisanje podatkov v podatkovno bazo bi zmanjšalo čas izvajanja aplikacije na strežniku, ki ne piše podatkov v lokalno podatkovno bazo, vendar je potrebno raziskati možne rešitve razdeljenega pisanja, ki rešujejo problem nekonsistentnosti podatkov.

Vredno bi bilo testirati stabilnost rešitve *MySQL-mmm*, ki na generičen način omogoča aktivno-pasiven mehanizem *master-master* replikacije, zato da za preklop ob izpadu ni potrebno reševati na aplikacijskem nivoju.

5.4. Naučeno pri projektu

Med izdelavo diplomske naloge sem nadgrajeval znanje, ki sem ga pridobil med študijskim ter praktičnim izobraževanjem, predvsem o možnostih visoke dosegljivosti pri sistemu za upravljanje podatkovnih baz *MySQL*.

Opazil sem tudi velik vpliv na hitrost izvajanja aplikacij pri uporabi predpomnilnika. V predpomnilnik sem večinoma shranjeval rezultate kompleksnih in pogostih poizvedb na podatkovnemu strežniku. S primerno uporabo predpomnilnika se zelo zmanjša obremenitev strežnika, če v predpomnilnik shranjujemo podatke, ki se večkrat nepotrebno izračunavajo in povečujejo obremenitev na strežniških resursih.

Pri programiranju vtičnega strežnika sem pridobil veliko izkušenj iz programiranja vtičnikov v PHP ter distribucije podatkov med neposrednimi in posrednimi povezavami v topologiji.

Predvsem sem pa prišel do ugotovitve, da je poglavje o visoki dosegljivosti nezaključeno, možnih konfiguracij je veliko in rešitve se prilagajajo na zahteve, pričakovanja in okoliščine.

6. VIRI IN LITERATURA

[1] (2006) An Introduction to RAID. Dostopno na: http://www.ecs.umass.edu/ece/koren/architecture/Raid/intro.html

[2] (2006) Basic RAID Organizations. Dostopno na: http://www.ecs.umass.edu/ece/koren/architecture/Raid/basicRAID.html

[3] (2011) FakeRaidHowto - Community Ubuntu Documentation. Dostopno na: https://help.ubuntu.com/community/FakeRaidHowto

[4] (2011) Memcached - a distributed memory object caching system. Dostopno na: http://memcached.org/

[5] (2009) Repcached - add data replication feature to memcached. Dostopno na: http://repcached.lab.klab.org/

[6] (2011) About Memcached. Dostopno na: http://memcached.org/about

[7] (2009) KnowThyUbuntu - Community Ubuntu Documentation. Dostopno na: https://help.ubuntu.com/community/KnowThyUbuntu#The%20Init%20Process

[8] (2011) Configuring repcached service on Debian/Ubuntu. Dostopno na: http://dev.kafol.net/2011/03/configuring-repcached-service-on.html

[9] (2008) Update-rc.d – LQWiki. Dostopno na: http://wiki.linuxquestions.org/wiki/Update-rc.d

[10] (2008) MySQL :: MySQL 5.0 Reference Manual :: 14 High Availability and Scalability. Dostopno na: http://dev.mysql.com/doc/refman/5.0/en/ha-overview.html

[11] (2011) MySQL :: MySQL 5.0 Reference Manual :: 5.2.3 The Binary Log. Dostopno na: http://dev.mysql.com/doc/refman/5.0/en/binary-log.html

[12] (2011) MySQL :: MySQL 5.0 Reference Manual :: 15.2.2.1 The Slave Relay Log. Dostopno na: http://dev.mysql.com/doc/refman/5.0/en/slave-logs-relaylog.html

[13] (2007) MySQL Forums :: Replication :: Ring multi-master replication. Dostopno na: http://forums.mysql.com/read.php?26,141609,148420#msg-148420

[14] (2005) MySQL Forums :: Replication :: Slave Replication Error. Dostopno na: http://forums.mysql.com/read.php?26,24736,24884

[15] (2011) MySQL :: MySQL 5.0 Reference Manual :: 15.1.2.3 Replication Slave Options and Variables. Dostopno na: http://dev.mysql.com/doc/refman/5.0/en/replication-options-slave.html

[16] (2011) Split-Site R2 configurations for Business Continuity. Dostopno na: http://www.linux-ha.org/SplitSite

[17] (2009) Multi-Master Replication Manager for MySQL. Dostopno na: http://mysql-mmm.org/start

[18] (2010) Changelog for MMM 2.x. Dostopno na: http://mysql-mmm.org/mmm2:changelog

[19] (2011) DNS basics. Dostopno na: http://uits.iu.edu/page/adns
[20] (2001) BIND Configuration File. Dostopno na. http://www.isi.edu/~bmanning/v6DNS.html#named.conf
[21] (2008) Information Technology Services: NetDB. Dostopno na: http://www.stanford.edu/services/netdb/
[22] (2001) MX Record Explanation Dostopno na: http://dnsdb.cit.cornell.edu/explain_mx.html
[23] (2011) PHP: General Information – Manual. Dostopno na: http://si2.php.net/manual/en/faq.general.php
[24] (2011) Download Ubuntu Server. Dostopno na: http://www.ubuntu.com/download/server/download
[25] (2009) PHP: socket_set_nonblock – Manual. Dostopno na: http://www.php.net/manual/en/function.socket-set-nonblock.php
[26] (2011) MySQL :: The world's most popular open source database. Dostopno na: http://www.mysql.com/
[27] (2011) Client Server Model. Dostopno na: http://www.tutorialspoint.com/unix_sockets/client_server_model.htm
[28] (2003) PHP: gzdeflate – Manual. Dostopno na: http://www.php.net/manual/en/function.gzdeflate.php
[29] (2011) What is Ajax? Dostopno na: http://www.wrox.com/WileyCDA/Section/id-303217.html
[30] (2002) The crontab Command And File Syntax. Dostopno na: http://content.hccfl.edu/pollock/unix/crontab.htm
[31] (2011) Briskula.si. Dostopno na: http://www.briskula.si
[32] (2011) MySQL :: MySQL 5.0 Reference Manual :: 15.1.2.4 Binary Log Options and Variables. Dostopno na: http://dev.mysql.com/doc/refman/5.0/en/replication-options-binary-log.html
[33] (2007) The Structure of a DNS SOA Record. Dostopno na: http://support.microsoft.com/kb/163971

KAZALO SLIK