Analiza zmogljivosti oblačnih in strežniških storitev

Uredil prof. dr. Miha Mraz

Maj 2016

Kazalo

Predgovor

Pričujoče delo je razdeljeno v deset poglavij, ki predstavljajo analize zmogljivosti nekaterih tipičnih strežniških in oblačnih izvedenk računalniških sistemov in njihovih storitev. Avtorji posameznih poglavij so slušatelji predmeta *Zanesljivost in zmogljivost računalniških sistemov*, ki se je v štud.letu 2015/2016 predaval na 1. stopnji univerzitetnega študija računalništva in informatike na Fakulteti za računalništvo in informatiko Univerze v Ljubljani. Vsem študentom se zahvaljujem za izkazani trud, ki so ga vložili v svoje prispevke.

prof. dr. Miha Mraz, Ljubljana, v maju 2016

Poglavje 1

Analiza zmogljivosti oblačnih storitev različnih ponudnikov

Timotej Osolin, Aleš Travnikar, Matej Dolenc

1.1 Opis problema

Oblačne storitve so na voljo že kar nekaj časa, kljub temu pa se njihova popularnost ne zmanjšuje. Potrebe po računskih virih prav tako z velikostjo problemov naraščajo [1]. Velika podjetja (Amazon, Microsoft, Google) že dolgo časa ponujajo oblačne storitve, obstaja pa tudi množica drugih ponudnikov (Digital Ocean, V2 Cloud, Elastic Host). Za končnega uporabnika to predstavlja po eni strani veliko različnih možnosti, a po drugi dilemo pri izbiri ustreznega ponudnika.

1.2 Namen

Namen pričujočega poglavja je analizirati zmogljivost in zanesljivost določenih oblačnih storitev na podlagi različnih metrik kot so čas in hitrost dostopa do podatkov, hitrost prenosa datotek, hitrost bralno/pisalnih operacij, hitrost procesiranja zahtev itd.

Na podlagi opravljenih meritev želimo pokazati prednosti in slabosti posameznega ponudnika oblačnih storitev, ki bi končnemu uporabniku pomagali pri izbiri, planiranju in iskanju anomalij v delovanju storitev.

1.3 Izbira ponudnikov

V pričujočem razdelku so na kratko opisani ponudniki oblačnih storitev, katere smo želeli testirati in primerjati med seboj.

1.3.1 Digital Ocean

Digital Ocean je bila naša prva izbira za testiranje [2]. Ponuja oblačno infrastrukturo, ki bazira na Linuxovih distribucijah kot sta Ubuntu in CentOS. Izkoristili smo naročnino za študente, pri kateri smo za 5$ dobili 100$ kredita, ki ga lahko porabimo na storitvah DigitalOcean. Naročnina vsebuje uporabo Linux operacijskih sistemov (Ubuntu, FreeBSD, Fedora, Debian, CoreOS, CentOS), 40GB pomnilnega prostora na SSD disku za hranjenje podatkov, 2GB RAM, 2 centralno procesni enoti na Intel Xeon procesorju ter omejitev prenosa podatkov na 3TB.

1.3.2 Amazon Web Services

Za drugega ponudnika smo izbrali Amazon Web Services [3]. Amazon omogoča enoletno brezplačno uporabo njihovih oblačnih storitev. To vključuje uporabo Linux ali Windows platforme, uporabo shrambnih storitev, podatkovne baze, itd. Pri Amazonu smo izkoristili ponudbo AWS Free Tier, kjer lahko v določenih mejah 12 mesecev uporabljamo naslednje storitve: 750 ur mesečne uporabe Linux operacijskega sistema ali 750 ur mesečne uporabe Windows operacijskega sistema, pri čemer lahko uporabljamo eno instanco cel mesec ali dve instanci, vsako pol meseca. Strojna oprema sestoji iz Intel Xeon procesorja s taktom 3.3 GHz in 1GB RAMa. Za shrambo dobimo 5GB prostora v oblaku, kjer smo omejeni na 20000 GET (zahteve, s katerimi pridobimo določen objekt iz oblačne shrambe) in 2000 PUT zahtev (zahteve, s katerimi določen objekt dodamo v oblačno shrambo).

1.3.3 Google Cloud Platform

Naša zadnja izbira je bila Google Cloud Platform [4]. Omogočena nam je 60 dnevna brezplačna uporaba, polega tega pa dobimo še 300$ kredita, ki ga lahko poljubno porabimo na Google Cloud storitvah. Prav tako ponuja široko ponudbo uporabe programskih jezikov in podatkovnih baz. V naročnini je prav tako vsebovana uporaba Windows in Linux operacijskih sistemov. V preizkusni dobi smo imeli na voljo 8 računalnikov, pri čemer je imel vsak eno centralno procesno enoto na procesorju Intel Xeon serije E5 (2.6 GHz Intel Xeon E5 (Sandy Bridge), 2.5 GHz Intel Xeon E5 v2 (Ivy Bridge) ali 2.3 GHz Intel Xeon E5 v3 (Haswell)) z 3.75GB RAMa.

1.4 Izbira tehnologij

V tem razdelku so na kratko opisane vse izbrane tehnologije, ki smo jih potrebovali za našo analizo.

1.4.1 PerfKit Benchmarker

PerfKit Benchmarker je odprtokodno orodje, ki je namenjeno merjenju zmogljivosti in primerjanju oblačnih storitev med seboj [5]. Podpira testiranje množice ponudnikov, med njimi vse naše izbrane kandidate kot tudi Microsoft Azure, Rackspace, OpenStack, itd. Omogoča merjenje hitrosti dostopa do storitev, merjenje latence, prepustnosti, ponuja pa tudi teste procesiranja in meritve bralno pisalnih operacij.

1.4.2 Python

Testno ogrodje, ki sestavlja PerfKit Benchmarker, je spisano v programskem jeziku Python [6]. Ogrodje ponuja veliko število v naprej spisanih standardiziranih testov, kot tudi testov prilagojenih za določene ponudnike. Poleg tega omogoča hitro in enostavno razširljivost, saj je možno preprosto spreminjanje že obstoječih, kot tudi kreiranje lastnih testnih skript.

1.4.3 Odjemalec

Odjemalca predstavlja Ubuntu Server, ki velja za enega najbolj razširjenih Linux distribucij. Testni strežnik je priklopljen na širokopasovno povezavo v internet in predstavlja enakopravnega odjemalca za vse tri ponudnike oblačnih storitev. Na njem bo teklo orodje PerfKit Benchmarker, kjer bodo nastavljene povezave do vseh treh ponudnikov, kar zadeva tudi avtentikacijo in sinhronizacijo s samimi strežniškimi napravami v oblaku.

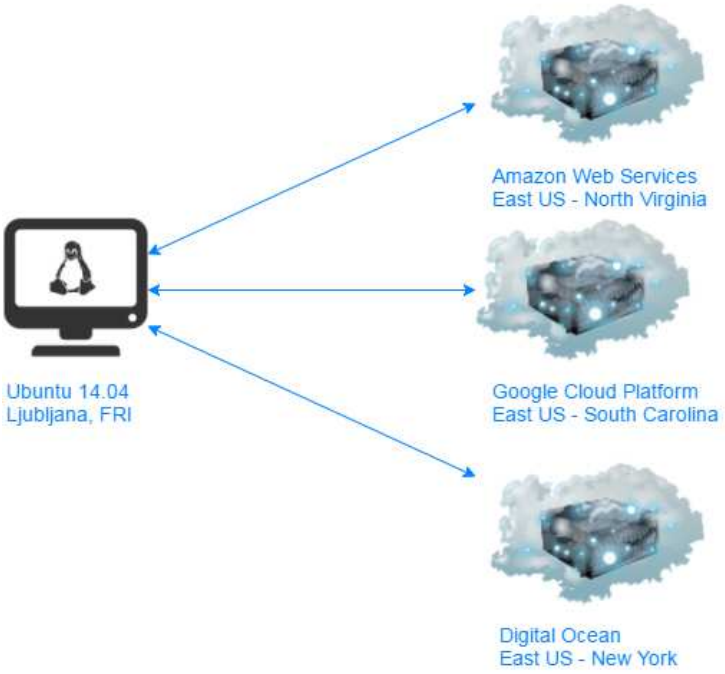
Na sliki 1.1 je prikazana shema oblačnega sistema. Sestavljena je iz uporabnika, ki dostopa do oblaka, ki nudi storitve glede na njegove zahteve. Na strani uporabnika namestimo PerfKit Benchmarker, s katerim bomo nato izvajali in simulirali zmogljivostne teste za oblak.

1.5 Izbrane metrike

V pričujočem poglavju so opisane izbrane metrike za merjenje zmogljivosti oblačnega sistema.

1.5.1 Prepustnost omrežja

Prepustnost omrežja je eden ključnih dejavnikov današnjega komunikacijskega sveta. Vse več podatkov se prenaša preko omrežja. Potreba po pomnilnih prenosnih medijih, s katerimi bi prenašali podatke, praktično izginja. Tako kot



Slika 1.1: Shema testnega sistema.

v vsakdanjem življenju, je tudi pri oblačnih storitvah seveda prenos podatkov preko omrežja skorajda edina možnost. Posledično gre za zelo zanimivo metriko s stališča končnih uporabnikov. Vsakogar zanima kako hitro bo lahko prenesel svojega podatke iz točke A do točke B v omrežju.

1.5.2 Hitrost procesiranja

Tudi hitrost procesiranja ima velik pomen, saj danes na računalniku ne teče več samo en proces oziroma aplikacija, ampak je teh aplikacij mnogo in se morajo zato boriti za procesorski čas. Končni uporabnik želi vedno imeti občutek tekočega delovanja sistema, kljub temu da hkrati uporablja več aplikacij ali servisov. Zmogljivost procesne enote (CPE) je tako zelo pomembna, saj omogoča hitro procesiranje zahtev in tako možnost delovanja tudi ostalih aplikacij na sistemu. Če bi recimo ena operacija na procesorju porabila preveč časa, bi posledično ostali procesi obstali in bi uporabnik dobil občutek, da sistem ne deluje najbolje.

1.5.3 Zmogljivost datotečnega sistema

Zmogljivost datotečnega sistema je izrednega pomena pri prenosu podatkov, bodisi da gre za prenos preko omrežja ali lokalni prenos med različnimi shranjevalnimi mediji. To ne zajema zgolj ročnega kopiranja datotek, ampak tudi vse interne prenose podatkov, ki jih opravljajo aplikacije. Gre za enega izmed kriterijev, ki je tudi zelo zanimiv za končnega uporabnika, saj predstavlja operacije s katerimi se uporabniki srečujejo dnevno.

1.6 Opis bremena

V pričujočem razdelku so na kratko opisana vsa bremena, ki smo jih uporabljali pri testiranju.

1.6.1 Iperf

Prepustnost omrežja za promet po protokolarnem skladu TCP/IP lahko enostavno izmerimo z orodjem *Iperf* [7]. *Iperf* deluje kot običajna TCP storitev, in sicer na eni strani povezave je strežnik, na drugi pa odjemalec. Odjemalec generira in pošilja zahteve, ki predstavljajo breme, na drugi strani pa je strežnik v vlogi prejemnika zahtev. *Iperf* izmeri količino prenesenih podatkov in na podlagi znanega trajanja meritve izračuna hitrost prenosa podatkov. Rezultat torej predstavlja hitrost prenosa podatkov v bitih na sekundo (bps). Pri velikih hitrostih povezav in daljših časih potovanja paketov med strežnikom in odjemalcem je zelo pomembna velikost okna v TCP/IP skladu (angl. *TCP buffer size*). V lokalnem ethernet omrežju so časi potovanja paketov zelo kratki, zato običajna velikost okna za TCP povsem zadošča. Priporočena velikost okna je enaka produktu hitrosti povezave in časa potovanja paketa (angl. *round trip*

time). Privzeta velikost je 32 KB. Na primer v lokalnem 100 Mb/s omrežju z zakasnitvijo 1 ms zadošča že okno veliko 16 KB.

1.6.2 CoreMark

CoreMark [8] je preprosto testno orodje za testiranje centralno procesnih enot (CPE). Njegova uporaba je zelo razširjena in ima možnost, da postane del standardne specifikacije, vsaj kar se tiče osnovnega testiranja hitrosti procesiranja. Je neodvisen od procesorske platforme in tako omogoča primerjavo različnih platform med sabo. Rezultat testa predstavlja desetiška vrednost, ki je seštevek rezultatov vseh internih testov, ki jih uporablja *CoreMark*, kar omogoča hitro in učinkovito primerjavo rezultatov. Program je napisan v programskem jeziku C in breme gradi na implementacijah splošno znanih algoritmov: iskanje in urejanje povezanih seznamov (uporaba kazalcev v pomnilniku), osnovne matrične matematične operacije, 'state machine' (za simulacijo različnih protokolov) in CRC (angl. *cyclic redundancy check*, ki se uporablja v množici protokolov predvsem za odkrivanje in popravljanje napak v podatkih). S pomočjo omenjenih bremen nam test prikaže zmogljivost procesorskega sistema iz perspektive zmogljivosti podatkovnih vodil v sistemu, hitrosti pomnilnika in obstoječega predpomnilnika ter hitrosti procesiranja računskih operacij.

1.6.3 FIO

Fio (angl. *Flexible IO*) predstavlja orodje, ki generira breme, nato pa s pomočjo tega bremena testira in analizira delovanje datotečnega sistema [9]. Gre za lokalni transport podatkov in ne za transport podatkov preko omrežja. *Fio* sestavljata dva dela: prvi del je definiranje bremena, kjer uporabnik definira ali se bo testiralo branje ali pisanje, število poslov, velikost posameznega posla ter velikost posameznega bloka za vhodno/izhodno operacijo. Drugi del predstavlja testiranje. Po uspešnem zaključku testa uporabnik pridobi informacije o hitrosti branja in pisanja, minimalno, maksimalno in povprečno latenco pri branju ali pisanju posameznega bloka, število iops (angl. *input/output operations per second*), itd.

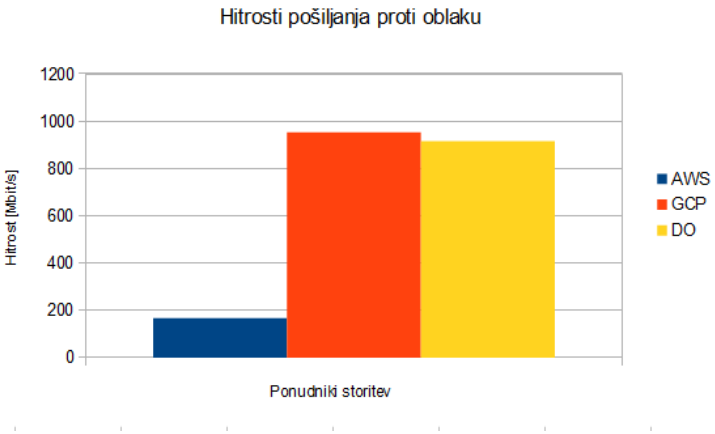
1.7 Rezultati meritev

V tem razdelku so opisani postopki izvajanja testov in analize rezultatov, ki so bili pridobljeni po izvedenih testih. Vse teste smo izvajali na operacijskem sistemu Ubuntu 14.04. Teste smo pognali sredi tedna, torej v sredo, izvedli pa smo jih 24-krat - na vsako polno uro. Pri vsakem tipu meritev je tako podana tudi tabela z osnovnimi statističnimi postavkami - minimum, maksimum, povprečje ter standardna deviacija meritev. Pri orodju *Fio* je testov razmeroma veliko - skupno več kot 700 - zatorej smo se odločili, da v statistični analizi predstavimo le dve, za uporabnika morda najpomembnejši metriki - to sta bralna in pisalna hitrost.

1.7.1 Testiranje prepustnosti omrežja

Za testiranje prepustnosti omrežja smo uporabili orodje *Iperf*. Breme pri testiranju prepustnosti omrežja predstavljajo zahteve za prenos datotek. Datoteke se pošiljajo od oblaka proti uporabniku in od uporabnika proti oblaku. Na podlagi prenesene količine datotek (v MB) in pretečenega časa izračunamo povprečno hitrost prenosa po omrežju. Virtualne računalnike smo locirali na vzhodni obali ZDA (East US). Pri izvajanju testa smo merili internetno povezavo med entitetami v omrežju. Ena izmed entitet je služila kot oblačna storitev, druga pa kot uporabnik, ki do storitve dostopa. Test smo izvedli obojesmerno (torej od oblaka proti uporabniku in obratno). *Iperf* test se je v vsakem primeru izvajal 60 sekund.

Hitrosti pošiljanja podatkov proti oblaku posameznega ponudnika oblačne storitve so vidne na sliki 1.2. Občutno najmanjšo zmogljivost prenosa smo odkrili pri ponudniku Amazon (AWS), kjer je bila povprečna hitrost prenosa pod 200 Mbit/s. Največjo povprečno hitrost prenosa smo izmerili pri ponudniku Google (GCP), ki je le za odtenek premagal rešitev Digital Ocean (DO). Obe vrednosti presegata hitrost 900 MBit/s in GCP se zelo približa meji 1 Gbit/s. Pri GCP, kot tudi pri DO se je tako izkazalo, da imata več kot 4x večjo povprečno hitrost nalaganja datotek v oblak, v primerjavi s ponudnikom AWS.



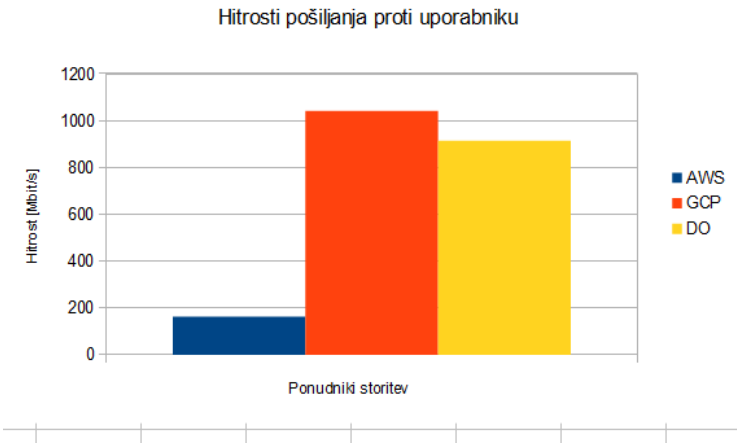
Slika 1.2: Povprečna hitrost pri prenosu podatkov proti oblaku (angl. *upload speed*).

Na sliki 1.3 so razvidni osnovni statistični podatki za meritve *Iperf - upload*. Do največjih odstopanj med minimalno in maksimalno vrednostjo je prišlo pri ponudniku AWS, čeprav ima daleč najslabši povprečni rezultat. Najbolj stabilno in povprečno tudi najboljše se je odrezal GCP.

Enota:MB/s	AWS	DO	GCP
MIN	118	440	926
MAX	914	938	1201
AVG	255	869	1035
STDEV	226	125	79

Slika 1.3: Statistična analiza rezultatov *Iperf - upload*.

Hitrosti pošiljanja podatkov proti uporabniku so vidne na sliki 1.4. Zgodba je skoraj da identična, kot pri hitrostih prenosa v obratni smeri. Občutno najhitrejša sta zopet GCP in DO, daleč zadaj pa je ponovno AWS. Pri primerjavi hitrosti se je izkazalo, da je razmerje zmogljivosti praktično enako, kot v prejšnjem primeru.



Slika 1.4: Povprečna hitrost pri prenosu podatkov proti uporabniku (angl. *download speed*).

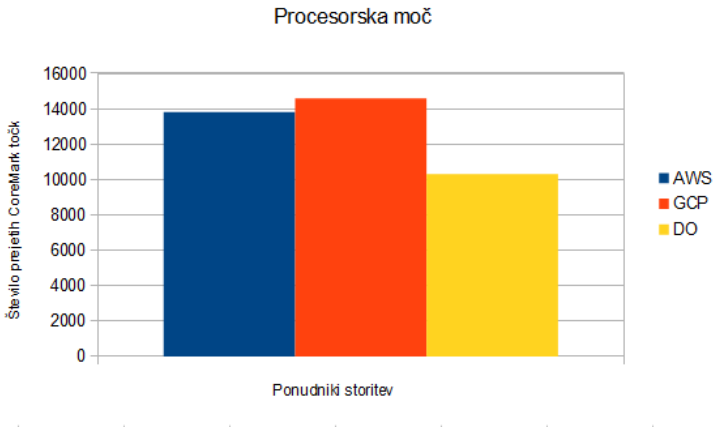
Tudi statistična analiza *Iperf - download* testa, vidna na sliki 1.5, prikazuje podobne rezultate. Do največje standardne deviacije rezultatov je prišlo pri ponudniku AWS. DO in GCP pa sta pokazala precej boljše in bolj stabilne rezultate.

Enota: MB/s	AWS	DO	GCP
MIN	113	473	598
MAX	913	938	1236
AVG	281	888	972
STDEV	229	95	122

Slika 1.5: Statistična analiza rezultatov *Iperf - download*.

1.7.2 Testiranje hitrosti procesiranja

Procesorsko moč smo merili z orodjem *CoreMark*. Breme pri testiranju hitrosti procesiranja predstavljajo splošni algoritmi kot so npr. iskanje in urejanje po seznamih, osnovne matrične operacije ter izvajanje CRC. Ker pri nobenem ponudniku nismo imeli izbire, kateri procesor bomo imeli na razpolago, je ta test zanimiv predvsem iz stališča procesorsko bolj zahtevnih aplikacij oziroma uporabnikov. Kot je razvidno iz slike 1.6, je najbolj zmogljiv procesor na voljo pri ponudniku Google Cloud, pri katerem rezultat testa presega 14.000 točk. Tesno mu sledi Amazon Web Services, z le nekaj manj kot 14.000 točkami, medtem ko je procesor pri DigitalOcean-u nekoliko manj zmogljiv, in se s skoraj 4000 tockami manj uvršča v *CoreMark* rang 10.000 točk. Pri vseh ponudnikih smo pazili, da smo izbrali takšne storitve, ki niso trenutno deljene med druge uporabnike (torej imamo pri testiranju na voljo vso procesorsko moč; to nam zagotavlja ponudnik). Obstajajo namreč tudi rešitve, kjer je procesorska moč deljena med mnoge uporabnike hkrati, imamo pa le zagotovilo, da imamo v nekem trenutku vedno na razpolago nek odstotek skupne procesorske moči (npr. 10%).



Slika 1.6: Število prejetih *CoreMark* točk.

Statistični podatki, ki izhajajo iz *CoreMark* testa zmogljivosti procesorskega sistema so vidni na sliki 1.7. Povprečna rezultata AWS in GCP sta podobna in tudi standardna deviacija je skoraj enaka. Pri testu DO je prišlo do anomalije in sicer se je pri eni izmed ponovitev testa zgodila napaka, ki je privedla do rezultata 0. Ta rezultat je delno pokvaril vrednosti ostalih statističnih kazalcev.

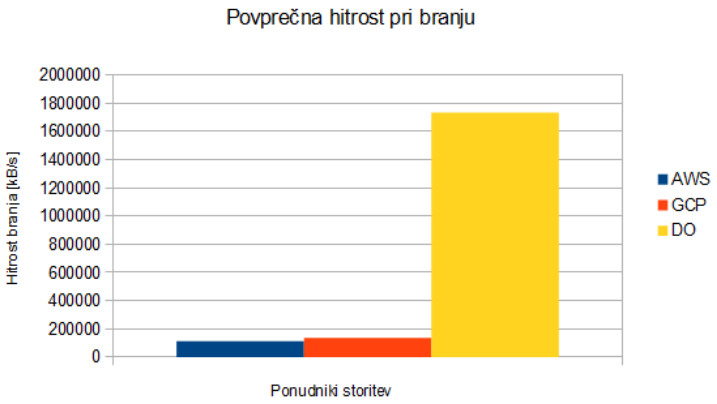
Enota: / (coremark)	AWS	DO	GCP
MIN	13065	0	13549
MAX	14479	12516	15105
AVG	13728	8953	14521
STDEV	376	3904	374

Slika 1.7: Statistična analiza rezultatov *CoreMark*.

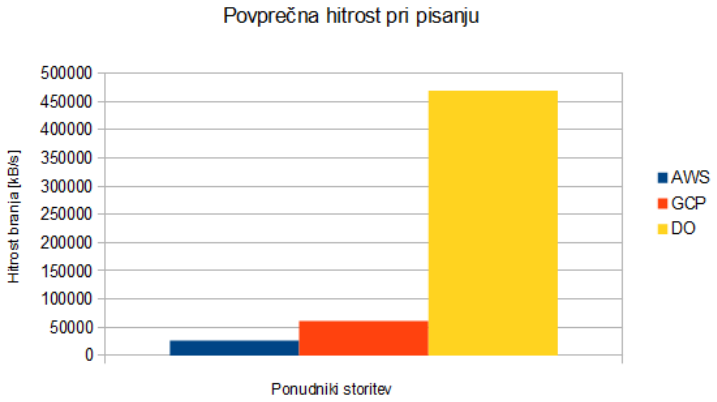
1.7.3 Testiranje zmogljivosti datotečnega sistema

Pri testiranju datotečnega sistema smo se poslužili orodja *Fio*. Preden smo teste izvedli, se je bilo potrebno odločiti za posamezne parametre (definiranje bremena). Breme je definirano v *.job* datoteki, kjer se definirajo parametri kot so velikost bremena samega, velikost posameznega bloka, IO tip (ali gre za bralno ali pisalno operacijo), število datotek (čez koliko datotek je breme razporejeno), itd. Velikost bloka smo nastavili na 512KB. Pri testiranju smo zahtevali izvedbo enega posla, ki smo mu nastavili različne velikosti. Kot opombo bi vnaprej omenili, da platformi Amazon Web Services (AWS) in Google Cloud (GCP) uporabljata standardne trde diske, pri platformi Digital Ocean (DO) pa smo imeli na voljo SSD diske.

Na slikah 1.8 in 1.9 so prikazani rezultati za povprečno hitrost pri bralnih ter pisalnih operacijah. Platforma Digital Ocean ima pričakovano največjo hitrost branja in pisanja, saj teče na SSD diskih. Bolj zanimiva je primerjava platform AWS in GCP. Povprečni hitrosti branja sta približno enaki. Razlika se pojavi pri hitrosti pisanja, kjer pa ima GCP več kot 2x večjo hitrost kot AWS. DO kot že rečeno tukaj ni primerljiv, saj rešitev teče na novejši pomnilniški tehnologiji, katere bralno pisalne hitrosti daleč presegajo klasični diskovni sistem.

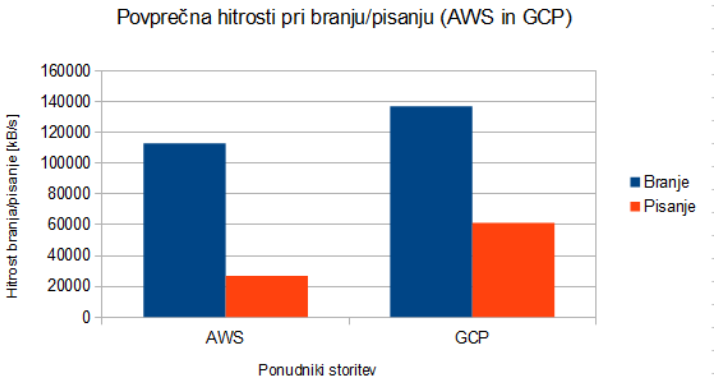


Slika 1.8: Povprečna hitrost pri branju.



Slika 1.9: Povprečna hitrost pri pisanju.

Glede na to, da platforma DigitalOcean uporablja SSD diske in ima s tem veliko prednost pred ostalima dneva ponudnikoma, je na sliki 1.10 prikazana še primerjava bralne in pisalne hitrosti samo za platformi AWS in GCP. Kot smo že omenili v prejšnjem odstavku sta bralni hitrosti približno enaki, pisalna hitrost pa je pri GCP več kot 2x večja od AWS.



Slika 1.10: Povprečna hitrost pri branju in pisanju za platformi AWS in GCP.

Na slikah 1.11 in 1.12 je prikazana statistična analiza rezultatov *Fio - branje* in *Fio - pisanje*. Rezultati AWS in GCP so občutno slabši, zaradi že prej omenjenih razlogov. Presenetljivo pa je velika standardna deviacija pri rezultatih DO, kljub temu da je povprečni rezultat občutno boljši kot pri konkurentih, so odstopanja velika. Vzroke gre iskati v SSD tehnologiji sami, ki res da ponuja višje maksimalne hitrosti, a je tudi sama hitrost bolj nepredvidljiva in odvisna od več dejavnikov.

Enota: kb/s	AWS	DO	GCP
MIN	35853	661333	102342
MAX	173730	2506105	184785
AVG	110777	1671112	142746
STDEV	28259	525425	26046

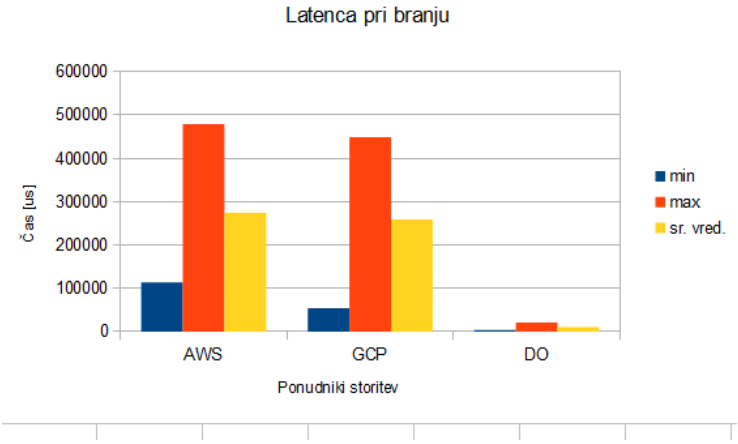
Slika 1.11: Statistična analiza rezultatov *Fio - branje*.

Enota: kb/s	AWS	DO	GCP
MIN	10546	203487	32817
MAX	69268	835368	89026
AVG	28450	475430	57928
STDEV	14418	169920	17309

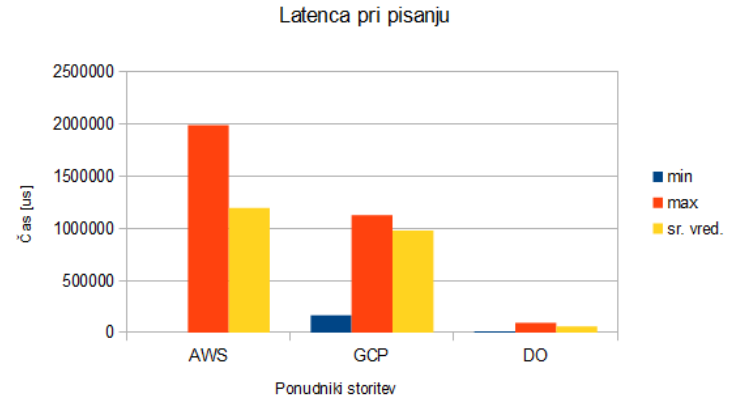
Slika 1.12: Statistična analiza rezultatov *Fio - pisanje*.

Naslednja metrika, ki smo si jo ogledali pri testiranju podatkovnega sistema

je latenca ene vhodno/izhodne operacije za posamezen blok podatkov. Rezultati so vidni na slikah 1.13 (branje) in 1.14 (pisanje). Platforma DO ima najmanjše latence, zaradi že prej omenjenega razloga (SSD disk), zanimivejši pa so rezultati latenc branja in pisanja pri platformah AWS in GCP. GCP ima sicer minimalno latenco branja precej manjšo, kot je minimalna latenca pri AWS, in tudi povprečna vrednost je nekoliko nižja. Drugače so pa rezultati latenc pri branju dokaj primerljivi. Večja razlika se pojavi pri latenci pisanja, kjer so razlike precej večje. GCP ima precej manjšo povprečno latenco branja, kar je seveda ugodneje. To je bilo pravzaprav za pričakovati, saj smo opazili tudi precej večji razkorak v samih hitrostih pisanja pri teh dveh platformah.

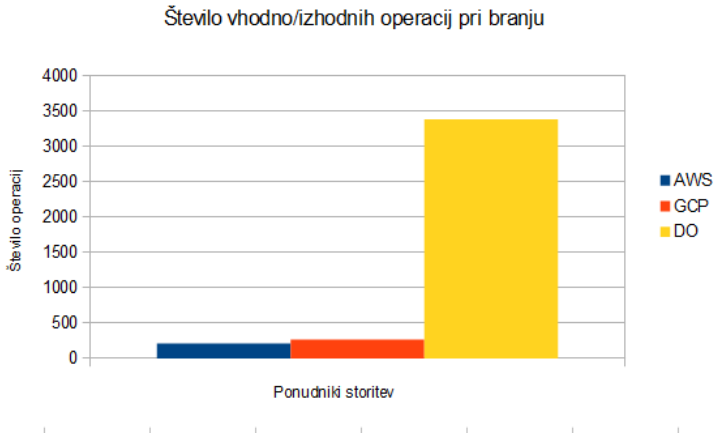


Slika 1.13: Minimalna, maksimalna in povprečna latenca pri branju enega bloka.



Slika 1.14: Minimalna, maksimalna in povprečna latenca pri pisanju enega bloka.

Za zadnjo metriko smo si izbrali število vhodno/izhodnih operacij na sekundo. Rezultati so vidni na slikah 1.15 in 1.16. Iz prejšnjih dveh analiz (hitrost in latenca) je moč pričakovati podobno razmerje moči tudi pri številu vhodno/izhodnih operacij. Platforma DO ima pričakovano največje število operacij, tako pri branju kot pri pisanju. Pri platformah AWS in GCP pa je ponovno opazna podobnost pri bralnih operacijah, pri številu pisalnih operacij pa je zopet večja razlika, in sicer GCP je v primerjavi z AWS pri pisalnih operacijah ponovno več kot 4x zmogljivejši .



Slika 1.15: Število vhodno/izhodnih operacij pri branju.

Slika 1.16: Število vhodno/izhodnih operacij pri pisanju.

1.8 Komentarji rezultatov

Katerega ponudnika torej izbrati? Kot lahko razberemo iz rezultatov, ima vsak ponudnik svoje prednosti in slabosti. Za uporabnike, ki potrebujejo veliko hitrih I/O operacij, hkrati pa niso preveč zahtevni glede zmogljivost procesorja, je DigitalOcean zagotovo prva izbira, saj novejša pomnilniška tehnologija omogoča neprimerljivo višje hitrosti bralno pisalnih operacij. Google Cloud Platform ponuja izjemno hitrost prenosa po njihovih optičnih povezavah, hkrati pa ima tudi rahlo boljšo procesorsko moč ter hitrost I/O operacij od njihovega rivala Amazon Web Services. Splošno gledano, se je od vseh najslabše odrezal prav AWS, saj v nobeni izmed kategorij ni bil najboljši. Kot že omenjeno, pa je izbira v veliki meri odvisna od samega uporabnika ter namena uporabe storitve, deloma pa tudi od drugih zunanjih dejavnikov, kot so npr. lokacija, cena, osebne preference in ostale omejitve.

Poglavje 2

Primerjava zmogljivosti oblačnih sistemov različnih ponudnikov

Urban Leben, Nataša Vodopivec, Jakob Predin

2.1 Predstavitev ideje

V današnji dobi dobivajo čedalje večjo uporabno vrednost storitve v oblaku, specifično za gostovanje in uporabo v t.i. "backend sistemih". Čeprav so le-ti sistemi pri različnih večjih ponudnikih na voljo z zelo podobnimi tehničnimi specifikacijami in s približno enako ceno mesečnega najema obstajata vprašanji ali so ti sistemi res enako zmogljivi in ali delujejo kot oglaševani. Namen naše naloge je, da analiziramo nivo ponudbe ponudnikov Amazon, DigitalOcean in Transip, izberemo primerljive cenovne in tehnične nivoje in nad njimi poženemo tako teste zmogljivosti kot tudi druge uporabniške aplikacije, da preverimo kvaliteto delovanja. Glavni fokusi naloge so oglaševana zmogljivost, cenovno-performančna analiza in analiza kvalitete omrežnega delovanja.

2.2 Izbira konfiguracije strežniškega sistema

Da je strežniški sistem študentu čim bolj dosegljiv v namene razvoja, smo za testiranje izbrali najcenejše konfiguracije, ki bi po našem mnenju še lahko služile kot "backend" za razvoj določene aplikacije. Ti sistemi vsebujejo:

- 1 CPE,
- 1GB RAM-a,
- okoli 30GB prostora za shranjevanje na SSD disku,
- lokacijo v Ameriki (če je aplikabilna pri ponudniku),
- ne presegajo cene 10 evrov na mesec.

Tako smo za testiranje izbrali sledeče strežniške sisteme pri treh ponudnikih:

1. Amazon sistem z 1 CPE, 1GB RAM-a, 30GB SSD prostora za shranjevanje, lociran na vzhodu ZDA.
2. DigitalOcean sistem z 1 CPE, 1GB RAM-a, 20GB SSD prostora za shranjevanje, lociran na vzhodu ZDA.
3. Transip sistem z 1 CPE, 1GB RAM-a, 50GB SSD prostora za shranjevanje, lociran na zahodu Evrope.

2.3 Izbira ponudnikov oblačnih storitev

Za namene našega testiranja smo ustvarili račune pri treh ponudnikih strežniškega gostovanja, specifično pri Amazonu z uporabo njihove AWS storitve, pri ponudniku DigitalOcean in pri ponudniku Transip. Pri vseh ponudnikih bomo izbrali konfiguracije, ki so podrobneje opisane v prejšnjem podpoglavju.

2.4 Izbira tehnologij za primerjavo ponudnikov

Vse naše sisteme bomo testirali na operacijskem sistemu Ubuntu Server 14.04 LTS, saj jih brez večjega dela pri namestitvi ponujajo prav vsi trije ponudniki. V sklopu seminarske naloge smo se odločili za število "stress" testov, s katerimi bi testirali ne-oglaševane performančne zmogljivosti ponujenih sistemov. Zaradi razloga testiranja oblačnih storitev smo se odločili fokusirati na tri glavne aspekte sistemov in sicer na vsesplošno zmogljivost sistema, zmogljivost posameznih komponent in omrežna zmogljivost sistema.

2.4.1 Vsesplošna zmogljivost sistema

Da lahko univerzalno primerjamo vse tri sisteme s splošnega vidika, potrebujemo vnaprej pripravljene teste in orodja, ki so glede na svoje izvajanje neodvisni od sistema. Za testiranje vsesplošne zmogljivosti smo uporabili programsko orodje Phoronix Test Suite, za posamezne komponente pa program Sysbench in orodja ping ter tracert. Po pridobljenih rezultatih smo numerično določili zmagovalca tako, da smo zmagovalcu posameznega testa pripisali eno točko, ostalima dvema pa prišteli eno točko z odšteto procentualno razliko v primerjavi

z zmogljivostjo zmagovalca. Iz rezultatov je izvzet test s pingom, saj Transip v času pisanja ne ponuja gostovanja strežniškega sistema kjerkoli v Ameriki. Ovrednoteni rezultati in seštevek točk je viden v poglavju 2.9.

2.4.2 Zmogljivosti posameznih komponent

Z zmogljivostjo posameznih komponent želimo preveriti kako zmogljive so nam ponujene komponente, ki so ponavadi ob nakupu opisane zelo bežno (npr. 8 jedrni Intel Xeon, vendar brez pripisa modela ali 30GB SSD prostora, brez omembe hitrosti zapisovanja). Od slednjega želimo testirati predvsem

- zmogljivost procesorja s testi, podobnimi tistim v realnem svetu,
- zapisovalno in bralno hitrost trdega diska,
- bralno hitrost pomnilnika.

Racionalizacija testov je, da iz njih najprej dobimo približno idejo kako zmogljive so komponente v posameznih sistemih in, da lahko kasneje na podlagi drugih meritev zmogljivosti komponent, ki jih najdemo na spletu, ugotovimo katere komponente bi se v teh sistemih lahko nahajale.

2.4.3 Omrežna zmogljivost

Ker se vsi omenjeni sistemi nahajajo na nam neznani lokaciji ponudnika, je smiselno testirati tudi omrežno zmogljivost sistema. Pri tem mislimo predvsem na odzivnost samega sistema s testom ping in testiranje s trace-route za primerjavo lokacije ter števila skokov.

2.5 Osnovno testiranje sistema

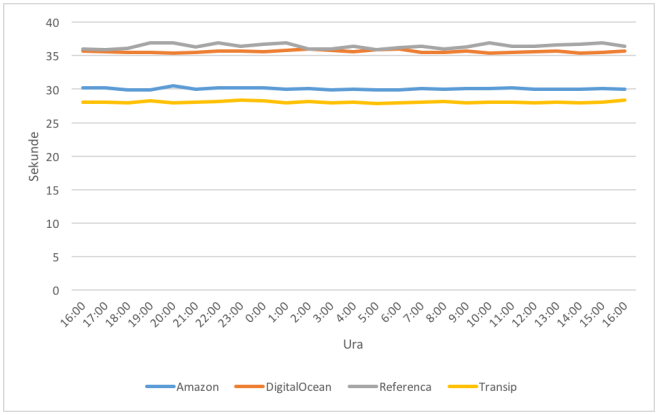
Kot **referenco** smo uporabili virtualni stroj s specifikacijami podobnimi tistim, ki so podrobneje opisane v poglavju 2.2.

Na virtualni stroj smo z orodjem VMWare Workstation naložili instanco operacijskega sistema Ubuntu Server, verzijo 14.04 LTS. Dodelili smo ji 1 CPE, 1GB RAM-a in 30GB SSD prostora za shranjevanje. Sistem, na katerem so tekli testi za referenco, vsebuje sledeče komponente (omenjene so le komponente, ki so relevantne za namene testiranja):

- Intel Core i5 4670K na frekvenci 3.7GHz,
- 8GB 1600MHz DDR3 pomnilnika,
- Samsung 840 EVO SSD,
- ASUS Maximus VI Hero matična plošča z Intlovo omrežno kartico.

2.5.1 Testiranje CPU

Za testiranje CPU smo pognali sysbench CPU test. Ta deluje na principu deljenja števil, kjer opravlja končno število operacij deljenja. Manj časa, kot je potrebovanega za izvedbo testa, bolje je.

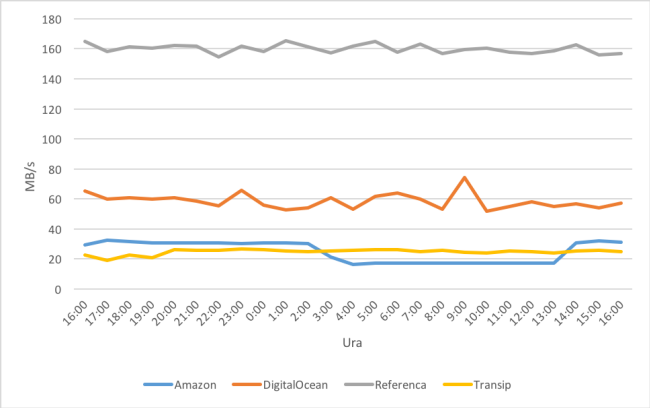


Slika 2.1: Testiranje CPU s testom sysbench. Vrednosti predstavljajo porabljen čas za izvajanje testa. Test je tekel 5. maja 2016.

Na sliki 2.1 so vidni rezultati testiranja zmogljivosti CPU skozi en dan (v našem primeru četrtek), pri čemer opažamo, da so dobljeni rezultati skozi cel dan praktično konstantni, kar je tudi smiselno, saj se ob dodelitvi procesorskega časa akcija izvede do konca tudi pri deljenih strežniških sistemih. Pri testu je zmagovalec ponudnik Transip, ki je bil od ponudnika DigitalOcean pri izvedbi testa hitrejši za več kot 15%, od ponudnika Amazon pa 7%.

2.5.2 Testiranje zapisovalne in bralne hitrosti trdega diska

Tudi za testiranje zapisovalne in bralne hitrosti smo pognali sysbench test. Najprej smo ustvarili datoteko naključnih števil, ki je bila veliko večja od našega RAM-a, saj smo se le tako izognili temu, da bi sistem uporabil predpomnilnik. Test naključno piše in bere iz diska.

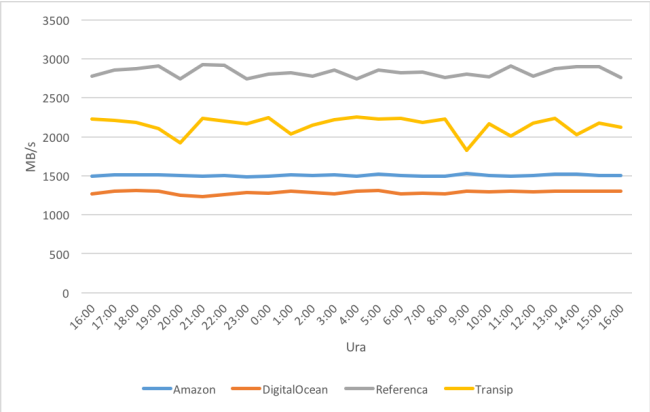


Slika 2.2: Zapisovalno-bralna hitrost trdega diska [MB/s]. Test je tekel 5. maja 2016.

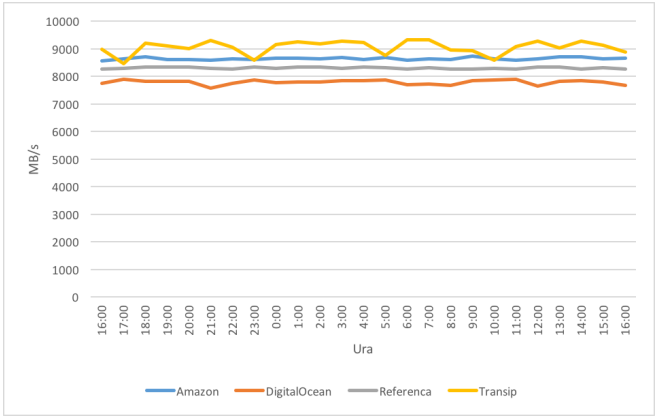
Na sliki 2.2 so vidni rezultati zmogljivosti trdega diska v stežniških sistemih skozi 24ur, kjer je DigitalOcean očiten zmagovalec kategorije. Tako od ponudnika Amazon, kot tudi ponudnika Transip je v največjem razponu hitrejši za kar za 350%. Opazili smo tudi, da prihaja do performančnih razlik glede na uro uporabe v dnevu, kjer je le ta pri npr. pri Amazonu med jutranjimi in popoldanskimi urami skoraj 50%. Opaziti je vredno tudi, da je delovanje na Amazonu kljub počasnejšemu delovanju veliko bolj konsistentno kot tisto pri DigitalOceanu.

2.5.3 Testiranje zapisovalne in bralne hitrosti pomnilnika

Z uporabo sysbench smo pisali in brali 1KB in 1MB bloke naključnih podatkov v pomnilnik strežniškega sistema.



Slika 2.3: Hitrost pomnilnika pri pisanju 1KB blokih [KB/s]. Test je tekel 5. maja 2016.



Slika 2.4: Hitrost pomnilnika pri 1MB blokih [MB/s]. Test je tekel 5. maja 2016.

Na sliki 2.3 vidimo rezultate 24 urnega testiranja 1KB blokov in na sliki 2.4 vidimo rezultate 24 urnega testiranja 1MB blokov. Na obeh slikah vidimo, da je zmagovalec v kategoriji zmogljivosti pomnilnika ponudnik Transip, ki je najhitrejši (v nekaterih primerih več kot 10% hitrejši od obeh konkurentov), vendar pa tudi z najbolj nekonsistentnimi rezultati. Razlike glede na ure lahko pripisujemo predvsem dejstvu, da ima isto pomnilniško vodilo več hkrati delujočih instanc virtualnih strojev, ki pa so, kot kažejo rezultati, boljše optimizirane pri Amazonu.

2.5.4 Testiranje odzivnosti in število skokov do strežnika

Z uporabo orodij ping in tracert smo pogledali odzivnost in število skokov. Testirali smo iz neodvisne strani, ki se nahaja v Nemčiji (saj se je tam nahajal ponudnik).

Referenca	42 ms	8 skokov
Amazon	103 ms	11 skokov
Digital Ocean	127 ms	7 skokov
Transip	32 ms	7 skokov

Tabela 2.1: Povprečna vrednost ping in število skokov. Izmerjeno 20.5.2016.

V tabeli 2.1 vidimo rezultate testiranja omrežne zmogljivosti strežniških sistemov. Jasno je, da je zaradi lokacije v zahodni Evropi najhitrejši ponudnik Transip, vendar pa so rezultati teh testiranj vseeno zelo zanimivi, saj je kljub večjemu številu skokov do cilja ponudnik Amazon hitrejši od ponudnika DigitalOcean. Pri instanci na DigitalOcean očitno nekje na poti do cilja v eni od strežniških enot prihaja do latence.

2.6 Testiranje sistema z večnamenskimi testi

2.6.1 Opis večnamenskega testnega programa Phoronix Test Suite

Phoronix Test Suite je večnamenski program za univerzalno testiranje različnih aspektov sistemov. Podprt je na večini popularnih operacijskih sistemov, na Linux sistemih pa je množično uporabljen, ker v času pisanja nima podobnih konkurentov in ker je odprtokoden program. Uporabniku je ponujenih ogromno testov, ki jih je program sposoben poganjati. Namestijo se avtomatsko z uporabo "apt-get install" ukaza.
Pri univerzalnem testiranju sistemov oblačnih ponudnikov smo se odločili za uporabo kompleta testov pod ukazom "Run Complex System Test", ki ga lahko poženemo z ukazom "Phoronix Test Suite Interactive 3". Vsebuje teste opisane v nadaljevanju.

Postmark

Postmark je testno orodje, ki simulira operacije z majhnimi datotekami, ki so podobne opravilom, ki jih imajo spletni in poštni strežniki. Privzet testni profil bo opravil 25.000 transakcij (TPS) s 500 datotekami, katerih velikost varira med 5 in 512 KB. V prvem koraku se ustvari zbirko datotek, v naslednjem se z njimi izvede enega od štirih tipov transakcij (izbor, brisanje, branje in dodajanje-append), v zadnjem koraku pa se jih izbriše. Celoten postopek se zgodi večkrat v hitrem zaporedju. Pri izvedbi testa se preveri, koliko takšnih transakcij je sistem sposoben izvesti v določenem času.

RAMspeed

RAMspeed je testno orodje, ki testira zmogljivost predpomnilnikov na procesorju in zmogljivost RAM-a. V Phoronix Test Suite sta v okvirju "Complex System Testa" pognana dva testa. Prvi simulira pomnilniške operacije nad celimi števili, drugi pa pomnilniške operacije nad realnimi števili.

C-Ray

C-Ray je testno orodje, ki testira zmogljivost procesorja nad realnimi števili. V privzetem testnem profilu program ustvari 16 niti vsakemu jedru in virtualno izstreli 8 žarkov za posamezen piksel (simulacija "anti-aliasinga"). Privzeta velikost generirane slike je 1600x1200 pikslov

Apache Benchmark

Apache Benchmark je testno orodje priloženo k Apache sistemu, s katerim se testira zmogljivost obdelovanja HTTP zahtev strežniškega sistema. Specifično testira koliko zahtev je strežniški sistem sposoben sprocesirati v eni sekundi.

2.6.2 Rezultati testiranja s Phoronix Test Suite

Opomba: vsi rezultati predstavljajo povprečje večih poganjanj istih testov, ki so izvedeni avtomatsko.

Referenca	5229 TPS
Amazon	548 TPS
Digital Ocean	2021 TPS
Transip	1403 TPS

Tabela 2.2: Rezultati testa Postmark 1.1.0. TPS - Transactions per second.

Zmagovalec testa Postmark, kot je vidno v tabeli 2.2 je ponudnik DigitalOcean, ki je od Amazona hitrejši kar štirikrat. Ta rezultat predstavlja zanimivo anomalijo v seriji testov, ki drugače niha v prid Amazonu. Predpostavljamo, da je razlog v tem, da je na enem strežniškem sistemu pri ponudniku DigitalOcean manj sočasnih instanc virtualnih strojev. Skoraj dvakrat hitrejši je tudi od ponudnika Transip.

Referenca	11711.05 MB/s
Amazon	11188.84 MB/s
Digital Ocean	9306.97 MB/s
Transip	9432.84 MB/s

Tabela 2.3: Rezultati testa RAMspeed SMP 3.5.0 - Integer.

Referenca	11534.45 MB/s
Amazon	8390.18 MB/s
Digital Ocean	7817.16 MB/s
Transip	7559 MB/s

Tabela 2.4: Rezultati testa RAMspeed SMP 3.5.0 - Floating point.

Kot je vidno v rezultatih v tabeli 2.3 in v tabeli 2.4 je zmagovalec testov RAMspeed Amazon, kar zanimivo, saj je bil pri pomnilniških testih v prejšnjem poglavju najboljši ponudnik Transip. Vzrok za zmago Amazona lahko iščemo v nekonsistentnosti pomnilniške zmogljivosti pri ponudniku Transip.

Referenca	23.82 s
Amazon	133.21 s
Digital Ocean	212.35 s
Transip	130.76 s

Tabela 2.5: Rezultati testa C-Ray 1.1.

Pri testiranju s programom C-Ray je kot je vidno v tabeli 2.5 zmagovalec Transip. Predpostavljamo, da je arhitektura procesorjev, ki jih uporabljajo na Transipovih strežniških sistemih novejša od tiste, ki jo uporabljajo sistemi pri DigitalOcean in iz iste generacije kot tista, ki jo uporabljajo pri Amazonu.

Referenca	22061.02 req/s
Amazon	7101.66 req/s
Digital Ocean	5856.63 req/s
Transip	5903.95 req/s

Tabela 2.6: Rezultati testa Apache Benchmark 2.4.7.

Pri rezultatih Apache Bencmarka, vidnih v tabeli 2.6, je zmagovalec v kombinaciji omrežne in procesorske zmogljivosti Amazon. Rezultati so skladni s pričakovanji, saj je Amazon po velikosti daleč največji in predstavljajo skoraj 20% razliko med konkurentoma.

2.7 Zaključki inicialnega testiranja

Po incialnih testiranjih smo ugotovili očitne razlike med ponudnikoma Amazon, DigitalOcean in Transip. Več kot očitno je, da so komponente pri ponudniku Amazon bistveno novejše kot tiste pri DigitalOcean, saj je pri vseh testih (z izjemo PostMarka) dosegel boljše rezultate. Te komponente vsebujejo vsaj hitrejši RAM, hitrejše trde diske (naša hipoteza je, da so v RAID povezavo zvezani SSD diski) in večjo kapaciteto pri obdelovanju paketov, pri segmentu omrežnih komponent. Zaradi pridobljenih rezultatov na tem mestu hkrati sklepamo, da instanco strežniškega sistema pri Amazonu uporablja več uporabnikov hkrati. Ponudnik Transip je nekako mešanica obeh ponudnikov, saj se v aspektih omrežne zmogljivosti in zmogljivosti trdih diskov primerja z DigitalOcean, v aspetku procesorske in pomnilniške zmogljivosti pa Amazonu. Kot je vidno v poglavju 2.9 je v povprečju Transip tudi najzmogljivejši ponudnik.

2.8 Cenovna analiza in prednosti ponudnikov

Faktor testiranja in končne izbire strežniških sistemov pri ponudnikih poleg zmogljivosti predstavlja tudi mesečna cena. Pri naših izbranih sistemih smo se držali načela, da sistem vsebuje 1 procesor, 1GB RAM pomnilnika in okoli 30GB SSD kapacitetnega prostora. Pri ponudniku DigitalOcean stane izbrana konfiguracija 10$ na mesec oz. 0.0015$ na uro delovanja, na Amazonu je prvo leto delovanja brezplačno, sicer pa stane 0,013$ na uro delovanja oz. okoli 9,672$ na mesec. V tem aspektu je ponudba pri Amazonu za 3% cenejša od približno ekvivalentne ponudbe pri DigitalOcean. Evropski ponudnik Transip za približno enako konfiguracijo računa 10 evrov na mesec, kar predstavlja najdražjo konfiguracijo izmed vseh, vendar pa hkrati z 50GB ponuja tudi največ kapacitetnega prostora. Razlika tako v ceni kot tudi v pridobljeni zmogljivosti je približno enaka med ponudnikoma DigitalOcean in Transip, več kot očitno pa postaja, da je na tem cenovnem nivoju zmagovalec Transip, saj je najbolj konsistenten pri rezultatih vseh testiranj. Kljub temu ni edini faktor pri odločitvi zmogljivost, zato sledijo še nekatere prednosti posameznih ponudnikov:

Prednosti DigitalOcean:

- preprostost namestitve (laičen uporabnik bo bistveno lažje namestil programsko opremo pri DigitalOcean ponudniku),
- ugodna cenovna skalabilnost pri ponudbi,
- sedež v Evropski Uniji (pravno bistveno boljše gledano s strani podjetja).

Prednosti Amazon:

- integracija s številnimi ostalimi storitvami, ki jih ponuja Amazon,
- najboljša podpora strankam,
- v večini testov najhitrejši izmed testiranih ponudnikov.

Prednosti Transip:

- poleg gostovanja ponuja tudi nakup domene in povezavo z izbrano instanco,
- visok SLA (99.99%),
- ponudnik z največ prostora glede na ceno.

2.9 Končni rezultati in ovrednotenje

Test	Amazon	Digital Ocean	Transip
Sysbench CPU	0,93	0,79	**1**
Sysbench HDD	0,45	**1**	0,42
Sysbench 1KB RAM	0,70	0,59	**1**
Sysbench 1MB RAM	0,95	0,68	**1**
Postmark	0,27	**1**	0,69
Ramspeed Integer	**1**	0,83	0,84
Ramspeed Float	**1**	0,93	0,90
CRay	0,98	0,69	**1**
Apache Benchmark	**1**	0,82	0,83
Skupaj	7,28	7,33	**7,68**

Tabela 2.7: Ovrednotenje rezultatov po točkovnem sistemu, opisanem v poglavju 2.4.1.

Po izvedbi vseh testov in njihovem ovrednotenju, ki so vidni v tabeli 2.7, je očiten zmagovalec ponudnik Transip, ki je performančno najhitrejši v štirih izmed devetih testov. Kljub vsemu po procentualnemu izračunu opazimo, da je povprečna performančna razlika med prvo uvrščenim Transipom in trejte uvrščenim Amazonom manjša od 10%. Če v ozir temu dejstvu vzamemo še cenovne razlike, obrazložene v prejšnjem poglavju, lahko trdimo, da v aspektu razmerja cena/zmogljivost med ponudniki efektivno ni razlike. Uporabnikom zato svetujemo, da ponudnika izbere bodisi glede na specifične performančne potrebe njegove rešitve, bodisi glede na prednosti posameznih ponudnikov, opisanih v prejšnjem poglavju.

2.10 Zaključek

Po opravljenih testiranjih lahko zanesljivo trdimo, da performančne razlike med ponudniki strežniških storitev nedvomno obstajajo, vendar pa so gledano v celoti minimalne. Ker lahko to po naših pridobljenih rezultatih trdimo samo za cenovno ugodne strežniške sisteme opisane v poglavju 2.2, bi bilo teste smiselno ponoviti na večih različnih cenovnih nivojih s hitrejšimi in zmogljivejšemi komponentami, kjer bi bile lahko performančne razlike v prid večjim ponudnikom. Uporabljene informacije so podane v virih [11], [12], [13], [14], [15], [16], [17] in [18].

Poglavje 3

Analiza zmogljivosti medijskih strežnikov in prenosa datotek

Matej Kopar, Tilen Nedanovski, Julija Petrič

3.1 Opis problema

Tema pričujočega poglavja je analiza zmogljivosti operacijskega sistema Raspbian na nizkocenovnem računalniku Raspberry Pi 2 in sistema Bananian na nizkocenovnem računalniku Banana Pi.

Raspbian in Bananian sta odprtokodna operacijska sistema, ki temeljita na Debianu za strojno opremo nizkocenovnega računalnika Raspberry Pi 2 oz. Banana Pi. Verzija operacijskega sistema Raspbian je Jessie, Bananian pa 1.0. Raspbian je prosto dostopen operacijski sistem, ki temelji na Debian OS, narejen specifično za Raspberry Pi. Tudi Bananian temelji na Debian OS, optimiziran pa je za strojno opremo nizkocenovnega računalnika Banana Pi. Preveriti želimo, če lahko z nizkocenovno opremo dosežemo zadovoljive rezultate deljenja datotek ter predvajanja večpredstavnostnih vsebin za domačo rabo.

Za testiranje predvajanja večpredstavnostnih vsebin smo izbrali video v visoki resoluciji ter merili zakasnitev ob večih uporabnikih.

Za testiranje prenosa datotek smo merili hitrost prenosa datoteke z odjemalca na strežnik in obratno. Breme je množica zahtev in več zaporednih in vzporednih prenosov. Nekaj testov smo naredili preko zunanjega omrežja, vendar smo opazili, da ravno pasovna širina predstavlja največje ozko grlo. Zato

smo kasneje testirali samo na lokalnem omrežju. S tem smo se znebili dejavnikov, ki lahko povzročajo ozko grlo - pasovna širina in slab usmerjevalnik.

3.2 Rešitev problema

Vsa testiranja smo izvedli na sistemih Raspberry Pi 2 in Banana Pi z diskovnim poljem RAID 0 (tabela 3.1). Testiranje strežnikov in predvajanje video vsebin smo izvedli s predvajalnikom VLC, ki omogoča zagon preko ukazne vrstice z vnaprej podanim URL naslovom. Merili smo zakasnitev oz. čas ki ga potrebujemo za odgovor na našo poizvedbo ter kvaliteto prenosa oz. število izgubljenih okvirjev.

Za vsak protokol smo izvedli dva testa v ukazni lupini s pomočjo orodij CURL, TIME in AB (Apache Benchmark):

1. **Prenos datotek:** Merimo hitrost prenosa datotek z odjemalca na strežnik in s strežnika nazaj na odjemalec. Breme je množica zahtev in več zaporednih prenosov. Vsebina datotek je naključna.

2. **Tok podatkov:** Med odjemalcem in strežnikom ustvarimo tok podatkov in merimo zakasnitev glede na število zahtev in kvaliteto prenosa.

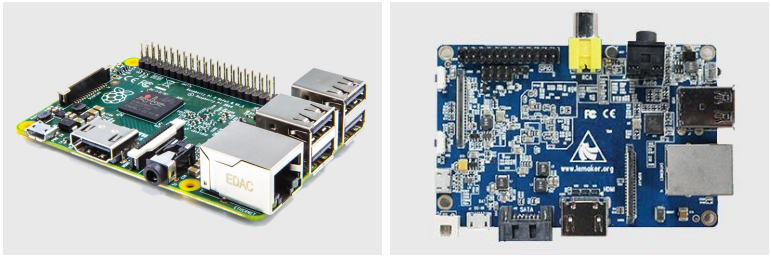
3.2.1 Raspberry vs Banana Pi

Banana Pi je zasnovana dovolj podobno kot Raspberry Pi, tako da lahko večina programske opreme ustvarjene za Raspberry Pi teče tudi na Banana Pi. Primerjava Raspberry Pi 2 Model B in Banana Pi je predstavljena v tabeli 3.1.

	Raspberry Pi 2 Model B	Banana Pi
Processor	ARM Cortex A7 - 900 MHz - Quad core	Allwinner A20 - Cortex A7 - 1 GHz - Dual core
RAM	1 GB LPDDR2	1 GB DDR3
GPU	VideoCore IV - Dual core	Mali 400 MP2 - Dual core
Video out	1 x HDMI - 1.2 ali 1.4	1 x HDMI, 1 x Composite
Network	1 x 10/100 Ethernet	1 x 10/100/1000 Ethertnet

Tabela 3.1: Primerjava sistemov Raspberry Pi 2 Model B in Banana Pi [19]

.



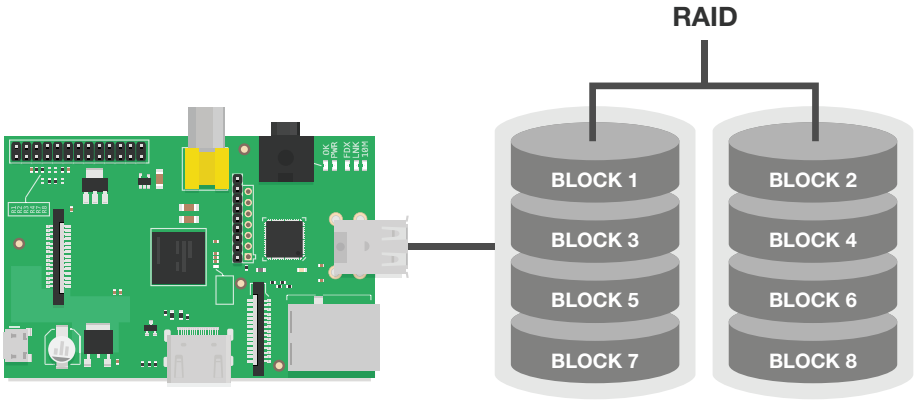
Slika 3.1: Raspberry Pi 2 Model B vs Banana Pi [19]

3.2.2 RAID 0

Testiranje sistemov smo izvedli na zgoraj omenjenih sistemih z diskovnim poljem RAID 0 [22] (*Redundant array of independent disks; originally redundant array of inexpensive disks*).

RAID 0 je namenjen povezovanju in upravljanju dveh ali več trdih diskov. S povezovanjem manjših trdih diskov dosežemo večjo in hitrejšo, lahko pa tudi zanesljivejšo logično enoto.

Po pregledu že znanih testiranj sistema Raspberry Pi z diskovnim poljem RAID 0 smo se odločili, da bomo testiranje izvedli na nekaj USB ključih (16 oz. 32GB), saj bo tako zahtevnost izdelave polja precej nižja. Na Banana Pi smo izvedli tesitranje z USB ključi kot tudi na disku, ker disk s SATA priklopom omogoča hitrejše prenose. Slika 3.2 prikazuje arhitekturo testiranega sistema.



Slika 3.2: Primer kompozicije z dvema pomnilniškima napravama v polju RAID 0.

3.3 Implementacija sistema

3.3.1 RAID 0

Postopek vzpostavitve RAID 0:

1. Diske prek USB vmesnika povežemo z Raspberry Pi.
2. Ukaz `sudo fdisk -l` izpiše vse pomnilniške naprave, ki so na voljo na sistemu.
3. Namestimo `Mdadm` z `sudo apt-get install mdadm`.
4. Z ukazom `mdadm -Cv /dev/md0 -l0 -n2 /dev/sd[ab]1` ustvarimo polje RAID 0 na lokaciji `/dev/md0` z `raid0` (zastavica `-l0`). V polje sta vključena dva diska: `/dev/sda1` in `/dev/sdb1`.
5. Datotečni sistem na polju ustvarimo z `mkfs /dev/md0 -t ext4`.

3.4 Nadzor delovanja

3.4.1 SYSBENCH

SYSBENCH [21] je skupno ime za zbirko orodij, ki na podlagi obremenitvenih testov merijo zmogljivost različnih aspektov sistema — procesno moč, hitrost vhodno-izhodnih enot, zmogljivost podatkovne baze ipd. Da pridobimo splošni občutek zmogljivosti sistemov, najprej naredimo splošne teste, ki zajemajo meritve CPU, pomnilnika ter pisanja in branja iz USB naprav oz. SATA priklopa. Orodje SYSBENCH vsakega izmed testov požene večkrat (nekaj tisočkrat), nato pa vrne povprečni rezultat. Tako so vsi rezultati naslednjih meritev z orodjem SYSBENCH povprečni rezultati.

CPU test

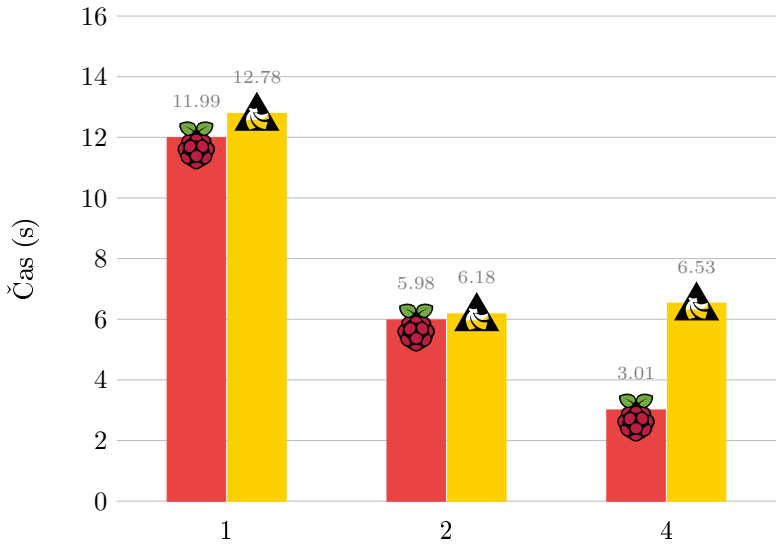
CPU test meri čas računanja prvih N praštevil s t nitmi, pri čemer sta N in t vhodna argumenta. Z ukazom

```
sysbench --test=cpu --cpu-max-prime=N --num-threads=t run
```

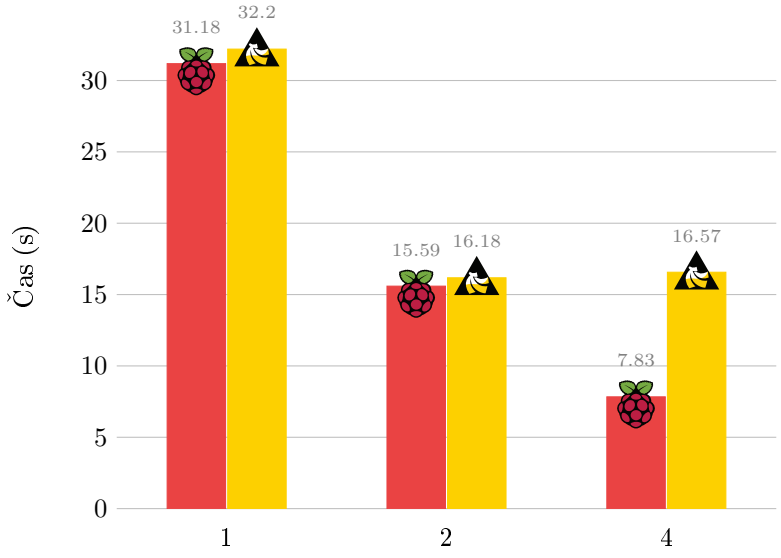
poženemo program za izračun praštevil pri podani omejitvi `--cpu-max-prime` in številu niti `--num-threads`. Teste smo izvedli za 1000, 2000 in 5000 praštevil in eno, dve ter štiri niti.

Predvidevamo, da bodo rezultati približno enaki, saj imata obe napravi približno enako procesorsko moč. Razlika bi morala biti očitna le pri uporabi štirih niti, saj ima sistem Banana Pi procesor z dvema jedroma, Raspberry Pi pa procesor s štirimi jedri.

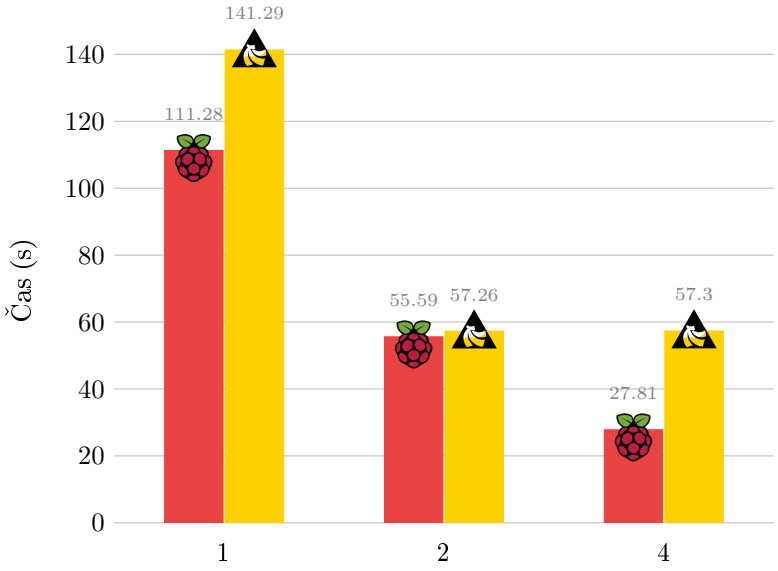
Rezultati meritev so predstavljeni na slikah 3.3, 3.4 in 3.5.



Slika 3.3: Obremenitev CPU enote z računanjem 1000 praštevil za oba sistema. Nižji rezultat je boljši.



Slika 3.4: Obremenitev CPU enote z računanjem 2000 praštevil za oba sistema. Nižji rezultat je boljši.



Slika 3.5: Obremenitev CPU enote z računanjem 5000 praštevil za oba sistema. Nižji rezultat je boljši.

Raspberry Pi za računanje porabi manj časa ne glede na vhodne argumente. Čas se zmanjšuje z vsako dodatno nitjo. Čas ki ga potrebuje Banana Pi za računanje praštevil se zmanjša, če za računanje uporabimo dve niti. Če je niti več, je čas enak ali daljši. Rezultati so smiselni. Računanje na Banana Pi namreč poteka na dvojedernem procesorju, na Raspberry Pi pa na štirijedernem procesorju.

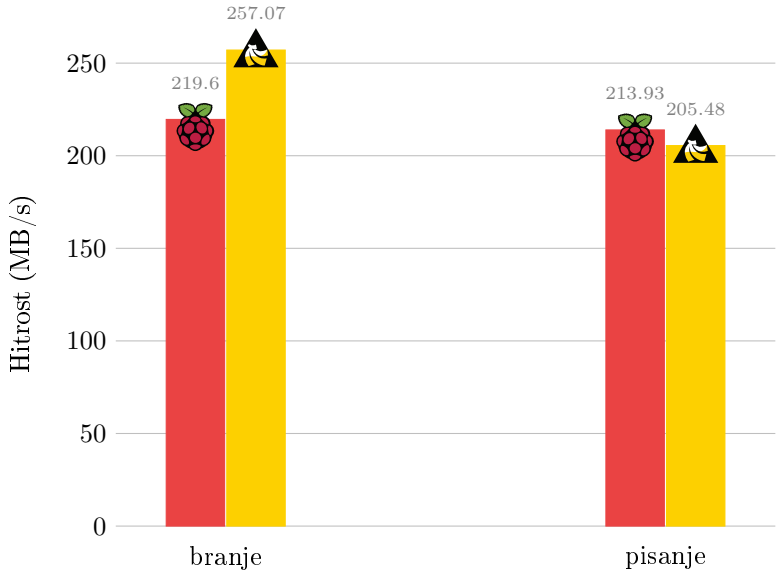
Memory test

Program za testiranje zmogljivosti spomina najprej alocira ustrezno količino spomina v pomnilniku, ki služi kot medpomnilnik za branje in pisanje. Nato v zanki piše ali bere podatke toliko časa, dokler ne zapolni dodeljenega dela pomnilnika t.j. medpomnilnika. Zahteve za branje ali pisanje so lahko zaporedne ali v naključnem vrstnem redu. Z ukazom

```
sysbench --test=memory --memory-total-size=N
--memory-oper=(read|write) --memory-access-mode=(seq|rnd) run
```

poženemo program za testiranje spomina pri velikosti medpomnilnika `--memory-total-size` in operaciji `--memory-oper`.

Teste smo izvedli z branjem in pisanjem 2GB podatkov z operacijami v naključnem vrstnem redu. Predvidevamo, da bodo rezultati na napravi Banana Pi boljši, ker uporablja zmogljivejši pomnilnik DDR3, medtem ko je na Raspberry



Slika 3.6: Rezultati testa pomnilnika za oba sistema. Višji rezultat je boljši.

Pi napravi v uporabi pomnilnik DDR2. Kapaciteta pomnilnika je pri obeh napravah enaka - 1GB. Rezultati meritev so prikazani na sliki 3.6:

Rezultati nas nekoliko presenetijo, saj sta napravi skoraj enakovredni (pri branju je naprava Banana Pi nekoliko boljša). Kljub boljšemu pomnilniku na napravi Banana Pi so rezultati naprave Raspberry Pi zelo primerljivi.

File I/O test

Test s pisanjem in branjem datotek z naključno vsebino meri prepustnost in s tem učinkovitost datotečnega sistema. Najprej z ukazom

```
sysbench --test=fileio --file-total-size=N prepare
```

pripravimo podatke, nato z ukazom

```
sysbench --test=fileio --file-total-size=N --file-test-mode=rndrw run
```

test poženemo. Na koncu počistimo testne datoteke z ukazom

```
sysbench --test=fileio --file-total-size=N cleanup.
```

Predvidevamo, da bo branje in pisanje pri uporabi USB diskov primerljivo, medtem ko bi moralo biti branje in pisanje na SATA disk bistveno hitrejše od pr-

vega. Ker se lahko zgodi, da so zmogljivosti različnih USB diskov neprimerljive, smo pri testih uporabili iste USB diske na obeh sistemih.
Rezultati testov so navedeni v naslednjih alinejah, njihov grafični prikaz pa je predstavljen na sliki 3.7:

- Banana Pi s podatki na zunanjem trdem disku SATA:

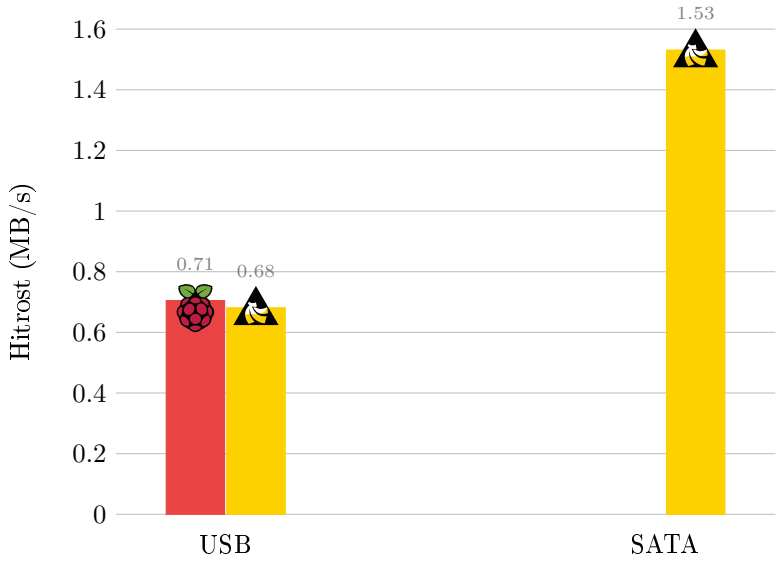
```
Read 275.62Mb Written 183.75Mb Total transferred 459.38Mb
     (1.5312Mb/sec)
   98.00 Requests/sec executed

Test execution summary:
    total time: 300.0006s
    total number of events: 29400
    total time taken by event execution: 165.9502
    per-request statistics:
          min: 0.04ms
          avg: 5.64ms
          max: 106.88ms
          approx. 95 percentile: 12.67ms
```

- Banana Pi s podatki na zunanjih trdih diskih USB:

```
Read 125.728Mb Written 80.429Mb Total transferred 206.157Mb
     (680.59Kb/sec)
    42.26 Requests/sec executed

Test execution summary:
    total time: 302.9108s
    total number of events: 12800
    total time taken by event execution: 20.6915
    per-request statistics:
          min: 0.04ms
          avg: 0.78ms
          max: 9.38ms
          approx. 95 percentile: 1.95ms
```

- Raspberry Pi s podatki na zunanjih trdih diskih USB:

```
Read 131.25Mb Written 87.5Mb Total transferred 218.75Mb (705.49Kb/sec)
   44.09 Requests/sec executed

Test execution summary:
    total time: 317.5096s
    total number of events: 14000
    total time taken by event execution: 22.3205
    per-request statistics:
          min: 0.03ms
```



Slika 3.7: Rezultati testa File I/O za oba sistema. Višji rezultat je boljši.

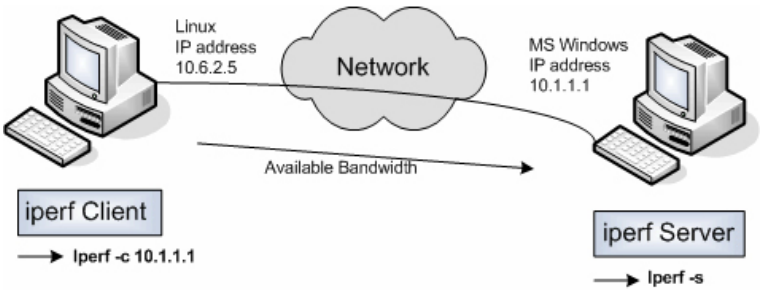
```
avg: 1.59ms
max: 13.31ms
approx. 95 percentile: 2.11ms
```

Kot lahko opazimo iz grafa, so rezultati primerljivi, kot smo predvideli v hipotezi pred izvajanjem testa. Testiranja z različnimi USB diski so se zelo razlikovala, zato smo testiranje izvedli z istimi USB diski ter tako zagotovili boljšo primerljivost. Opazimo tudi, da je zmogljivost SATA diska bistveno boljša od zmogljivosti USB naprav, kar smo predvideli že v hipotezi pred testiranjem. Zmogljivost SATA diska v primerjavi z USB diski je smiselna in pričakovana, ker omogoča SATA bistveno višje hitrosti od USB-ja.

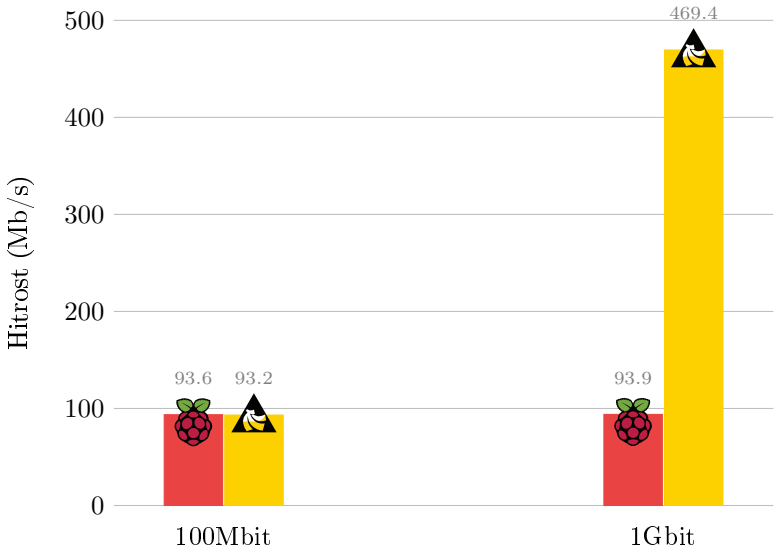
3.4.2 IPERF

IPERF [20] je orodje za merjenje pasovne širine in kvalitete mrežnega vmesnika. Test se izvaja na povezavi med dvema sistemoma — odjemalcem in strežnikom. Na obeh sistemih se izvaja program IPERF. To ponazarja slika 3.8. Kvaliteta mrežnega vmesnika se določa na podlagi naslednjih lastnostih:

- latenca oz. odzivni čas,
- jitter oz. variacija latence (meri se z IPERF UDP testi),
- število izgubljenih datagramov (meri se z IPERF UDP testi).



Slika 3.8: Povezava med odjemalcem in strežnikom. [20]



Slika 3.9: Rezultati testa IPERF za oba sistema. Višji rezultat je boljši.

Z ukazom

```
iperf -s [options]
```

poženemo program na strežniku in z ukazom

```
iperf -c host [options]
```

na odjemalcu, pri čemer je `host` mrežni naslov strežnika na katerem izvajamo iperf. Orodje je zasnovano tako, da izvaja testiranje nekaj časa, nato pa vrne povprečen rezultat.

Teste smo izvedli s prenosom 100Mbit podatkov in 1Gbit podatkov prek protokolov UDP in TCP. Pri testiranju 100Mbit pasovne širine pričakujemo približno enako zmogljivost, medtem ko pri testiranju 1Gbit pasovne širine pričakujemo bistveno razliko pri sistemu Banana Pi, ker le-ta vsebuje 10/100/1000 mrežno kartico, medtem ko sistem Raspberry Pi vsebuje 10/100 mrežno kartico. Rezultati so prikazani na sliki 3.9.

Rezultati so pričakovani. Pri prenosu 100Mbit podatkov je hitrost prenosa na Raspbery Pi in Banana Pi skoraj popolnoma enaka.
Banana Pi dosega boljše rezultate pri prenosu 1Gbit podatkov zaradi že prej omenjene boljše (10/100/1000) mrežne kartice. Zaradi boljše kartice sistem Banana Pi omogoča boljšo prepustnost in tako hitrejši prenos, kar vidimo tudi na grafu, kjer je sistema Banana Pi hitrejši za približno 5 krat.

3.4.3 FTPBENCH

Sprva smo prenos datotek želeli testirati z orodji CURL, TIME in AB (Apache Benchmark), vendar smo se nato odločili za orodje FTPBENCH, katerega osnovna funkcionalnost je testiranje hitrosti prenosa datotek s strežnika oz. nalaganje na strežnik. Program FTPBENCH [23] z obremenitvami meri zmogljivost in prepustnost FTP strežnika. Vključuje naslednje teste:

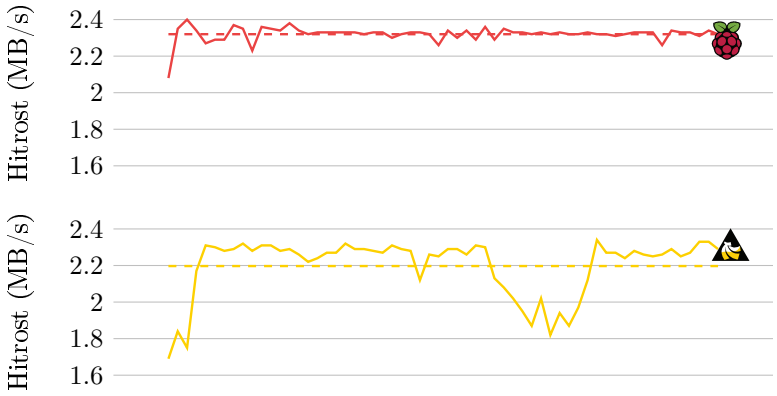
- login test,
- upload test (FTP ukaz STOR),
- download test (FTP ukaz RETR).

Teste smo izvedli od doma na obeh sistemih s priključenimi USB diski. Teste smo izvedli dne 2. 5. 2016. Specifikacije odjemalca so naslednje:

- lokacija: Šentrupert na Dolenjskem,
- ponudnik: Amis, simetrična optična povezava 20Mbit/20Mbit.

Ker je pasovna širina bistveno nižja od vseh prej navedenih pasovnih širin, pričakujemo, da bo pasovna širina odjemalca naše ozko grlo. Zaradi tega bi moral biti prenos in nalaganje največ 20Mbit/s (oz. okrog 2,5MB/s).
Rezultati so prikazani na slikah 3.10 in 3.11:

Slika 3.10: Rezultati testa Ftpbench - upload za oba sistema. Višji rezultat je boljši.



Slika 3.11: Rezultati testa Ftpbench - download za oba sistema. Višji rezultat je boljši.

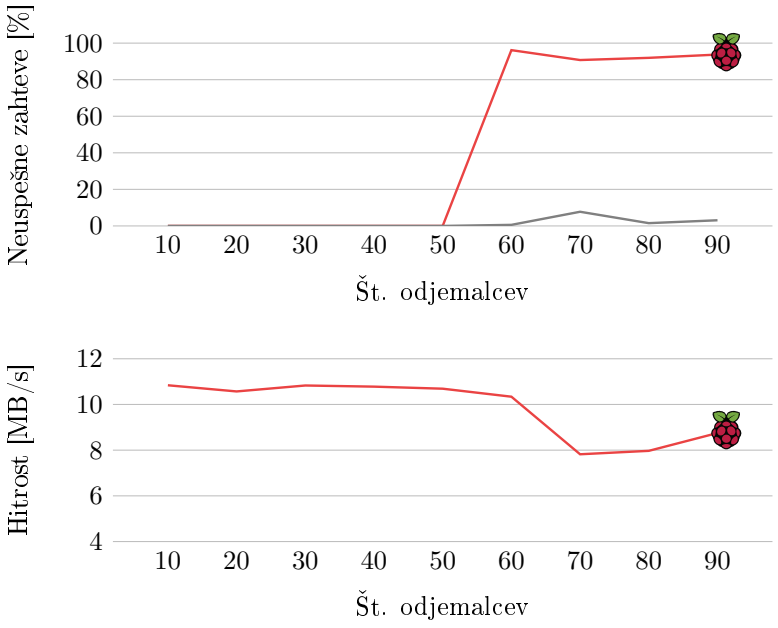
Polna črta na grafih prikazuje download oz. upload v enoti časa. Črtkana črta prikazuje povprečno vrednost.
Rezultati so smiselni. Oba sistema imata povprečno vrednost okrog 2,2MB/s, kar je ravno naše predvidevanje ozkega grla - pasovna širina odjemalca. V tem primeru sta tako oba sistema brez večjih težav zmogla obe testiranji in se odrezala zelo zadovoljivo.

Vzporednost prenosov

Z orodjem FTPBENCH smo testirali tudi zmogljivost FTP strežnika v primeru, ko je zahtev za prenos več in so med sabo vzporedne. Ob tej predpostavki smo skušali ugotoviti koliko lahko sistem obremenimo preden strežnik zahtevo zavrne (rejected) ali pa čas odziva strežnika na zahtevo preseže časovno omejitev (timeout). Da smo se znebili vpliva pasovne širine omrežja, smo test izvajali na lokalnem omrežju. V vsaki iteraciji smo prenesli 1GB podatkov. Časovna omejitev odziva na zahtevo je 10s. Rezultati so prikazani na sliki 3.12 za Raspberry Pi in na sliki 3.13 za Banana Pi. Rdeč graf predstavlja delež zavrnjenih zahtev, siv graf pa delež zahtev s preseženim časom odziva.

Raspberry Pi

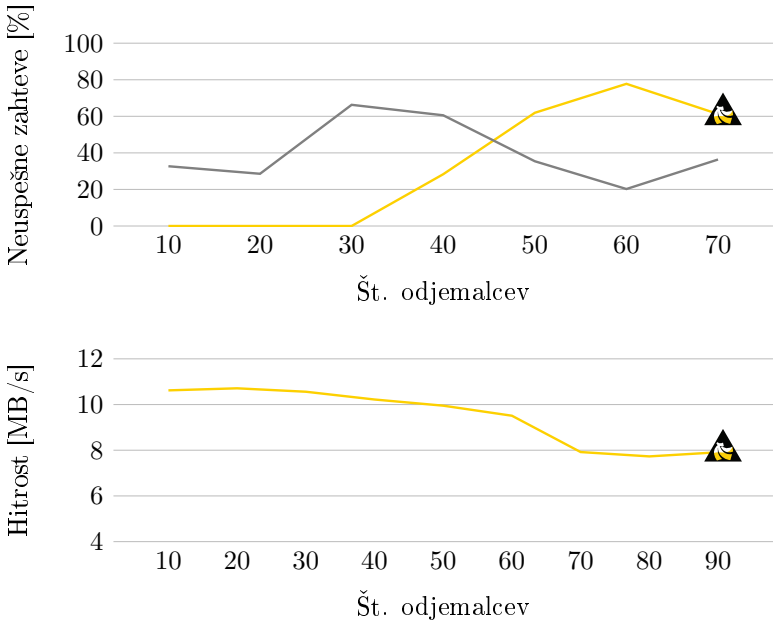
Če sistem obremenimo z do vključno 50 vzporednimi odjemalci, prenos poteka brez težav oz. odpovedi. Če je vzporednih odjemalcev več, se delež neuspešnih zahtev poveča, hitrost pa se zmanjša. Prenos je kljub temu uspešen, a zanj sistem potrebuje več časa.



Slika 3.12: Rezultati testa FTPBENCH z vzporednimi prenosi - sistem Raspberry Pi. Pri zgornjem grafu je nižji rezultat boljši, pri spodnjem pa višji.

Banana Pi

Do prekinitev prihaja že pri 10 vzporednih zahtevah, a je prenos kljub temu uspešen. Pri 40 vzporednih zahtevah prihaja do manjšega števila odpovedi. To število se v vsaki iteraciji testa poveča. Pri 60 vzporednih zahtevah je prenos neuspešen (pri prenosu 1GB podatkov se jih je preneslo le 554MB). 50 vzporednih zahtev je torej zgornja meja pri kateri je prenos še uspešen, kljub odpovedim.



Slika 3.13: Rezultati testa FTPBENCH z vzporednimi prenosi - sistem Banana Pi. Pri zgornjem grafu je nižji rezultat boljši, pri spodnjem pa višji.

3.4.4 VLC

Meritve toka podatkov smo opravili z medijskim predvajalnikom VLC. VLC je program, namenjen predvajanju večpredstavnostnih vsebin, poleg tega pa vključuje tudi druge funkcije kot sta pretvarjanje datotek in kompresija zvoka in videa. S pomočjo VLC smo merili zmogljivost predvajanja medijskih vsebin prek protokola RTSP (Real Time Streaming Protocol).

Na obeh sistemih smo vzpostavili pretočni strežnik in nato na odjemalcih predvajali vsebine, ki nam jih je strežnik posredoval. Merili smo zakasnitev in kvaliteto prenosa. Meritev zakasnitev smo opravili z orodjem PING, ki poleg testiranja povezave meri tudi latenco - koliko časa paket potuje od točke X do točke Y. Kvaliteto prenosa smo merili s številom zavrženih paketov. Število zavrženih paketov smo dobili iz statistike predvajalnika VLC na odjemalcu. Meritve smo opravili na lokalnem omrežju, saj se tako znebimo vpliva pasovne širine omrežja oz. vpliva usmerjevalnika. Testiranja smo izvajali na dveh odjemalcih:

- Lenovo X1 Carbon, 8GB RAM, 2.00GHz i7, Windows 8.1
- Apple Macbook Pro, 8GB RAM, 2.30GHz i7, OSX El Capitan

Strežnik smo vzpostavili z ukazom

`cvlc -vvv file://filepath -sout '#rtp{sdp=rtsp://:8080/test.sdp}'`.

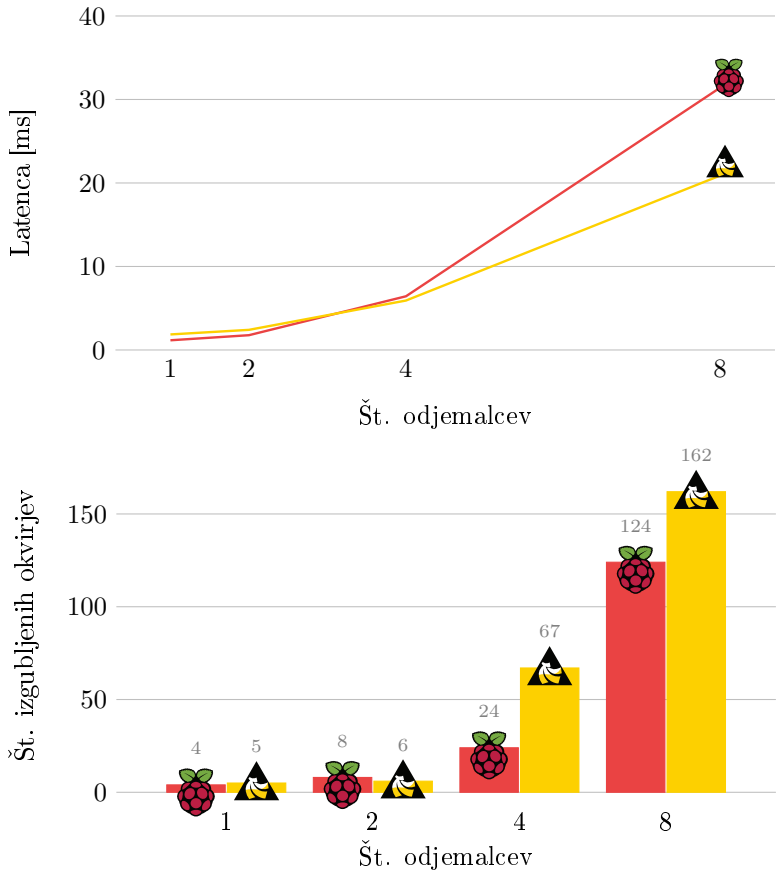
Nato smo vsebine predvajali z ukazom

`cvlc rtsp://naslov:8080/test.sdp`,

pri čemer `naslov` predstavlja IP naslov strežnika. Zavoljo testiranja smo si izbrali 3 minutni video v resoluciji 1920 x 1080, ki smo ga dobili na naslovu https://vimeo.com/16917950. Velikost videa je 223.56MB. Teste smo izvedli na obeh sistemih s številom odjemalcev 1, 2, 4 in 8. Za testiranje zgolj ene resolucije smo se odločili zato, ker ima danes večina monitorjev, prenosnih računalnikov in televizorjev že vsaj tako (1920 x 1080) resolucijo, zato se nam zdi, da testiranje slabših resolucij ne doprinese k ugotovivam.

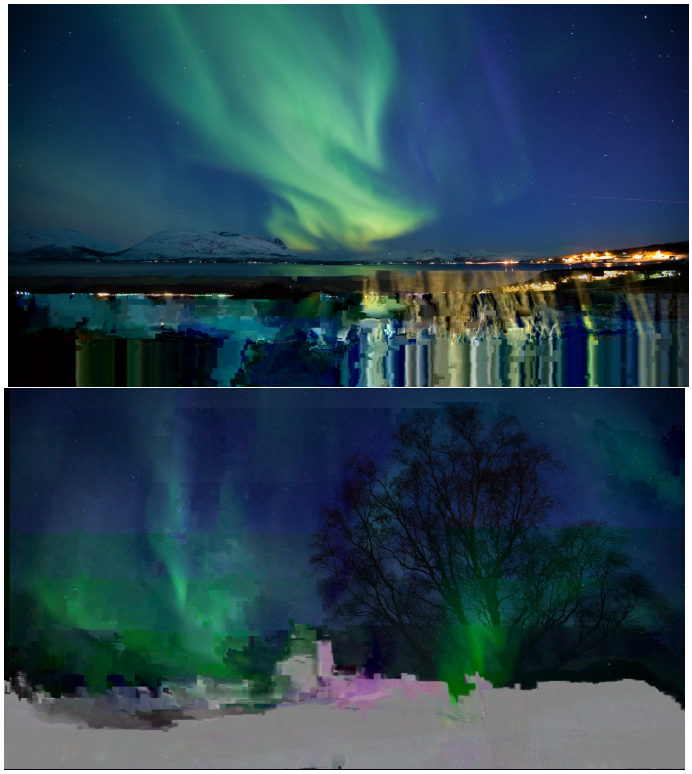
Glede na dosedajšnje rezultate menimo, da bo ogled s 4 vzporednimi odjemalci še zadovoljiv. Predvidimo tudi, da se bo ploščica Banana Pi odrezala bolje od ploščice Raspberry Pi zaradi boljše mrežne kartice. Vsi rezultati predstavljajo povprečje celotnega odjemanja ter nekaj sekund pred in nekaj sekund po odjemanju (pri testu zakasnitve). Rezultati meritev so prikazani na sliki 3.14.

Hipotezo za zadovoljiv ogled s 4 vzporednimi odjemalci smo zavrgli. Glede na videno lahko rečemo, da je za izbran video odlična kvaliteta prenosa pri izgubi do 10 okvirjev (najmanj oz. nič artefaktov) oz. pri 0,1% izgubi. Tako kvaliteto imamo pri odjemanju z 1 oz. 2 vzporednima odjemalcema. Pri izgubi do 30 okvirjev je video pogojno gledljiv, z nekaj več artefakti, ki bi bili pri predvajanju filma moteči, pri predvajanju npr. športnega dogodka verjetno



Slika 3.14: Rezultati zakasnitev (zgoraj) ter števila zavrženih okvirjev (spodaj) pri testih s programom VLC - sistema Raspberry Pi in Banana Pi. Nižji rezultati so boljši.

manj. Tako kvaliteto imamo pri 4 vzporednih odjemalcih za Raspberry Pi. Pri izgubi nad 30 okvirjev je video že zelo slabo gledljiv, saj je prisotnih ogromno artefaktov in posledično izrazito slabša kvaliteta slike. Tako kvaliteto imamo pri 4 vzporednih odjemalcih za Banana Pi ter pri 8 vzporednih odjemalcih za oba sistema. V primeru 8 vzporednih odjemanj dobimo zelo slabo kvaliteto prenosa z zelo povečano latenco na obeh sistemih. Zaradi rezultatov za 8 vzporednih prenosov smo se odločili, da za več vzporednih odjemalcev ne bomo testirali, ker so rezultati izrazito presegli mejo kvalitete.



Slika 3.15: Artefakti sistema Raspberry Pi 2 Model B in sistema Banana Pi.

3.5 Zaključek

V seminarski nalogi smo raziskovali, ali so nizkocenovni računalniki primerni za uporabo v strežniške namene v domačem okolju. Po osnovnih testih se naši nizkocenovni računalniki niso odrezali ravno najbolje, vendar smo s testiranjem prenosa in nalaganja datotek preko omrežja ugotovili, da ozko grlo prej dosežemo s pasovno širino povezave kot pa s pasovno širino vmesnika. Po testiranju vzporednih prenosov smo ugotovili, da je sistem ranljiv na veliko število odjemalcev. Pri testiranju predvajanja vsebin smo ugotovili, da sta oba sistema primerna za predvajanje do 2 hkratnih odjemalcev, za kaj več pa sta strežnika prešibka, saj pride (ob večji izgubi okvirjev) do slabše kvalitete prenosa in bolj motečih artefaktov.

S tem pridemo do zaključka, da sta sistema primerna za nezahtevne domače uporabnike če gre za prenos datotek in predvajanje medijskih vsebin oz. za nekoliko zahtevnejše uporabnike, če gre samo za prenos datotek. Pri polni uporabi sistemov (torej tako za prenos datotek kot tudi za predvajanje medijskih vsebin) smo s testi ugotovili, da je kvaliteta prenosa odlična z največ dvema hkratnima odjemalcema in da je predvajanje vsebine pogojno sprejemljivo pri

štirih odjemalcih. Pri uporabi sistema samo za prenos datotek pa smo ugotovili da je možna uporaba z več hkratnimi uporabniki.

Poglavje 4

Analiza zmogljivosti oblačne storitve Cloud9 IDE

Ivan Antešić

4.1 Opis problema

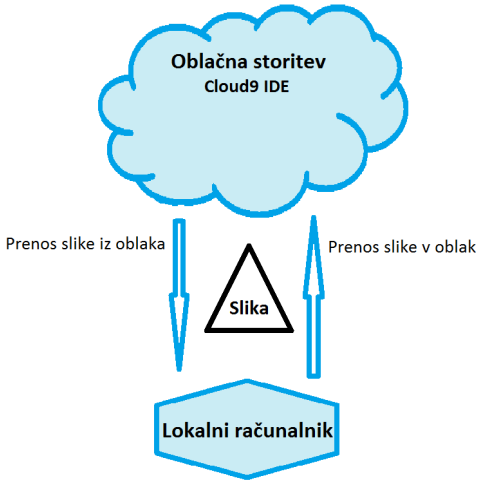
Želel sem preveriti zmogljivost oblačne storitve Cloud9 IDE. Pri testiranju sem se osredotočil na prenašanje datotek med strežnikom na oblaku in lokalnim računalnikom.

Za potrebe testiranja zmogljivosti oblaka sem si zamislil naslednji scenarij. Uporabnik želi imeti shranjene različne datoteke na oblaku, npr. slike, programe itd. Za ta namen lahko uporabi različne že obstoječe storitve (Dropbox, Google Drive, ...). Hitro se zgodi, da porabi prostor, ki mu je namenjen v brezplačni različici. Če ne želi plačati in ima potrebno osnovno znanje spletnega programiranja, si lahko sam postavi takšno aplikacijo na enem od ponudnikov brezplačnih oblačnih storitev.

Zato sem naredil takšno testno aplikacijo in z njo analiziral zmogljivost Cloud9. Aplikacija sprejema poljubne datoteke in jih shranjuje na oblak. Po želji lahko datoteko izbrišemo ali pa jo prenesemo nazaj na računalnik. Na sliki 4.1 je prikazan diagram sistema.

4.2 Oblačna storitev

Cloud9 [24] je spletno integrirano razvijalno okolje, ki gostuje na Google Compute Engine - Googlovi oblačni različici IaaS (*infrastructure as a service* - in-



Slika 4.1: Diagram sistema.

frastruktura kot storitev). Cloud9 je odprtokodni projekt in podpira večino programskih jezikov z prednastavljenimi delovnimi okolji. Omogoča poganjanje aplikacije na oblaku brez posebnih priprav in drugih programov, zato sem za analizo zmogljivosti izbral to okolje. V zastonjski različici nudi 1 CPU, 1GB RAMa in 5GB HDD, kar zadostuje potrebam moje aplikacije. Z **tracert** ukazom v Microsoftovi ukazni lupini sem izsledil lokacijo strežnika na kateri teče aplikacija. Nahaja se v San Franciscu, v ZDA.

4.3 Tehnologije

4.3.1 HTML

HTML [25] ali *Hyper Text Markup Language* je standardni označevalni jezik, ki se uporablja pri izdelavi spletnih strani. HTML semantično opisuje strukturo strani in omogoča vključevanje drugih datotek, potrebnih za druge vidike spletnih strani.

4.3.2 CSS

CSS [26] ali *Cascading Style Sheets* je standardni jezik za opisovanje izgleda spletnih strani. V HTML vključena CSS datoteka določa barve in obliko elementov spletne strani. Z ločevanjem strukture in izgleda strani omogočimo večjo preglednost kode.

4.3.3 JavaScript

Javascript [27] je dinamičen, objektni programski jezik. Sintaksa jezika ohlapno sledi programskemu jeziku C. Veliko se uporablja za izdelavo spletnih strani skupaj z HTML in CSS, kjer skrbi za interaktivnost in dinamiko strani.

4.3.4 Node.js

Node.js [28] je odprtokodno okolje (jezik) namenjeno razvijanju strežniških spletnih aplikacij. Zgrajen je na Googlovem V8 Javascript engine-u. Uporablja neblokirajoč vzhodno/izhodni model (ki je hiter in raztegljiv) in uporablja sistem odprtokodnih knjižnic npm.

4.3.5 Firefox Network Monitor

Firefox Network Monitor [29] je brezplačno, odprtokodno orodje vgrajeno v novejše različice brskalnika Firefox namenjeno nadzorovanju delovanja spletne strani in njenih komponent (CSS, HTML, Javascript). Z orodjem lahko vidimo katere zahteve se pošiljajo na stran, kaj vsebujejo in koliko časa traja obdelovanje teh zahtev. Čas obdelave zahtev se nato deli na več podčasov, ki jih lahko spremljamo.

4.4 Implementacija aplikacije

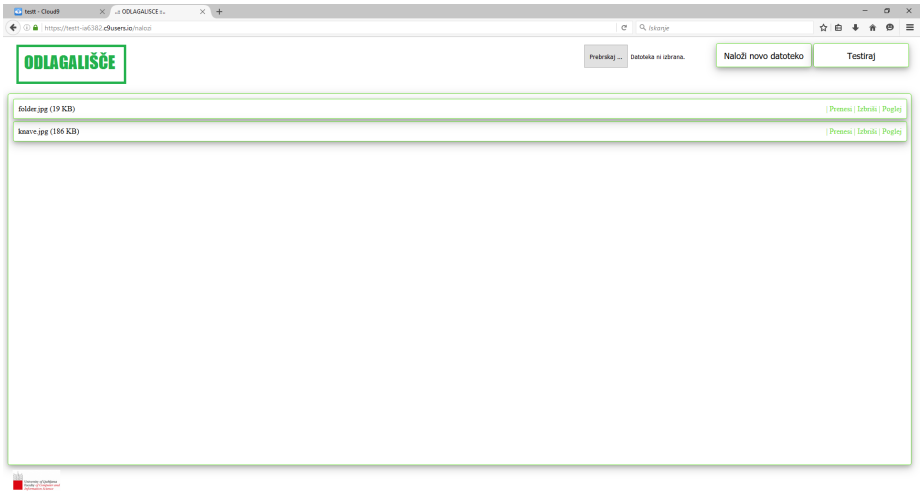
Aplikacija je sestavljena sestavljena iz 2 glavnih delov in sicer strežnika in odjemalca.

4.4.1 Strežnik

Strežnik je napisan v Node.js. Kreira http strežnik in omogoča posredovanje različnih statičnih vsebin ter storitev odjemalcu. Z odjemalcem komunicira preko XMLHttp zahtev in odgovorov. Uporabnik tako lahko naloži različne datoteke na strežnik, jih zbriše, pogleda v novem zavihku, ali prenese na svoj lokalni računalnik. Strežnik shranjuje datoteke v prostor na oblaku, ki mu ga dodeli Cloud9.

4.4.2 Odjemalec

Odjemalec je sestavljen iz treh glavnih datotek: iz HTML, CSS in JavaScript datoteke. HTML opiše spletno stran in njene gradnike, CSS jim spremeni izgled, JavaScript datoteka pa omogoča funkcionalnost in komunicira s strežnikom. Od njega zahteva seznam obstoječih datotek na strežniku, njihovo velikost in ime ter implementira funkcionalnost gumbov in drugih elementov. Na sliki 4.2 je prikazana slika izgleda aplikacije.



Slika 4.2: Izgled aplikacije.

4.4.3 Avtomatizacija

Namesto ločene lokalne skripte sem za potrebe testiranja, dodal funkcionalnost avtomatizacije kar direktno v aplikacijo. Na strani odjemalca sem dodal v html datoteki dodaten gumb **Testiraj**. Ob pritisku gumba se na strežnik pošlje zahteva po testiranju, ki sproži avtomatsko nalaganje izbrane slike. Slike se naložijo večkrat v ciklu kjer se izvede nalaganje, brisanje, in čakanje določeno število milisekund pred naslednjo ponovitvijo. V samem programu na strežniku lahko poljubno določimo število ponovitev nalaganj ter čas, ki določa zakasnitev med cikli (in s tem frekvenco nalaganja bremena). Na odjemalcu sem napisal še dodatno funkcijo, ki se pokliče, ko želimo izmeriti čas potovanja zahtevka na strežnik.

V nadaljevanju je prikazan del izvorne kode, ki skrbi za avtomatizacijo (Listing 4.1, 4.2, 4.3 in 4.4).

```
<form action="/nalozi" method="post" enctype="multipart/form-data">
      <input type="file" id="datoteka" style="height: 50px; margin-right: 20px;"
          name="upload">
      <button id="nalozi" class="rob senca" type="submit">Naloži novo
          datoteko</button>
      <button id="test" class="rob senca" type="submit"
          formaction="/testiraj">Testiraj</button>
</form>
```

Listing 4.1: Del Html datoteke, ki pošlje zahtevo /testiraj na strežnik.

```
var streznik = http.createServer(function(zahteva, odgovor) {
```

```
    if (zahteva.url === '/') {
        posredujOsnovnoStran(odgovor);
    } else if (zahteva.url === "/testiraj") {
        console.log("----TESTIRAM----------");
        console.log("----delay: "+postanek+" ms");
        console.log("----stevilo datotek: "+N);
        testirajDatoteko(zahteva, odgovor);
    }
});
```

Listing 4.2: Tukaj strežnik sprejme zahtevo /testiraj in požene funkcijo testirajDatoteko.

```
function testirajDatoteko(zahteva, odgovor){
    zacetniPrenos(zahteva, odgovor, function(){
        brisiDat(odgovor, dataDir+ime[1]+koncnica[1], function(){
            console.log("*");
            for(var i = 0;i < N;i ++){
                cikel(i, zahteva, odgovor);
            }
        });
    });
}

function cikel(i, zahteva, odgovor) {
  setTimeout(function() {
      var t = process.hrtime();
      naloziDat(i, zahteva, odgovor, zacasnaPot, function(indeks){
        t = process.hrtime(t);
        console.log(t[0]+((t[1] / 1000000).toFixed(3))+"; ");
        brisiDat(odgovor, dataDir+ime[1]+indeks+koncnica[1], function(){
        });
    });
  },i*postanek);
}

var naloziDat = function(indeks, zahteva, odgovor, zacasnaPot, callback) {
    fs.copy(zacasnaPot, dataDir + ime[1] + indeks + koncnica[1],
        function(napaka) { //kopiraj dat iz zacasnaPot v dataDir+datoteka
        if (napaka) {
            console.log("error nalaganje");
        } else {
            callback(indeks);
        }
    });
}

var brisiDat = function(odgovor, absolutnaPotDoDatoteke, callback){
    fs.unlink(absolutnaPotDoDatoteke, function(err) {
```

```
        if (err) {
            return console.error(err);
        } else {
            callback();
        }
    });
}

var zacetniPrenos = function (zahteva, odgovor, callback){
    var t = process.hrtime();
    var form = new formidable.IncomingForm();

    form.parse(zahteva, function(napaka, polja, datoteke) {
        util.inspect({fields: polja, files: datoteke});
    });

    form.on('end', function(fields, files) {

        var imeReg = /(.+?(?=\..*))/;
        var koncnicaReg = /.*(\..*)/;
        zacasnaPot = form.openedFiles[0].path;
        var datoteka = form.openedFiles[0].name;

        t = process.hrtime(t);
        console.log("cas priprave= "+t[0]+((t[1] / 1000000).toFixed(3)));

        ime = datoteka.match(imeReg);
        koncnica = datoteka.match(koncnicaReg);

        var velikost = form.openedFiles[0].size;
        console.log("----velikost: "+ velikost + " b ");
        console.log("zacetna datoteka: "+ zacasnaPot+ " "+ datoteka);
        fs.copy(zacasnaPot, dataDir + datoteka, function(napaka) {
            if (napaka) {
                posredujNapako500(odgovor);
            } else {
                posredujOsnovnoStran(odgovor);
                callback();
            }
        });
    });
}
```

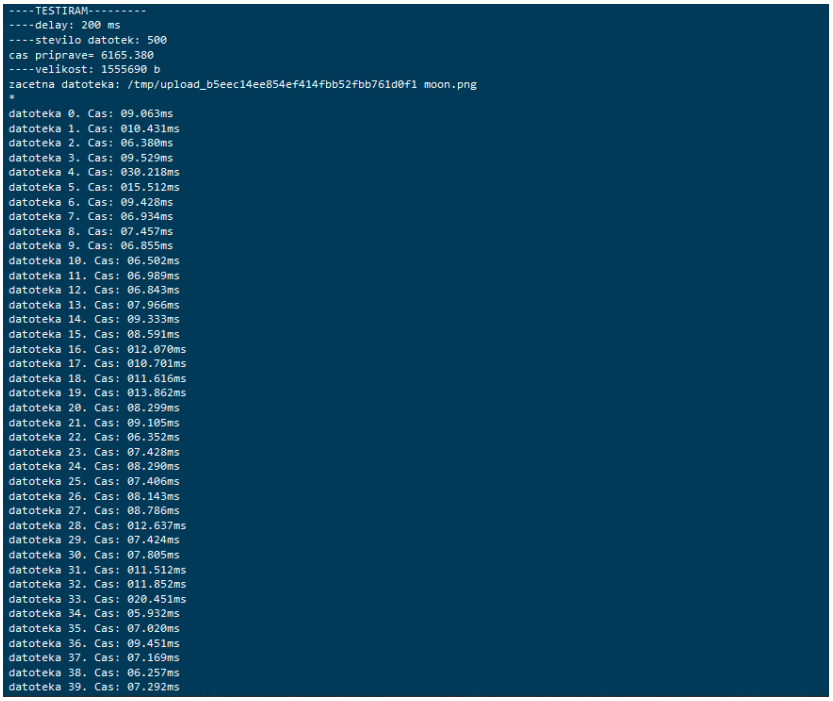
Listing 4.3: Funkcija testirajDatoteko N-krat požene cikel nalaganj in brisanj ter izpiše rezultate meritev. V funkciji se kličejo še pomožne funkcije za nalaganje in brisanje datoteke.

```
  var posljiZahtevek = function(event) {
    var start = new Date().getTime();
```

```
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
       if (xhttp.readyState == 2 && xhttp.status == 200) {
          var end = new Date().getTime();
          console.log(end-start);
       }
    };
    var t1 = new Date().getTime();
    xhttp.open("GET", "/zahtevek/"+t1, true);
    xhttp.send();
}

document.querySelector("#t1").addEventListener('click', posljiZahtevek);
```

Listing 4.4: Del kode na odjemalcu, kjer pošljemo zahtevek na strežnik in merimo čas potovanja zahtevka.

Ta funkionalnost mi omogoča pridobivanje velikega števila meritev za poljubna bremena. Rezultati se izpišejo v obliki primerni za uvoz v Microsoft Excel, kjer lahko hitro prikažem podatke v obliki grafov in iz njih izvlečem zaključke o delovanju aplikacije. Na sliki 4.3 je primer izpisa.



Slika 4.3: Del izpisa meritev z avtomatskim nalaganjem datotek.

4.5 Breme

Zmogljivost aplikacije sem testiral s prenašanjem datotek na strežnik *(upload)*. Za osnovno testno datoteko sem izbral manjšo sliko veliko 1,48 MB. Manjša velikost slike skrajša merjenje časov prenosa in omogoča pošiljanje večjega števila slik, saj je prostor na Cloud9 v brezplačni različici omejen na 1 GB. Zaporedno sem prenesel večjo količino slik z različnimi frekvencami prenosa in tako opazoval vpliv na zmogljivost aplikacije.

4.6 Metrike

Za lažje razumevanje časov T_1, T_2 in T_3, ki jih merim pri testiranju, je na sliki 4.4 prikazan časovni diagram nalaganja (*upload*) slike na strežnik. T_1 je čas prenašanja zahteve z odjemalca na strežnik.

T_2 je čas obdelave zahteve na strežniku. Pri meni se deli na dva dela; $T_{2.1}$ in $T_{2.2}$. $T_{2.1}$ predstavlja pripravo datoteke na prenos (iz polja za izbiro datoteke prebere ime in lokacijo na lokalnem računalniku) in dejanski prenos datoteke iz računalnika na strežnik. $T_{2.2}$ je čas režije prenešene datoteke na strežniku (prikaz na spletni strani, določitev velikosti, po potrebi tudi preimenovanje ali brisanje). T_3 je čas potreben za prenos odgovora s strežnika k odjemalcu. Ta odgovor je lahko potrditev uspešnega prenosa ali koda napake.

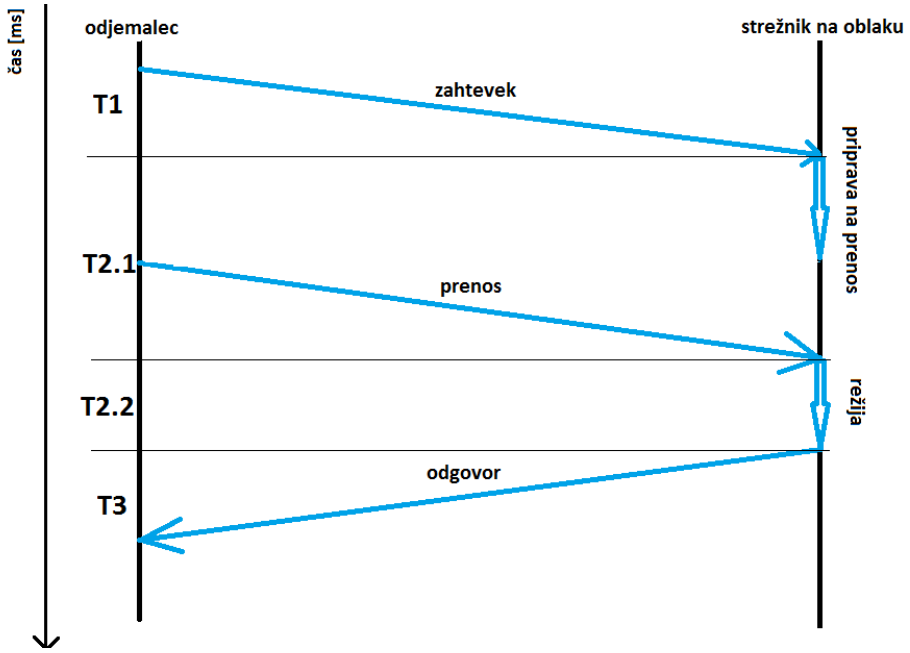
T_2 sem lahko pridobil z avtomatskim testiranjem opisanim v prejšnjem poglavju. Ker se obdelava zahteve v celoti dogaja na strežniku, sem lahko na željenih mestih preprosto postavil časovne točke in izmeril potreben čas. T_1 sem tudi pridobil programsko saj v Javascriptu obstaja dogodek (*event*), ki javi odjemalcu, kdaj je njegova zahteva prispela na strežnik.

Težave sem imel pri pridobivanju T_3. Pri merjenju časa so bile ure na strežniku in odjemalcu neusklajene in so pokvarile natančne meritve. Zato sem uporabil Firefox Network Monitor, ki prikazuje različne čase pošiljanja in obdelave zahtev. Zame je bil pomemben predvsem čas **Sprejemanje**, ki je definiran kot čas sprejemanja odgovora. Točno to pri meni predstavlja čas T_3. Ker nisem mogel pridobiti T_3 z avtomatskim testiranjem sem „ročno“ za vsako zahtevo pogledal čas v Network Monitorju (kot je prikazano na sliki 4.5 in ocenil potreben čas.

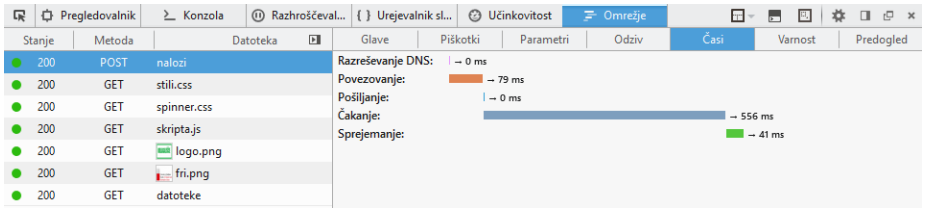
4.7 Rezultati meritev

Odločil sem se, da zaporedno pošiljam 500 datotek velikosti 1,48 MB s štirimi različnimi frekvencami (različno velike zakasnitve med zaporednimi prenosi) in analiziram podatke teh meritev. Meritve so bile izvedene v petek 6.5., soboto 7.5. in v ponedeljek 9.5. Ob teh dnevih sem izvedel meritve trikrat na dan ob 11:00, 16:00 in 21:00. S tem sem želel zmanjšati zunanji vpliv ure in dneva na rezultate.

Čas T_2 predstavlja glavni in najzanimivejši del tega testiranja, saj je najbolj odvisen od velikosti datoteke in frekvence pošiljanja datotek na strežnik. Na



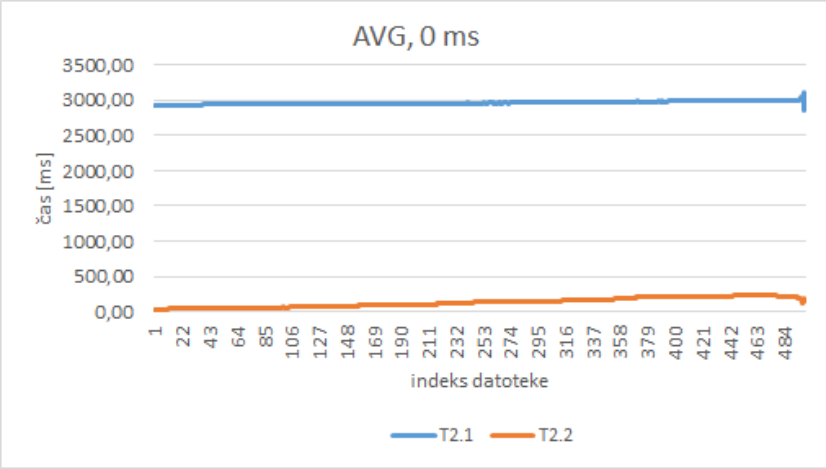
Slika 4.4: Časovni diagram poteka nalaganja datoteke na strežnik.



Slika 4.5: Prikaz različnih časov za zahtevo /naloži v Firefox Network Monitor.

tem času bom lahko tudi opazil delovanje predpomnilnika na oblaku (če je sploh prisoten).

Spodaj so prikazani grafi za čas T_2 ob različnih frekvencah. Za vsako frekvenco sem naredil maksimalni in minimalni graf, ter graf, ki prikazuje povprečje vseh meritev za to frekvenco. Ko sem izvajal meritve sem opazil, da je čas $T_{2.2}$ zanemarljiv v primerjavi z $T_{2.1}$ (kot lahko vidimo na sliki 4.6), zato sem ga izpustil iz prikaza meritev.

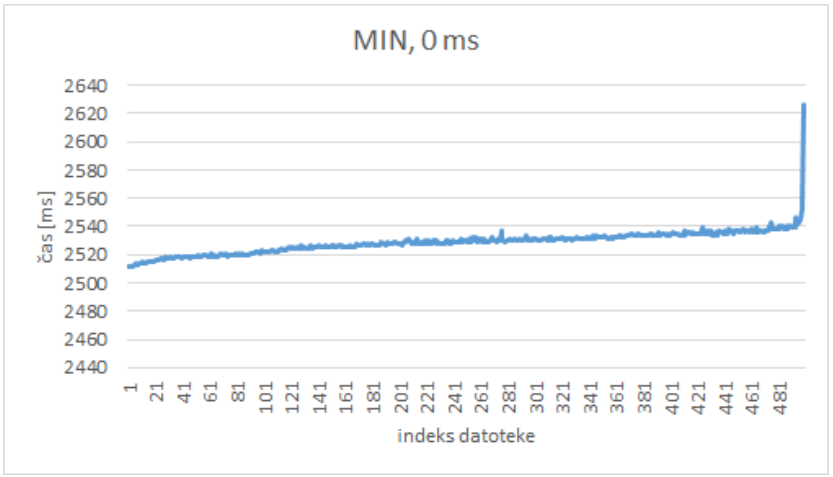


Slika 4.6: Povprečni graf za zakasnitev 0 ms za $T_{2.1}$ in $T_{2.2}$.

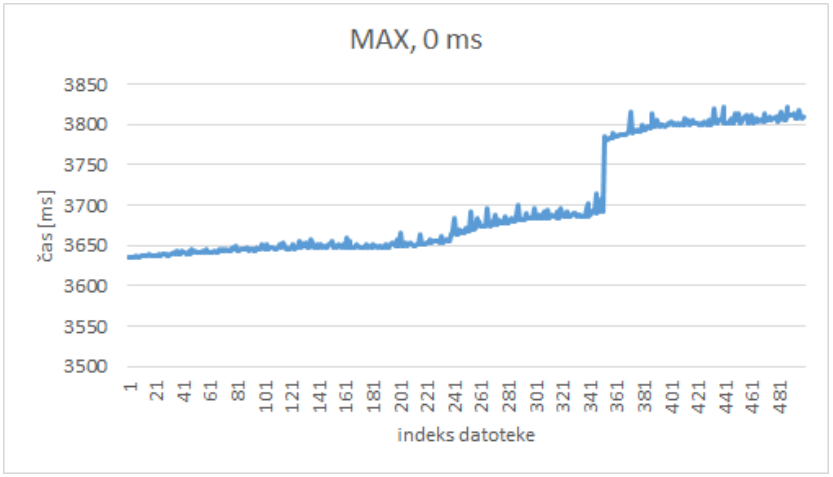
4.7.1 Zakasnitev 0 ms

Najprej sem pognal testiranje brez zakasnitve, torej vseh 500 datotek se bo začelo nalagati praktično naenkrat. Verjetno je rezultat takšen, ker se v zanki datoteke zaporedno prenašajo skoraj naenkrat, vendar še vedno z majhnim zamikom. Prva datoteka se začne prenašati, nato nekaj nanosekund kasneje druga itd. Tako tudi končajo prenos: prejšnja malo prej kot naslednja, kar privede do linearnega naraščanja. Čas prenosa je večji kot, če bi se prenašala vsaka posebej saj si med sabo podaljšujejo prenos.

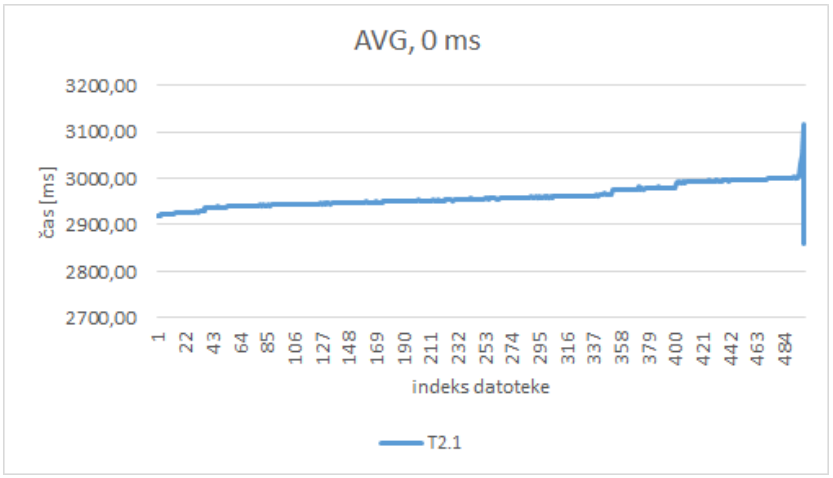
Minimalni graf (slika 4.7) je bil izmerjen v soboto ob 21:00, maksimalni (slika 4.8) v petek ob 16:00. Slika 4.9 prikazuje graf povprečja vseh meritev za to frekvenco.



Slika 4.7: Minimalni graf za zakasnitev 0 ms. Povprečni čas je 2528,92 ms.



Slika 4.8: Maksimalni graf za zakasnitev 0 ms. Povprečni čas je 3701,08ms.

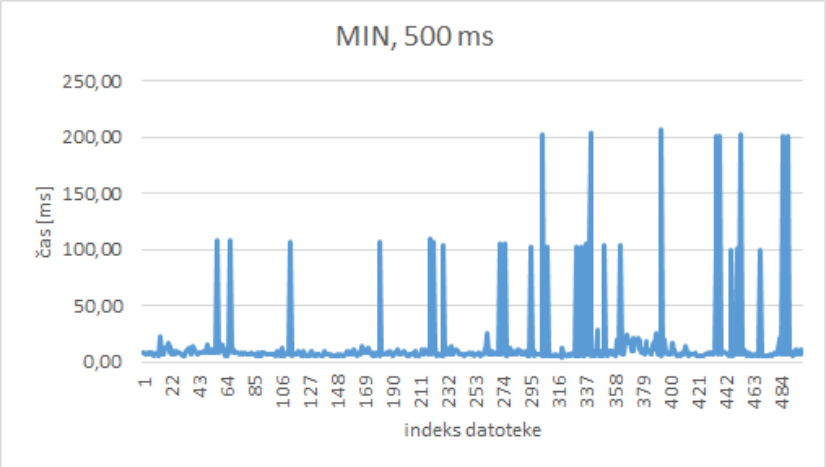


Slika 4.9: Povprečni graf za zakasnitev 0 ms. Povprečni čas je 2960,7 ms.

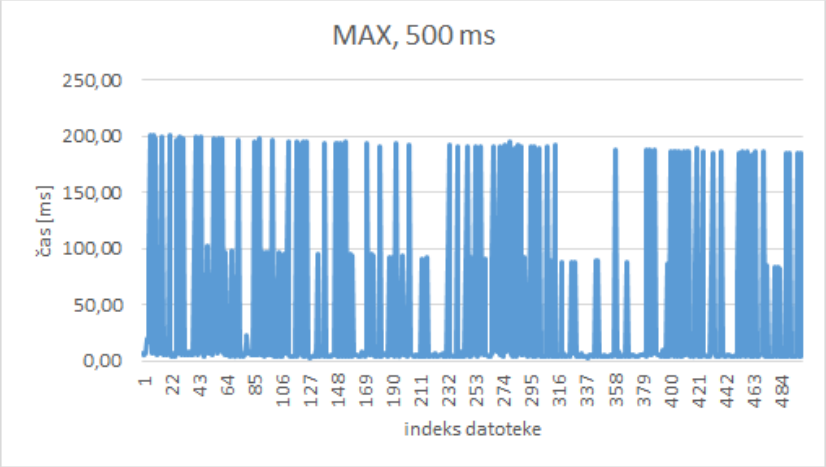
4.7.2 Zakasnitev 500 ms

Nato sem uporabil zakasnitev veliko 500 ms, ki je dovolj velika, da se bo vsaka datoteka prenesla posebej, preden se bo začel prenos naslednje datoteke.

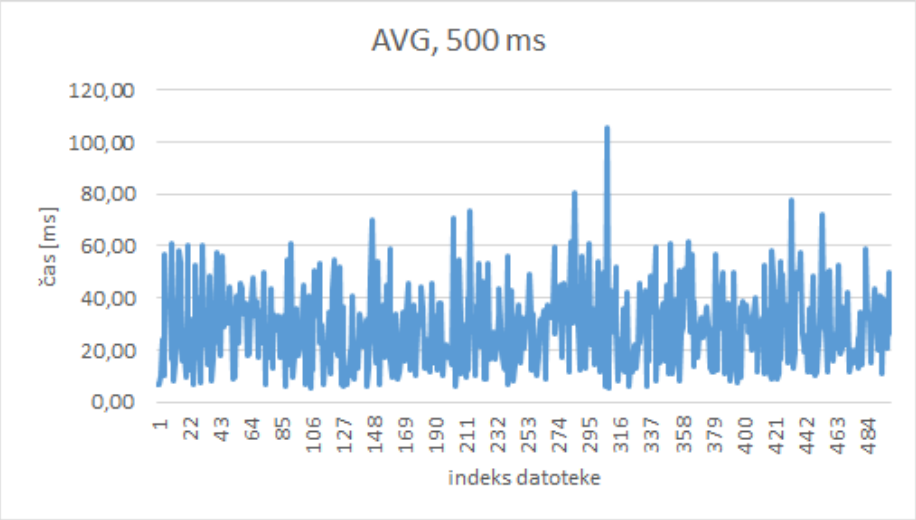
Pričakoval sem enakomerno krivuljo grafa, saj se prenos posameznih datotek ne bo prekrival, in to tudi dobil. Minimalni graf (slika 4.10) je bil izmerjen v petek ob 21:00, maksimalni (slika 4.11) v ponedeljek ob 21:00. Slika 4.12 prikazuje graf povprečja vseh meritev za to frekvenco.



Slika 4.10: Minimalni graf za zakasnitev 500 ms. Povprečni čas je 15,47 ms.



Slika 4.11: Maksimalni graf za zakasnitev 500 ms. Povprečni čas je 41,08 ms.

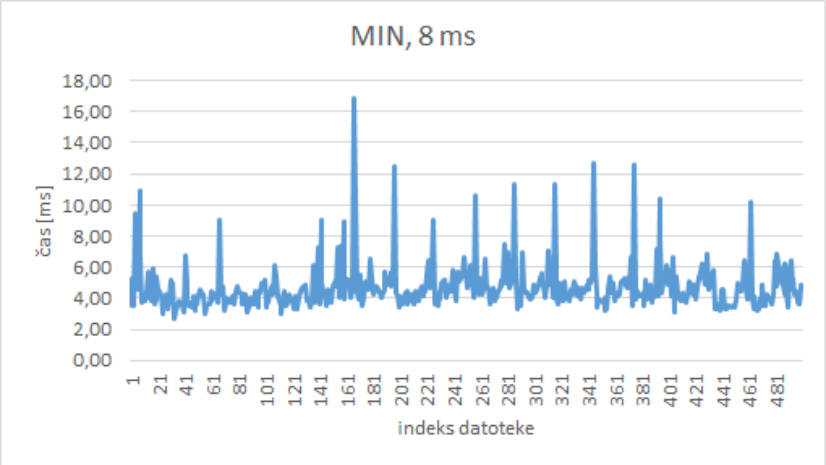


Slika 4.12: Povprečni graf za zakasnitev 500 ms. Povprečni čas je 28,48 ms.

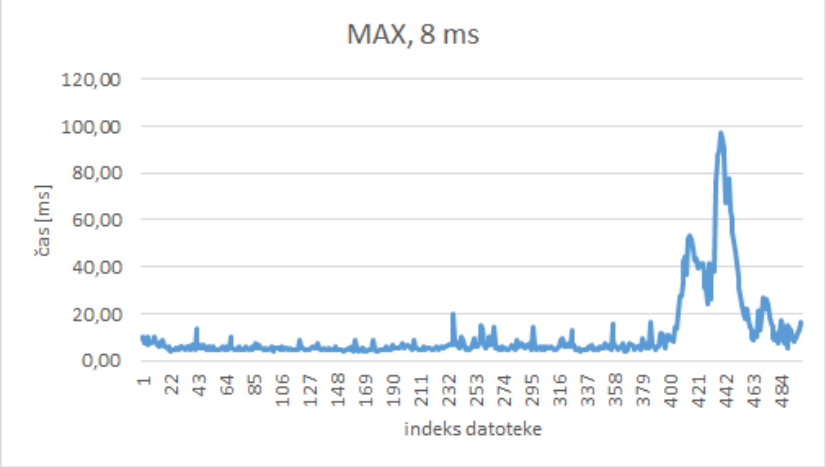
4.7.3 Zakasnitev 8 ms

Zakasnitev sem postopoma zmanjševal, da bi našel mejo pri kateri je graf še enakomeren (kot pri zamiku 500 ms) preden se spremeni v naraščajočega (kot brez zamika). Pri tem sem opazil, da se z zmanjšujem zakasnitve rahlo zmanjšuje tudi čas prenosa datoteke (čeprav zakasnitev ni všteta v meritve). Verjetno se to dogaja zaradi predpomnilnika na oblaku. Pri manjšem presledku med prenosi predpomnilnik še pomni, pri večji zakasnitvi pa se ponastavi. Po nadaljnjem testiranju sem odkril, da se to zgodi pri približni zakasnitvi 200 ms.

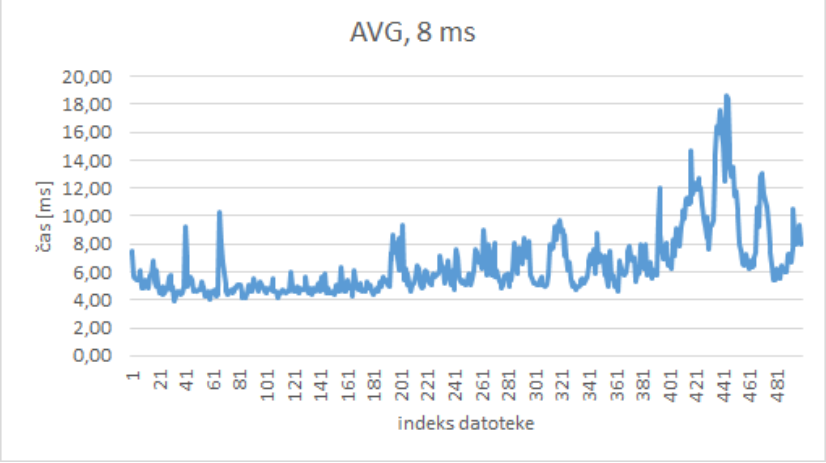
Minimalni graf (slika 4.13) je bil izmerjen v ponedeljek ob 16:00, maksimalni (slika 4.14) v petek ob 16:00. Slika 4.15 prikazuje graf povprečja vseh meritev za to frekvenco. Pri njih lahko vidimo, da se je čas prenosa res zmanjšal v primerjavi z 500 ms.



Slika 4.13: Minimalni graf za zakasnitev 8 ms. Povprečni čas je 4,7 ms.



Slika 4.14: Maksimalni graf za zakasnitev 8 ms. Povprečni čas je 11,22 ms.

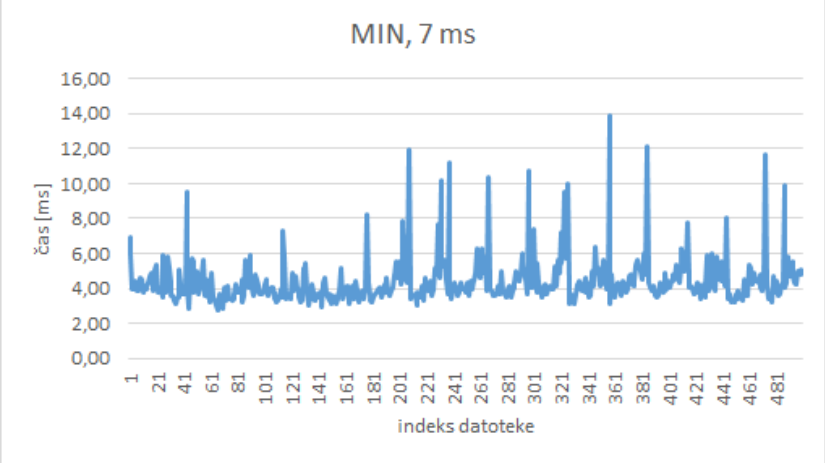


Slika 4.15: Povprečni graf za zakasnitev 8 ms. Povprečni čas je 6,53 ms.

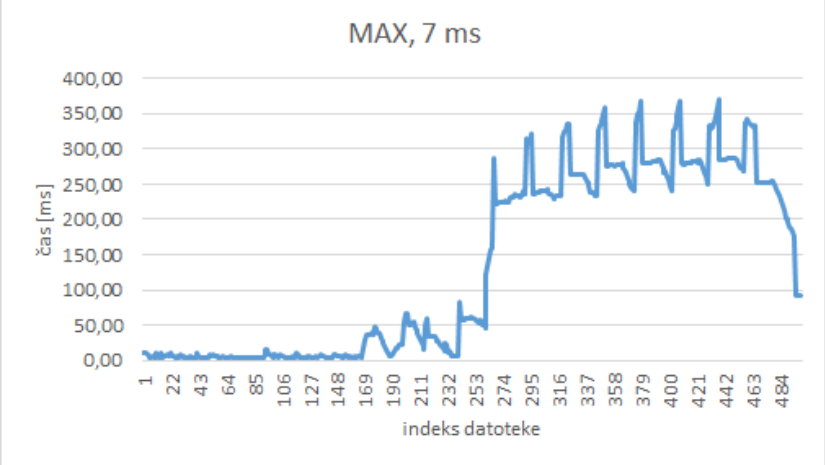
4.7.4 Zakasnitev 7 ms

Seveda pri določeni zakasnitvi pride do naraščanja časa prenosa. Z testiranjem sem to mejo določil pri 8 ms. Pri tej zakasnitvi je graf enakomeren, nato pa naraščujoč. To se dogaja, ker se pri tej zakasnitvi prenos prejšnje datoteke nadaljuje v prenos trenutne, kar povzroči čedalje večje čase.

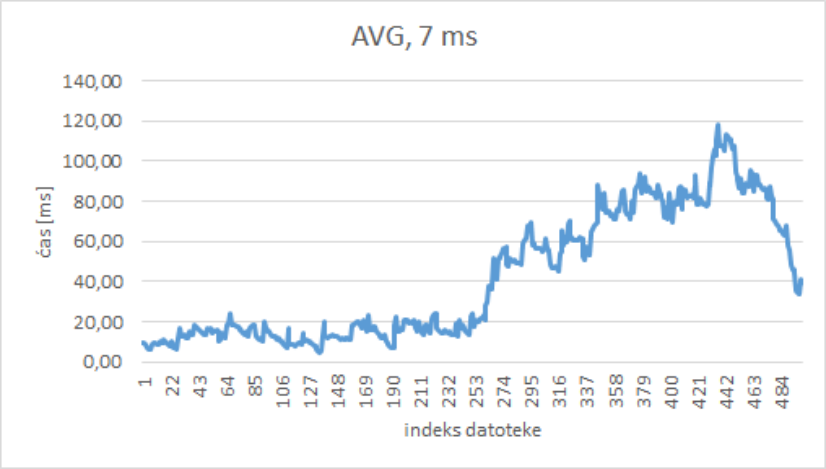
Minimalni graf (slika 4.13) je bil izmerjen v soboto ob 16:00, maksimalni (slika 4.14) v soboto ob 11:00. Slika 4.15 prikazuje graf povprečja vseh meritev za to frekvenco. V soboto ob 16:00 so se datoteke prenašale tako hitro, da je graf postal naraščujoč šele pri 3 ms.



Slika 4.16: Minimalni graf za zakasnitev 7 ms. Povprečni čas je 4,04 ms.



Slika 4.17: Maksimalni graf za zakasnitev 7 ms. Povprečni čas je 135,22 ms.



Slika 4.18: Povprečni graf za zakasnitev 7 ms. Povprečni čas je 42,09 ms.

4.7.5 Datoteke drugih velikosti

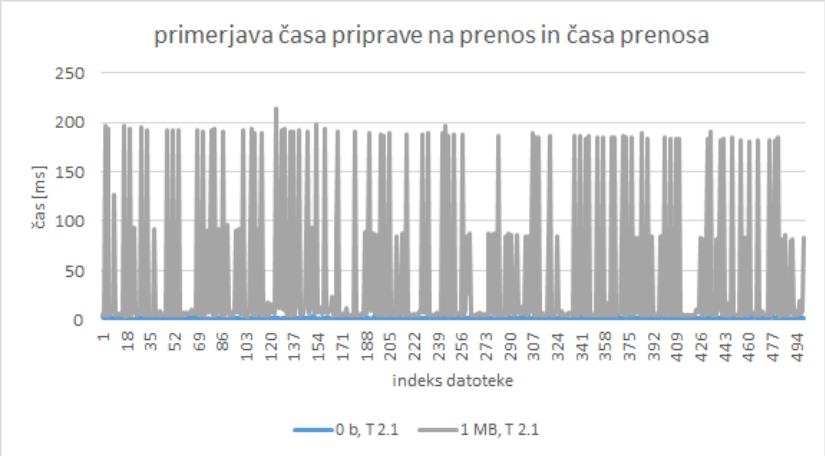
Na koncu sem testiral aplikacijo še za dve različni velikosti datotek, da bi bolje razumel razmerje med časovi T. Vse naslednje meritve so bile izvedene z zamikom 500 ms, saj me je zanimal predvsem čas priprave in prenosa posamezne datoteke, ne pa vpliv časovnega zamika med prenosi. Kot sem omenil že prej, se bo pri zamiku 500 ms vsaka datoteka prenesla posebej. Meritve so bile izvedene ob istih dnevih in urah kot prejšnja testiranja.

Ker čas $T_{2.1}$ vsebuje še pripravo datoteke na prenos, me je zanimalo koliko časa pravzaprav porabi aplikacija za pripravo v primerjavi z dejanskim prenosom datoteke. Da bi to izmeril sem se odločil poslati datoteko velikosti 0b (torej prazno datoteko). Pri takšnem testu se bo prenos zgodil instanto, ostal pa bo samo čas priprave. Za merjenje časa prenosa sem uporabil isto datoteko kot pri prešnjih testiranjih. Izkazalo se je, da je delež časa priprave podatkov bistveno manjši od prenosa datoteke (kot prikazuje slika 4.19).

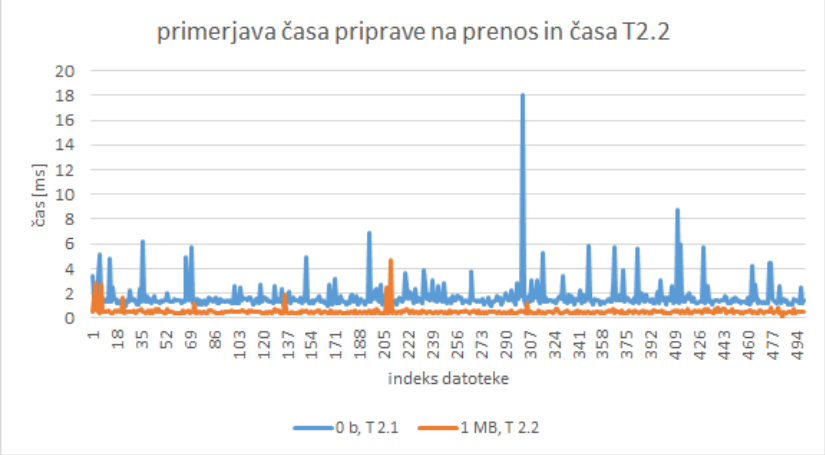
Povprečen čas prenosa znaša 42,19 ms, povprečen čas priprave pa le 1,70 ms, maksimalen 18,01 ms in minimalen 0,96 ms.

Ker se je izkazal čas priprave podatkov tako majhen me je zanimalo kako se primerja z časom $T_{2.2}$ (časom ažuriranja na strežniku po sprejetju poslane datoteke). Pri merjenju časa $T_{2.2}$ sem seveda uporabil datoteko velikosti 1,48 MB, kot pri prejšnih testih, saj je pri prazni datoteki čas ažuriranja neznaten. Rezultat meritev je viden na sliki 4.20. Izkazalo se je, da je čas priprave datoteke na prenos večji od časa ažuriranja.

Čas priprave je isti kot prej: povprečen 1,70 ms, maksimalen 18,01 ms in minimalen 0,96 ms, medtem ko je povprečen čas $T_{2.2}$ 0,52 ms, maksimalen 4,46 ms in minimalen 0,07 ms.

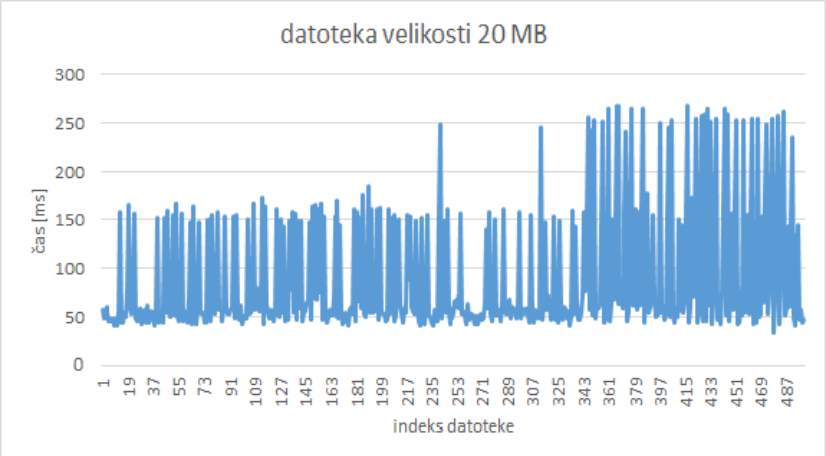


Slika 4.19: Povprečni graf časov priprave in prenosa, pri zamiku 500 ms.



Slika 4.20: Povprečni graf časov priprave na prenos in ažuriranja po prenosu, pri zamiku 500 ms.

Nato sem izvedel testiranje z datoteko velikosti 20 MB. Zanimalo me je koliksen bo čas $T_{2.1}$ če pošiljam veliko večjo datoteko, predvsem pa sem potreboval meritve, da lahko primerjam razmerje časov $T_1 : T_2 : T_3$ pri veliki in majhni datoteki. Na sliki 4.21 so vidni rezultati meritev za večjo datoteko. Povprečen čas je 88,49 ms, maksimalen 268,13 ms in minimalen 32,69 ms.



Slika 4.21: Povprečni graf časa $T_{2.1}$ za datoteko velikosti 20 MB, pri zamiku 500 ms.

4.7.6 Razmerje $T_1 : T_2 : T_3$

Za čas T_1 in T_3 nisem naredil grafov saj nisem mogel pridobiti dovolj meritev in ker nista tako pomembna pri tem testiranju (potrebujem ju le za razmerje s časom T_2), zato sem ju samo ocenil. Kljub temu sem meritve za T_1 in T_3 pridobil ob istih dnevih in časih kot za T_2. T_1 sem vsakič pomeril 50-krat, T_3 pa 20-krat.

Minimalni T_1 je znašal 82 ms, maksimalni 171 ms, povprečni pa 103 ms. Za T_3 sem izmeril minimalni čas 12 ms, maksimalni 75 ms in povprečni 39 ms. Za izračun razmerja med T časi sem vzel povprečna časa T_1 in T_3 ter povprečni čas T_2 pri zakasnitvi 500 ms, saj se tako izognem vplivu predpomnilnika in vplivu frekvence (pri tej zakasnitvi se prenos obnaša kot, da bi pošiljali vsako datoteko posebej, saj ne vplivajo ena na drugo). Čas T_2 je sestavljen iz $T_{2.1}$ (povprečni čas 28,48 ms) in $T_{2.2}$ (povprečni čas 0,55 ms) zato je razmerje $T_1 : T_{2.1}+T_{2.2} : T_3$ približno enako $3.6 : 1 : 1.3$.

Pri merjenju časov pri večji datoteki (20 MB) se čas $T2$ poveča. Povprečni čas $T_{2.1}$ je 88,49 ms in povprečni $T_{2.2}$ je 4,90 ms, medtem ko čas $T1$ in $T3$ ostaneta približno enaka. Razmerje za večjo datoteko je zato enako $2.6 : 2.4 : 1$. Večja kot je datoteka večji je čas T_2.

4.8 Zaključek

Pri analiziranju oblačne storitve Cloud9 IDE sem se predvsem osredotočil na preverjanje zmogljivosti prenašanja datotek. Skupaj je bilo pri tem testiranju naloženih 36.000 datotek. Po dobljenih rezultatih lahko sklepam, da storitev ima nek predpomnilnik, ki pomni do določenega časa in ima opazen vpliv na zmogljivost.

Iz rezultatov se vidi precej velik vpliv dneva in ure na hitrost prenosa, kar me je presenetilo, saj sem mislil, da v današnjem času, kjer nas je večina konstanto povezanih na internet, dan in čas merjenja ne bo imel takšnega vpliva. Ne morem pa z zagotovostjo reči, da je to posledica zasedenosti omrežja zaradi drugih uporabnikov ali pa samo naključje. Za to bi bilo potrebno narediti veliko več meritev.

Zanimivo je tudi razmerje $T_1 : T_2 : T_3$. Pri tako majhni datoteki (1,48 MB), kakršno sem prenašal pri večini testiranj, se opazi, da je v bistvu čas obdelave zahteve T_2 najmanjši. Pošiljanje zahteve (T_1) traja skoraj štirikrat več, T_3 pa je približno enak T_2. Pri takšni velikosti torej velja: $T_3 > T_2 > T_1$. Pri večji datoteki so časi drugačni, T_2 se poveča, T_1 in T_3 pa ostaneta ista.

Poglavje 5

Zmogljivostna analiza Wiki aplikacije

Jan Robas, Žiga Pirih, Rok Oblak

5.1 Opis problema

Vsled zmogljivostne analize smo ustvarili Wiki aplikacijo, kjer ima vsak uporabnik dostop do štirih osnovnih operacij, in sicer dodajanje strani, brisanje strani, pregled poljubne strani in urejanje poljubne strani. Pri pregledovanju uporabnik zahteva določeno vsebino strani, medtem ko pri urejanju posamezno vsebino prepiše z novo. Uporabnik lahko išče po naslovih strani. Vsaka operacija terja vhodno-izhodne operacije na bazi podatkov.

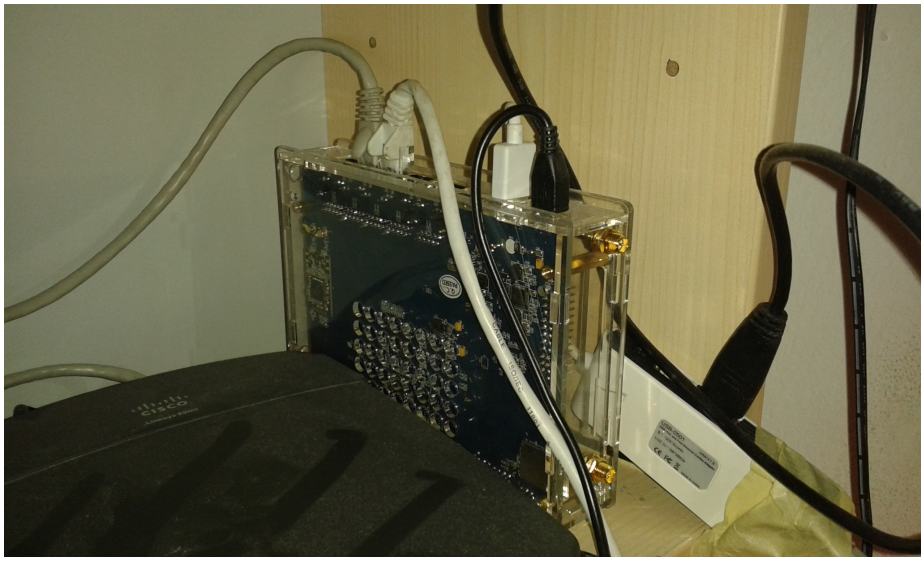
V ta namen smo postavili strežnik, na katerem teče spletna aplikacija, ki obdeluje zahteve in upravlja z dokumentno usmerjeno podatkovno bazo. Za odjemalca smo napisali ločeno aplikacijo, ki ustvarja zahtevke za primere različnih situacij. Tretji člen v shemi je predpomnilniški strežnik oziroma predpomnilniška storitev [34] (angl. *caching service*), ki predpomni zahteve odjemalca in v primeru večjega števila enakih zahtevkov ne obremenjuje po nepotrebnem aplikacijskega strežnika. Testne scenarije smo preizkusili z in brez omenjenega predpomnilniškega strežnika in s tem ovrednotili uporabnost le-tega.

5.2 Rešitev problema

V tem razdelku predstavimo uporabljene programske tehnologije in infrastrukturo, ki smo jih uporabili.

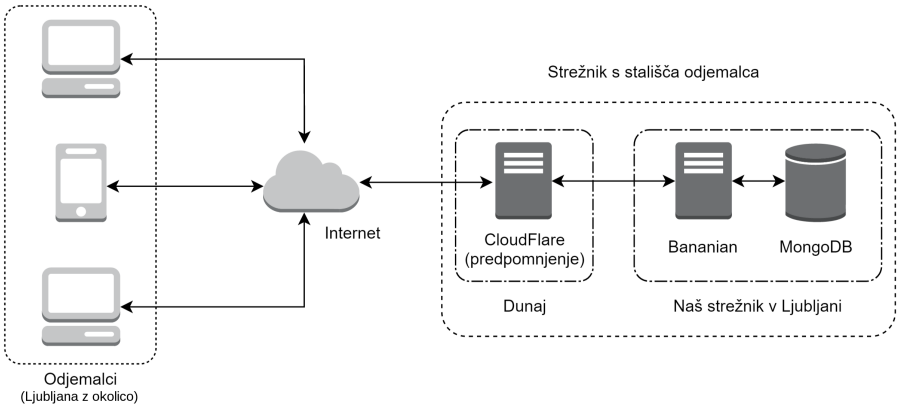
5.2.1 Izbira strežnika in ponudnika oblačnih storitev

Aplikacija teče na lastnem strežniku. Strežnik je Banana Pi R1 [33]. Ima dvojedrni procesor ARM Cortex-A7 s taktom 1 GHz ter 1 GB pomnilnika. Na njem teče operacijski sistem Bananian (različica Debiana, prilagojena za računalnike Banana Pi). Kot spletni strežnik uporabljamo nginx, s katerim prihajajoči promet preusmerimo iz vrat 80 na tista vrata, ki jih uporablja aplikacija. Strežnik se nahaja v enem izmed študentskih domov v Ljubljani. Hitrost internetne povezave je na papirju 100 Mb/s v vsako stran, v praksi pa je bila največja izmerjena hitrost prenosa v smeri proti internetu zgolj med 45 - 65 Mb/s. Količina prenosa, ki ga lahko opravimo od strežnika proti internetu, je omejena s 100 GB na teden. Od tega ostale aktivnosti na strežniku zahtevajo 10 - 20 GB, vendar je popačenje rezultatov ob zadostnih ponavljanjih testov zanemarljivo.



Slika 5.1: Naš strežnik na lokaciji v Ljubljani.

Za predpomnenje (angl. *caching*) smo uporabili storitev CloudFlare. CloudFlare smo izbrali, ker je brezplačen in ker nam lahko izpiše grobe podatke o številu predpomnjenih in nepredpomnjenih zahtev. CloudFlare izvaja predpomnjenje na svojih strežnikih, ki so optimalno blizu tako uporabniku kot strežniku. Nam najbližji strežnik ponudnika CloudFlare, preko katerega gre ves promet do našega strežnika (brez izjem), je na Dunaju. Za primerjavo smo uvedli tudi dostopno točko, kjer gre promet preko CloudFlare strežnikov, brez da bi se izvajalo predpomnjenje in dostopno točko, ki gre neposredno do našega strežnika.



Slika 5.2: Diagram strežniške arhitekture naše aplikacije.

5.2.2 Izbira tehnologij

Node.js

Strežniška aplikacija je napisana v jeziku JavaScript in teče v okolju Node.js. Node.js omogoča asinhrone vhodno-izhodne operacije s pomočjo dogodkovno gnanega programiranja (angl. *event-driven programming* [32]) in posledično ponuja visoko propustnost in skalabilnost realno-časovnih spletnih aplikacij. Zaradi svoje razširjenosti in velike izbire knjižnic omogoča preprosto razvijanje in nameščanje na strežnik.

MongoDB

Za shranjevanje podatkov smo uporabili dokumentno usmerjeno bazo MongoDB. Vsak zapis v bazi je shranjen kot objekt v obliki, podobni JSON (*JavaScript Object Notation*). Wiki stran bi na primer lahko vsebovala tako kraje kot ljudi. Ljudje bi imeli na primer podatek o datumu rojstva, kraji pa bi imeli podatek o nadmorski višini. Pri dokumentno usmerjeni bazi nam ni potrebno vnaprej določiti strukture, tako kot moramo to narediti pri uporabi relacijske baze. To nam omogoča veliko prostora za razširitve podatkov. Dodatna prednost, predvsem v našem primeru pa je, da je baza MongoDB zelo dobro podprta v Node.js.

5.2.3 Implementacija aplikacije

Programski vmesnik deluje po principih REST [30]. Ti principi so smernice za arhitekturo spletnih aplikacij. Po principu REST uporabljamo standardne HTTP metode (GET, POST, DELETE, itd.), ki predstavljajo operacije kot so branje, dodajanje in brisanje. Vsaka stran ima svoj URL naslov v obliki */wiki/ime_strani*, kjer je *ime_strani* nadomestna beseda (angl. *placeholder*)

za poljubno ime strani. Vmesnik je brezstanjski (angl. *stateless*), kar pomeni da ne vodimo evidence o odjemalcih med zahtevki. Vsak zahtevek vsebuje vse potrebne informacije za izvršitev željene operacije. V odgovor strežnika prav tako dodamo še informacije za predpomnilniški strežnik (največja starost strani). Vse to nam je zelo olajšalo testiranje.

Naš vmesnik podpira naslednje operacije:

- dodajanje / spreminjanje strani (POST zahtevek z vsebino na URL že obstoječe ali nove strani s poljubnim imenom),
- brisanje strani (DELETE zahtevek na URL strani),
- pregled strani (GET zahtevek na URL strani).

Vsaka od teh funkcij terja vhodno-izhodno operacijo na bazi podatkov.

5.2.4 Analiziranje zmogljivosti aplikacije

Analiziranje zmogljivosti aplikacije smo izvedli iz dveh vidikov:

- odjemalčeva stran (na različnih odjemalcih),
- strežnikova stran.

Za vnaprej določene scenarije smo generirali testne nabore zahtevkov, v katerih so nastopale različne operacije vmesnika z različnimi parametri. Beležili smo čas odgovora na posamezen zahtevek, povprečni čas odgovora na zahtevek in skupni čas vseh zahtevkov.

S primerjanjem testnega scenarija in analizo obremenjenosti strežnika ter odzivnega časa za odgovor na zahtevek odjemalca smo naredili zaključke o samem delovanju celotne sheme ter o učinkovitosti predpomnilniškega strežnika.

5.2.5 Testiranje

Zavoljo testiranja smo pripravili skripto, ki prejme sledeče parametre:

- URL - dostopna točka, na kateri teče naša Wiki aplikacija,
- število različnih strani, med katerimi skripta naključno izbira,
- število zahtevkov, ki jih skripta izvede,
- čas čakanja med posameznimi zahtevki (v sekundah; kot parameter lahko nastopa tudi necelo število),
- za kakšne zahtevke gre: GET ali POST,
- dolžina vsebine v znakih (v primeru POST zahtevkov, ko urejamo oziroma dodajamo nove strani).

Parameter za spanje med pošiljanjem posameznih zahtevkov je uporaben v primeru, ko hočemo dati strežniku nekaj časa med posameznimi zahtevki, sicer pa je lahko nastavljen na 0 in se zahtevki izvajajo v normalnem, zaporednem načinu.

Skripta vsak zahtevek naslovi na eno izmed strani, katerih število je določeno s prvim argumentom. S tem je mogoče določati porazdelitev zahtevkov med stranmi - veliko razpoložljivih strani in majhno število zahtevkov pomeni veliko verjetnost, da je posamezen zahtevek vedno naslovljen na unikatno stran, medtem ko v obratni situaciji več zahtevkov pade na isto stran. V tem primeru lahko vidimo učinke predpomnjenja v primeru GET zahtevkov.

Na strežniški aplikaciji smo implementirali tudi števec zahtevkov. Z dostopom na URL *wiki/stevec* pridobimo število zahtevkov pred dostopom in števec ponastavimo na 0. Tako smo lahko prešteli, koliko zahtevkov je bilo odgovorjenih s strani predpomnilniškega strežnika. S tem smo dobili dodatno metriko za učinkovitost predpomnjenja - delež zahtevkov, ki je bil predpomnjen oziroma delež zahtevkov, ki jih je moral servirati naš aplikacijski strežnik.

Pripravili smo dve dostopni točki - eno, ki gre preko predpomnilnika in eno, ki predpomnjenja nima. Poleg primerjave učinkovitosti predpomnjenja pri različnih testnih scenarijih smo tako lahko primerjali tudi testni scenarij brez predpomnilniškega strežnika.

5.2.6 Testni scenariji

Ker aplikacija predstavlja Wiki stran, kjer lahko vsak uporabnik pregleduje, spreminja in briše posamezne strani oziroma dokumente v bazi, smo definirali nekaj testnih scenarijev, ki posnemajo realno uporabo take strani. Testni scenariji so sestavljeni iz različnega tipa in števila zahtevkov ter različne velikosti podatkov (velikosti vsebin strani), ki se prenašajo z zahtevki. Tako smo spreminjali velikost bremena in s tem izvedli t. i. *test to pass* in *test to fail*. Vsled tega smo izbrali smiselno zgornjo mejo odziva strežnika na zahtevek. Če operacija traja dlje časa od naše zgornje meje, določimo da je test neuspešen, saj s predolgim čakanjem izgubljamo uporabnike [31]. Testni scenariji so potekali po naslednjem vzorcu:

- Zaporedje zahtevkov za pregled strani (GET): Strežniku posredujemo n zahtevkov za pregled strani, kjer vsak zahtevek naslavlja eno od m strani. Parametra n in m spreminjamo in tako izvedemo različne vrste testov.
- Zaporedje zahtevkov za spreminjanje strani (POST): Strežniku posredujemo n zahtevkov za spreminjanje strani, kjer ima vsak zahtevek naključno vsebino dolžine l in naslavlja eno od m strani. Parametre n, l in m spreminjamo.
- Mešano zaporedje: Sestavljeno je iz zgornjih dveh zaporedij.

Pri testiranju smo obravnavali več različnih scenarijev, kar smo dosegli s spreminjanjem spremenljivk. Spreminjali smo tip zahtevkov - POST in GET,

pri POST zahtevkih pa tudi količino informacije, ki se je prenesla na strežnik. Poskušali smo s primerno velikostjo Wiki strani, nekje med nekaj kilobajti do nekaj deset kilobajtov, z namenom da pride strežniško predpomnjenje do izraza. Za podatke se jemljejo naključni nizi iz */dev/random*, saj njihova vsebina ni pomembna. Spreminjali smo tudi število obravnavanih strani. Kot hipotezo smo vzeli, da je predpomnjenje bolj učinkovito pri velikem številu zahtevkov na majhno število strani.

Strežnik smo preizkušali tako, da smo ga iz ene strani čim bolj obremenili z velikim številom hkratnih zahtevkov, medtem pa iz druge strani izvajali zahtevke in merili njihov povprečni čas v primerjavi s situacijo, kjer strežnik ni obremenjen iz druge strani.

5.3 Metrike

Pri testnih scenarijih smo obravnavali različne metrke, s katerimi smo določali parametre testov ali pa jih merili v rezultatih testov. Glavne izmed teh metrik so naslednje:

- Število predpomnjenih in nepredpomnjenih strani: pri testih z večjim številom zahtevkom smo opazovali absolutno število strani oziroma zahtevkov, ki so bile predpomnjeni in jih je zato namesto aplikacijskega strežnika stregel strežnik CloudFlare.

- Delež predpomnjenih in nepredpomnjenih zahtevkov: opazovali smo deleže vseh zahtevkov, ki so bili predpomnjeni na strežniku CloudFlare od skupine več izvedenih zahtevkov v enem kosu.

- Povprečni čas posameznega zahtevka: ker zaradi velikega števila različnih dejavnikov čas za izvedbo zahtevkov niha, smo povprečili čase preko večjega števila zahtevkov in to večkrat ponovili.

- Nihanje časa za izvedbo zahtevkov v krajših časovnih in daljših časovnih intervalih: na krajših intervalih čas niha zaradi manjših okoljskih in fizičnih dejavnikov na linijah, na daljših pa ima večji vpliv sama obremenjenost linij zaradi različne prisotnosti drugih uporabnikov spleta.

- Število zaporednih zahtevkov v enem testiranju z uporabo testne skripte: s testno skripto smo pošiljali določeno število zaporednih zahtevkov, kar je vplivalo na dolžino časa, ko je bil strežnik zaposlen s streženjem.

- Število hkratno naslovljenih zahtevkov iz enega odjemalca: s skripto smo generirali tudi posamezne kose hkratno naslovljenih zahtevkov in jih zaporedno ponavljali; to je poleg dolžine časa, ki je bil porabljen za zahtevo, odgovor in prenos, določalo tudi nivo obremenitve strežnika.

- Količina prenesenih podatkov pri vsakem zahtevku: ker smo pri zahtevkih tipa POST lahko spreminjali količino naključno generirane vsebine,

ki se je zapisala na Wiki stran, smo tako lahko določali količino prenešenih podatkov. To je posledično vplivalo na količino prenešenih podatkov pri sledečih GET zahtevkih po zapisanih straneh. Pri posameznem testu smo obravnavali strani enake dolžine, tako da je bila količina prenesenih podatkov sorazmerna s količino zahtevkov. Količina prenesenih podatkov od aplikacijskega strežnika je sorazmerna s količino nepredpomnjenih zahtevkov.

- Pot zahtevkov oz. končna točka za zahtevke (angl. *endpoint*): glede na določen naslov končne točke smo določali, po kateri poti potujejo zahtevki in ali se v primeru potovanja preko strežnika CloudFlare za njih izvaja predpomnjenje. Ta naslov je sicer iz stališča uporabnika deloval enako v vseh primerih, vplival je le na čas za izvedbo zahtevkov.

5.4 Izvedba meritev

5.4.1 Veliko število zahtev za branje po majhnem številu različnih strani

Pri tem testu gre za izvedbo 1000 zahtev za branje (GET) po 100 različnih straneh. Testni scenarij je bil izveden s predpomnjenjem in brez predpomnjenja. Za namene testiranja smo vzpostavili 3 različne dostopne točke, ki kažejo na isti strežnik, njihova pot pa je različna:

- navadna: zahtevki gredo preko CloudFlare, kjer se izvaja predpomnjenje,
- nocache: zahtevki gredo preko CloudFlare, kjer pa se predpomnjenje preskoči,
- source: zahtevki ne gredo preko CloudFlare, temveč gredo neposredno do strežnika.

Hipoteza

Naša hipoteza je, da bo zaradi ponavljajočih se zahtev na veliko zahtev odgovoril predpomnilniški strežnik in s tem razbremenil aplikacijski strežnik. Strani so dolge 5000 znakov, tako da so zadosti velike, da je predpomnjenje sploh smiselno in da se pozna njegov učinek.

Meritve

Vsaka meritev predstavlja 1000 zahtev. Meritve so bile izvedene v petek zvečer, 8.4.2016 in ponovljene v soboto popoldne, 9.4.2016. V tabeli 5.2 so rezultati petkovih meritev. Število zahtevkov do aplikacijskega strežnika ni odvisno od razmer v omrežju, saj tako lokalni predpomnilnik kot CloudFlare ob drugi zahtevi za isto stran vrne predpomnjeno vsebino. Povprečen porabljen čas se je razlikoval za nekaj milisekund (največ 5 ms), vendar so razmerja, s katerimi

dokazujemo smiselnost uporabe spletnega predpomnilniškega strežnika, ostala podobna.

Številka meritve	1	2 - 7	8	povprečje
Brez predpomnjenja	1000	1000	1000	1000
S predpomnjenjem	100	100	100	100

Tabela 5.1: Število zahtevkov do aplikacijskega strežnika.

Številka meritve	1	2	3	4	5	6	7	8	povprečje
Brez predpomnjenja	76	77	76	74	75	78	75	74	75,6
Lokalno predpomnjenje	14	15	14	14	15	14	15	14	14,4
CloudFlare	30	37	37	39	31	32	31	32	33,6

Tabela 5.2: Povprečen porabljen čas na odgovor 1 zahtevka (v milisekundah). Pri meritvah brez uporabe predpomnjenja so zahtevki še vedno potovali preko strežnika CloudFlare, kjer pa se niso predpomnili (dostopna točka *nocache*)

Komentarji

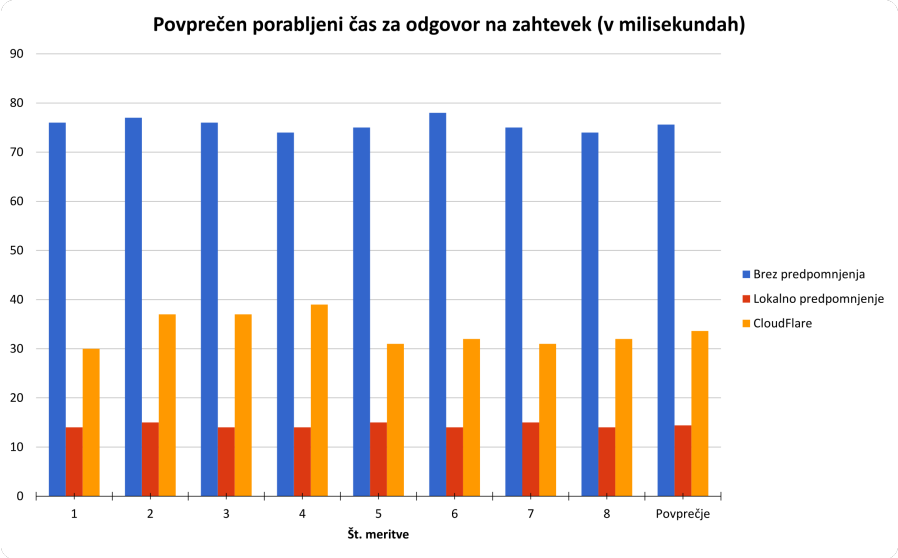
Kot je prikazano v tabeli 5.1, je brez predpomnjenja število zahtevkov do aplikacijskega strežnika, jasno, 1000, ob uporabi predpomnilnika pa se to število zmanjša na 100, saj pri vsakem zahtevku zahtevamo eno od 100 strani, ob ponovnem zahtevku strani pa le-to servisira predpomnilniški strežnik. Pri tem scenariju je uporaba predpomnilniškega strežnika zelo učinkovita. Aplikacijski strežnik servira le 10% zahtevanih strani.

Na tabeli 5.2 je za vsako meritev prikazan povprečni čas odgovora na zahtevek. Lokalno predpomnjenje je realizirano tako, da je na odjemalcu naložen strežnik nginx, ki je nastavljen kot predpomnilniški strežnik [35]. Povprečen čas odgovora je v primeru lokalnega predpomnjenja najkrajši, saj na večino (90%) zahtevkov odgovori program, ki je na samem odjemalcu. Ugotovili smo, da CloudFlare predpomni na enak način kot lokalni odjemalec, le da je čas odgovora primerno večji, ker se predpomnilniški strežnik v našem primeru nahaja na Dunaju.

Opisan scenarij odraža precej realno uporabo Wiki strani. Iz danih rezultatov je razvidno, da je v našem primeru uporaba spletnega predpomnilniškega strežnika zelo smiselna.

5.4.2 Majhno število zahtev za branje po velikem številu različnih strani

Pri tem testu gre za izvedbo 100 zahtev za branje (GET) po 1000 različnih straneh. Strani so velike 5 kB, beležili pa smo čas odzivanja na zahtevek. Predpomnilnik smo med poskusi pobrisali, za dostop pa smo uporabljali dve vrsti



Slika 5.3: Graf rezultatov prvega scenarija.

dostopov: s predpomnjenjem in brez predpomnjenja. Obe povezavi sta šli preko CloudFlare strežnikov. Test smo izvedli osemkrat. Test smo izvedli v noči iz 16. na 17. april. Izvedli smo ga osemkrat. Začeli smo ob 0:05, vsak test je trajal okoli 3 minute, med ponovitvami testa pa je pretekla točno ena ura.

Hipoteza

Ker izvajamo majhno število zahtev na velikem številu strani, predvidevamo, da ne bo kakšne razlike med tem, če uporabljamo predpomnjenje ali ne. Prav tako predvidevamo, da predpomnjenja v tem primeru skorajda sploh ne bo, saj se zahteva shrani v predpomnilnik šele po najmanj enem dostopu, verjetnost za to pa je sorazmerno majhna: v teoriji bi pričakovali tam okoli 5 dostopov do strani, do katerih smo tekom poskusa že dostopali.

Rezultati

Številka meritve	1	2	3	4	5	6	7	8	Povprečje
Brez predpomnjenja	130	128	121	130	144	133	122	117	128,12
S predpomnjenjem	121	133	133	133	127	122	129	148	130,75
Št. predpomnjenih zah.	2	1	2	3	3	2	5	4	2,75

Tabela 5.3: Tabela povprečnih časov odgovora ter število predpomnjenih zahtevkov (v milisekundah).

Komentarji

Kot je razvidno iz tabele 5.3, so dostopni časi približno enaki, kakšnih večjih razlik pa ni. V povprečju je čas dostopa brez predpomnjenja malo manjši kot čas dostopa preko povezave s predpomnjenjem. To je rahlo presenetljivo, vendar pa je razlika zadosti majhna, da bi jo lahko razložili kot šum v podatkih.

Prav tako vidimo, da je bilo predpomnjenih dostopov nekaj, vendar pa ne toliko, kot smo pričakovali. Iz tega lahko sklepamo, da Cloudflare pri majhni količini prometa čez večje število strani ne pomni vsakega zahtevka (seveda pa je možno tudi, da smo se ušteli pri teoretičnem izračunu).

5.4.3 Test z večjo obremenitvijo do odpovedi

Pri tem testu smo iz določenih strani (odjemalcev) poskušali strežnik čim bolj obremeniti, medtem pa smo poskušali s strani tretjega odjemalca izvajati zahtevke in beležiti, kako dobro mu je strežnik še sposoben servisirati pod veliko obremenitvijo.

Primerjali smo povprečne rezultate sto zahtevkov pri različni obremenjenosti strežnika in pri dveh načinih dostopa do strežnika - najprej po neposredni povezavi, ki zaobide CloudFlare in s tem predpomnjenje, nato pa še preko CloudFlare.

Strežnik smo obremenili tako, da smo napisali in izvajali posebno skripto, ki je hkrati (tj. vzporedno v različnih podprocesih) generirala določeno število GET zahtevkov in to ponavljala za čas testiranja. Zahtevki so bili izvedeni vzporedno zato, ker bi se sicer v skripti izvajali zaporedno in bi jih strežnik izvajal sinhrono, enega za drugim, ker pa procesiranje zahtevka na strežniku traja zgolj določen čas celotnega časa izvedbe, tako ne bi imeli zagotovitve, da je strežnik obremenjen celotni čas oziroma da ima v vrsti še čakajoče, neobdelane zahtevke. Skripta je poleg tega beležila še število izvedenih zahtevkov na sekundo oziroma minuto. Odgovori so sicer, kljub temu, da smo jih smatrali kot predolge, še vedno prispeli.

Hipoteza

Hipoteza pri tem testu je, da bo strežnik do določene obremenitve iz ene strani z vidika uporabnika deloval kot nemoten, od tam naprej pa se bo obremenjenost že poznala s strani uporabnika v obliki daljšega (in eventuelno predolgega [31]) časa za izvedene zahtevke. Prav tako je za pričakovati, da bo v primeru preobremenjenosti strežnika uporabnikovo čakanje na zahtevke hitro poskočilo.

Z uporabo predpomnjenja je pričakovati, da bo strežnik precej bolj odporen na obremenitev in bo zato vpliv oremenitve na uporabnika manjši, saj se bo razmeroma majhen del zahtevkov moral dejansko izvajati na strežniku, večino pa bo stregel CloudFlare.

Meritve

Pri meritvah smo pri različnem številu hkratno izvedenih zahtevkov poleg povprečnega porabljenega časa za zahtevek uporabnika beležili še število izvedenih zahtevkov na minuto s strani odjemalca.

Meritve so bile izvedene v petek, 15. aprila, med 17. in 20. uro. Skupno 2012 zahtevkov je za izvršitev potrebovalo 3304 sekund. Velikosti strani so znašale med 100 in 10000 znakov naključno generirane vsebine.

Hkratnih	0	5	10	15	25	40	50	60	70	80	100	150	200	250
Čas v ms	299	415	623	639	838	1052	1385	1935	2136	2185	2253	2324	3193	3836

Tabela 5.4: Povprečen porabljen čas na odgovor 1 zahtevka (v milisekundah) brez uporabe predpomnjenja.

Hkratnih	0	5	10	15	25	40	50	60	70	80	100	150	200	250
Čas v ms	323	320	341	352	362	417	434	439	437	477	547	519	578	565

Tabela 5.5: Povprečen porabljen čas na odgovor 1 zahtevka (v milisekundah) z uporabo predpomnjenja.

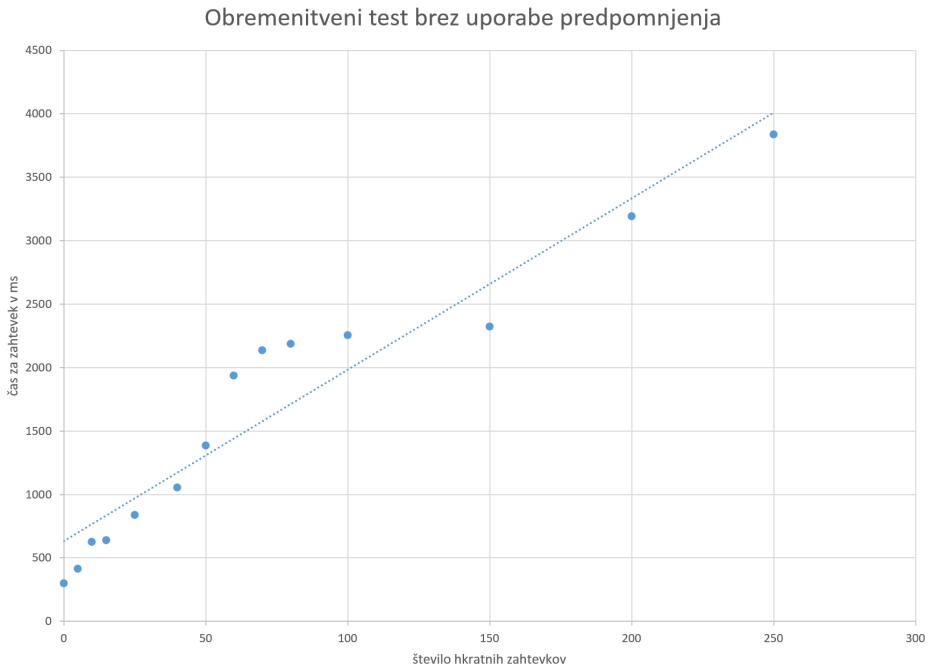
Komentarji

Za prvi preizkus hipoteze smo izvedli poskusen test, kjer smo strežnik preobremenili s 1000 hkratnimi zahtevki (izvedeni so bili paralelno), kar je simuliralo manjši DOS napad (angl. *denial of service*) in kot pričakovano je strežnik zares potreboval precej dlje, da se je odzval na zahtevek - namesto nekaj sto milisekund se je čas povečal na več kot 4 sekunde, kar je za uporabnika že moteče.

Za glavni test smo izvedli scenarij, kjer smo spreminjali število vzporedno izvedenih zahtevkov, začenši z 0 (uporabnik je bil edini, ki je takrat dostopal do strežnika) in vse do 250. Test smo izvedli s predpomnjenjem in brez njega. Da bi simulirali situacijo Wiki strani, smo test omejili na 1000 različnih vnosov - šlo je torej za 1000 različnih zahtevkov GET. Po vsakem testu smo izpraznili predpomnilnik, tako da so posamezni testi ostali neodvisni od prejšnjih.

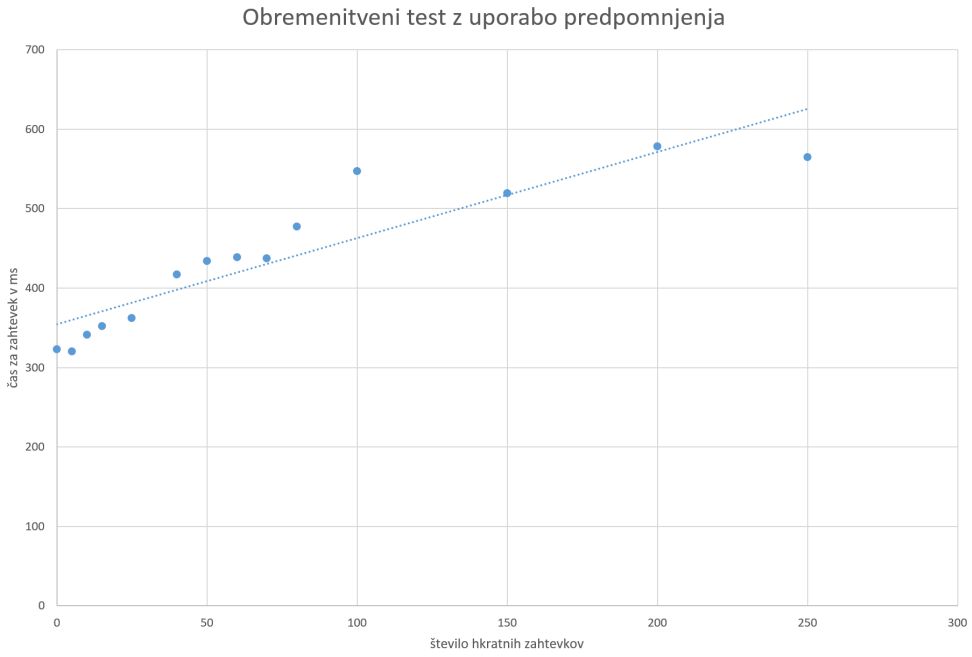
V tabelah 5.4 in 5.5 je razviden vpliv števila hkratno izvedenih zahtevkov iz tretje strani na povprečni čas enega zahtevka na uporabnikovi strani. Vidno je, kako v primeru neuporabe predpomnjenja ta čas hitro zraste na za uporabnika neugodne vrednosti, medtem pa z uporabo predpomnjenja te vrednosti ostanejo sprejemljive tudi pri vseh testiranih vrednostih hkratno izvedenih zahtevkov. Pri večjih vrednostih je za pričakovati, da bo velik del zahtevkov že v predpomnilniku, zato čas za uporabnika ne bo nadalje precej rastel, kar je iz rezultatov razvidno.

Ta test je služil tudi kot test do odpovedi (angl. *test to fail*), saj vrednosti nad nekaj sekund lahko smatramo kot odpoved sistema s perspektive ugodne uporabniške izkušnje. S predolgim čakanjem na vsebino namreč izgubljamo stranke in s tem profit, ki nam omogoča nadaljnje delovanje [31].

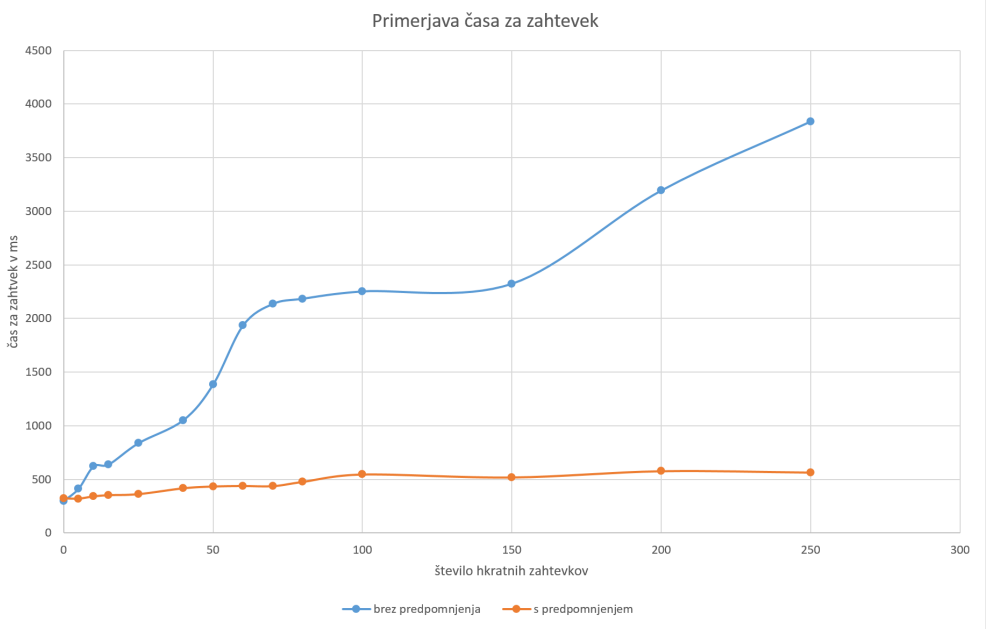


Slika 5.4: Graf rezultatov scenarija brez uporabe predpomnjenja.

Opazno je torej, da ima vključitev predpomnjenja velik pozitivni učinek pri takem scenariju kljub temu, da povezava ni neposredna (v tem primeru je neposredna povezava krajša). Za boljšo predstavo naraščanja časa za zahtevek in primerjave obeh scenarijev so vključeni grafi na slikah 5.4, 5.5 in 5.6.



Slika 5.5: Graf rezultatov scenarija z uporabo predpomnjenja.



Slika 5.6: Graf primerjave rezultatov obeh scenarijev.

5.4.4 Učinkovitost predpomnjenja pri pisanju

Pri tem testu smo merili učinkovitost predpomnjenja pri mešanih zahtevah. Izvedli smo več poskusov z različno velikostjo strani. Strani so bile velike 5, 10, 20 ali 50 kB. Vsak zahtevek je imel naključno možnost (5%), da bo spremenil stran — pri taki priložnosti smo na strežnik najprej naslovili zahtevek GET, zatem pa še zahtevek POST. Izvedli smo 10000 zahtevkov po 100 straneh. Posamezna stran po vpisu v predpomnilnik tam ostane tri minute.

Hipoteza

Hipoteza pri tem testu je, da bo zaradi naključnih POST zahtevkov predpomnilnik manj učinkovit v primerjavi s primerom, ko smo uporabljali le GET zahteve. Ker je največja starost predpomnilnika manj, kot pričakujemo, da bo test tekel, pričakujemo tudi, da bo pri večjih zahtevah predpomnilnik manj učinkovito uporabljen.

Meritve

Najprej smo zagnali meritve z mešanim prometom, zatem pa smo za primerjavo izvedli še meritve brez POST zahtevkov. Predpomnilnik smo izpraznili pred vsakim izvajanjem testa, zahteve pa so v predpomnilniku ostale le tri minute. Teste smo izvajali zaporedno, po vrsti od najmanjše velikosti strani do največje. Test smo izvedli osemkrat: za vsako velikost strani po dvakrat — enkrat z mešanimi zahtevami, enkrat pa samo z zahtevki tipa GET. Vseh osem poskusov smo izvedli enkrat brez premorov v noči iz 1. na 2. maj (nedelja, ponedeljek) med 22. in 2. uro zjutraj. Rezultati meritev so vidni v tabeli 5.6.

Velikost zahtevka		5	10	20	50
Mešani zahtevki	Št. POST zah.	471	498	504	528
	Št. GET zah. na strežniku	1044	1017	1183	1600
	Delež predpom. zahtev	89,56%	89,83%	88,17%	84,00%
Samo GET	Št. zah. na strežniku	412	447	600	791
	Delež predpom. zahtev	95,88%	95,53%	94,00%	92.09%

Tabela 5.6: Delež predpomnjenih zahtev pri mešanih zahtevah različnih velikosti. Za primerjavo smo izvedli tudi poskus, v katerem smo imeli le zahteve tipa GET.

Komentarji

Vidimo, da se ob 5% deležu POST zahtev število zahtevkov na strežniku približno podvoji. To pomeni, da je učinkovitost predpomnilnika v takih primerih manj učinkovita v primerjavi s primerom, da na strežnik pošiljamo samo GET zahteve — tako, kot smo pričakovali.

Zanimivo je, da učinkovitost predpomnilnika ne pada z faktorjem povečanja velikosti, čeprav se je trajanje testov podaljševalo približno s faktorjem velikosti.

Natančnih meritev za trajanje nismo opravili, vendar pa smo trajanje približno ocenili na podlagi opazovanja ter zahtevkov, ki so bili v določenem časovnem obdobju izmerjeni na CloudFlare. Čeprav se je test z zahtevki velikosti 50kB izvajal okoli ene ure, je bilo zahtevkov na strežniku le 791, pričakovali pa bi jih tam okoli 2000 (število strani * (čas izvajanja / dolžina veljavnosti strani v predpomnilniku)). Medtem pa je pri testu z najmanjšo velikostjo strani nepredpomnjenih zahtevkov vsaj toliko (ali pa celo več), kot bi jih glede na dane pogoje pričakovali. Iz tega lahko sklepamo dve stvari: prva je, da Cloudflare raje predpomni večje strani, druga pa, da nastavitev za največjo starost strani v predpomnilniku pri tako majhnih vrednostih ne deluje najbolj zanesljivo. To pomeni, da se lahko (če je največja dovoljena starost majhna) stran v predpomnilniku nahaja dlje časa, kot smo določili.

A če od deleža nepomnjenih zahtev pri poskusu z mešanimi zahtevami odštejemo delež nepomnjenih zahtev pri poskusu samo z zahtevki tipa GET (torej, odstranimo vpliv poteka predpomnilnika, tako da nam ostane samo vpliv invalidacije predpomnilnika zaradi POST zahtev) ugotovimo, da je pri večjih straneh vpliv invalidacije na predpomnjenje večji kot pri manjših straneh. Na prvi pogled se torej zdi, da predpostavka o tem, da Cloudflare raje pomni večje strani, ne drži. To neskladje se lahko v meri razloži z nesimetrično internetno povezavo: ker je bil prenos proti strežniku med tem poskusom okoli 6-krat manjši kot prenos v obratni smeri, zahtevki pa niso bili vzporedni, so lahko imeli POST zahtevki večji vpliv na potek veljavnosti strani v predpomnilniku, kot zahtevki tipa GET. Ta vpliv pa je bil spet večji pri straneh z večjo velikostjo.

5.4.5 Primerjava zahtevnosti pisanja in branja brez predpomnilnika

Pri tem testu smo primerjali povprečen čas odgovora na zahtevek pri 200 urejanjih (POST) po 100 straneh s povprečnim časom odgovora na zahtevek pri 200 branjih (GET) po 30 straneh brez vključenega predpomnilnika. Zahtevki tipa POST so na strani zapisali naključno generirano vsebino, dolgo 1000 znakov.

Hipoteza

Naša hipoteza pri tem testu je, da so pisanja (POST zahteve) malo bolj zahtevna kot branja in je zato povprečen čas odgovora pri pisanju malo daljši.

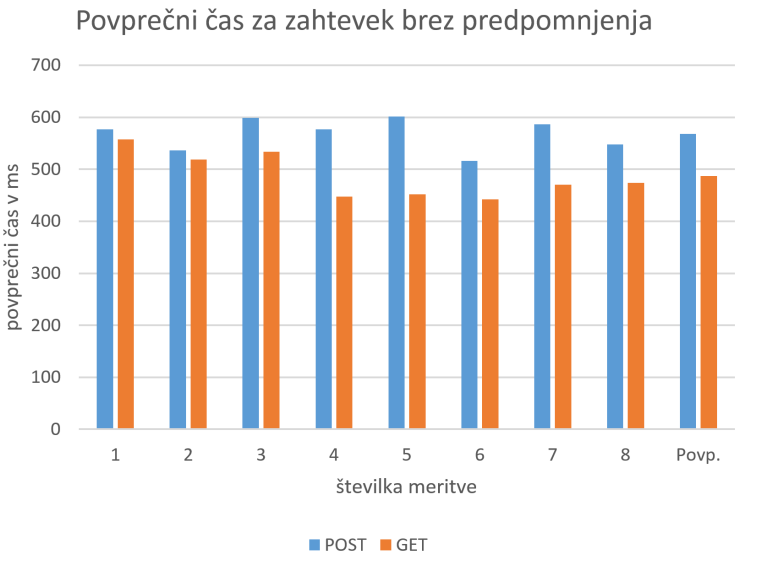
Merive

Pri merjenju povprečnega časa smo izvedli osem zaporednih meritev za oba tipa zahtevkov in beležili povprečje. Zahtevki v tem primeru niso šli preko strežnika CloudFlare in so tako prepotovali najkrajšo možno pot.

Meritve so bile izvedene v sredo, 27. 5. med 13. in 18. uro. Skupno je bilo poslanih 1600 zahtevkov, skupno trajanje vseh zahtevkov pa je znašalo 1687 sekund. Predstavljene so v tabeli 5.7 in v grafični obliki na sliki 5.7.

Številka meritve	1	2	3	4	5	6	7	8	Povprečje
POST	577	536	599	577	601	516	586	548	567.5
GET	557	519	534	447	452	442	470	474	486.875

Tabela 5.7: Tabela povprečnih časov odgovora glede na tip zahtevka.



Slika 5.7: Graf scenarija brez uporabe predpomnilnika.

Komentarji

Iz rezultatov je razvidno, da je povprečni čas za oba tipa zahtevkov podoben, a konsistentno večji za zahtevke tipa POST, kar je skladno z našo hipotezo. Zahtevki tipa POST so v tem primeru za izvršitev potrebovali okoli 16% več časa.

5.4.6 Primerjava zahtevnosti pisanja in branja s predpomnilnikom

Pri tem testu smo primerjali povprečen čas odgovora na zahtevek pri 200 urejanjih (POST) po 30 straneh s povprečnim časom odgovora na zahtevek pri 200 branjih (GET) po 30 straneh z vključenim predpomnilnikom. Zahtevki tipa POST so na strani zapisali naključno generirano vsebino, dolgo 1000 znakov.

Hipoteza

Naša hipoteza pri tem testu je, da pisanja zahtevajo veliko dlje časa kot branja, saj pri 200 zahtevkih za branje po 30 straneh na večino zahtevkov (90%)

odgovori predpomnilniški strežnik, tako kot pri našem prvem testu.

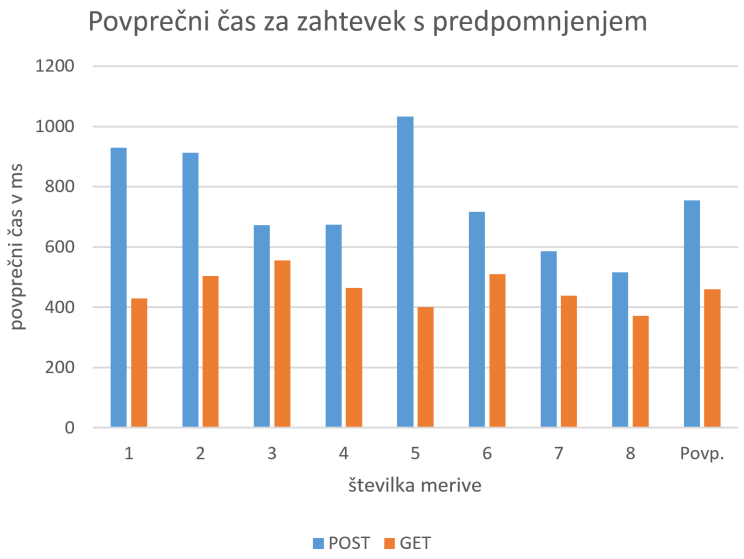
Meritve

Tudi pri tem scenariju smo zagnali zaporedje osmih meritev in beležili povprečje. Ker so zahtevki potovali preko strežnika CloudFlare, ki je izvajal predpomnjenje, so prepotovali daljšo pot. Po vsakem izmed osmih testov smo izpraznili predpomnilnik in tako zagotovili njihovo medsebojno neodvisnost.

Meritve so bile izvedene v sredo, 27. 4. med 13. in 18. uro. Skupno je bilo poslanih 1600 zahtevkov, skupno trajanje vseh zahtevkov pa je znašalo 1942 sekund.

Številka meritve	1	2	3	4	5	6	7	8	Povprečje
POST	930	912	672	674	1032	717	586	516	754.875
GET	429	504	556	464	401	510	438	372	459.25

Tabela 5.8: Tabela povprečnih časov odgovora glede na tip zahtevka.



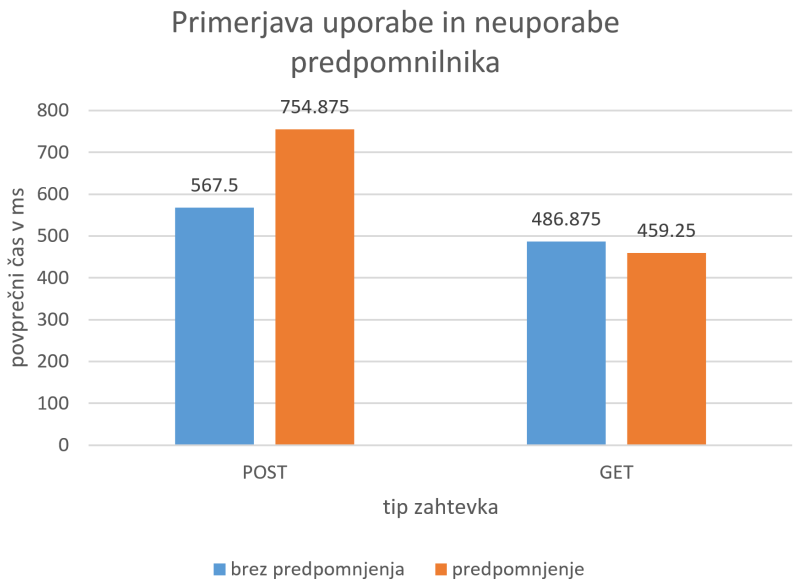
Slika 5.8: Graf scenarija z uporabo predpomnilnika.

Komentarji

Za razliko od scenarija brez vključenega predpomnilnika je tu moč opaziti nekajkrat večje razlike - zahtevki tipa POST so za izvršitev potrebovali okoli 64% dlje. Ker so zahtevki prepotovali daljšo pot, je bilo za pričakovati, da bodo tisti, ki bodo to pot prepotovali v celoti (in ne bodo servirani že s strani strežnika

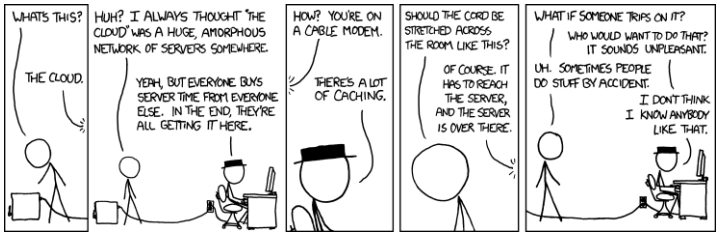
	brez	predp.
POST	567.5	754.875
GET	486.875	459.25

Tabela 5.9: Tabela povprečnih časov odgovora za scenarij brez uporabe predpomnilnika v primerjavi s scenarijem, kjer se ta uporablja.



Slika 5.9: Graf primerjave obeh scenarijev.

CloudFlare), trajali dlje časa kot zahtevki v scenariju brez uporabe predpomnilnika (in zato direktno, krajšo potjo). V tabeli 5.9 je razvidno, da to res velja za zahtevke tipa POST, medtem ko so zahtevki tipa GET kljub daljši prepotovani poti hitrejši.



Slika 5.10: Vir: https://xkcd.com/908/

5.5 Zaključek

V našem delu smo ocenjevali učinkovitost predpomnilniškega strežnika v primeru Wiki aplikacije. Pričakovali smo, da bo predpomnilniški strežnik razbremenil aplikacijski strežnik in zmanjšal čas čakanja na zahteve odjemalca.

Testne scenarije smo pripravili tako, da so v skladu z realno uporabo take strani. Naše hipoteze smo potrdili z meritvami. Izkaže se, da je predpomnilniški strežnik občutno pohitril čas strežbe Wiki strani in s tem omogočil boljšo uporabniško izkušnjo. Pohitritev je bila še posebej očitna v primeru velikega števila hkratnih zahtevkov, ko je bil v primeru neposrednega dostopa (tj. brez predpomnjenja) aplikacijski strežnik preveč obremenjen, da bi v primernem času odgovoril na posamezni zahtevek. Z obremenitvenim testom smo dosegli, da je bil brez predpomnjenja čas čakanja na odgovor okoli 4 sekunde, kar smo obravnavali kot odpoved storitve [31], medtem ko je bil z uporabo predpomilniškega strežnika v enaki situaciji čas čakanja približno pol sekunde. Predpomnilniški strežnik je bil učinkovit samo v zahtevkih za branje, saj je pri zahtevkih za pisanje v vsakem primeru bilo potrebno podatke poslati aplikacijskemu strežniku, da jih je le-ta shranil v bazo. V primeru Wiki strani je v primerjavi z zahtevki za branje zahtevkov za pisanje sorazmerno malo, zato je bila uporaba predpomnilniškega strežnika zelo smiselna. Iz tega lahko sklepamo, da je učinkovitost predpomnilniškega strežnika odvisna od narave aplikacije.

Poglavje 6

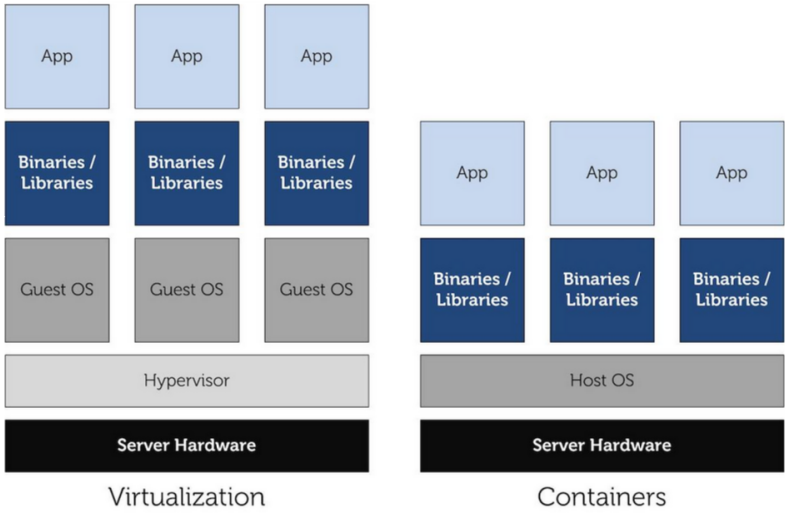
Zmogljivostna analiza virtualnih implementacij

Nejc Nadižar, Eva Križman, Klemen Klanjšček

6.1 Predstavitev ideje

V okviru seminarske naloge pri predmetu Zanesljivost in zmogljivost računalniških sistemov, smo se v skupini odločili narediti primerjavo med dvema realizacijama strežnika za virtualno okolje.
Prva implementacija (levo na sliki 6.1) je realizirana na najbolj pogost način, kar pomeni da na strojni opremi teče namenski operacijski sistem ("hypervisor"), nad katerim se potem kreirajo virtualne delovne postaje s svojim nespremenjenim operacijskim sistemom, do katerih lahko dostopamo preko oddaljenega dostopa [37].
Druga implementacija (desno na sliki 6.1) je realizirana s programskimi zabojniki ("software containers"), kjer direktno na strojni opremi teče nespremenjen operacijski sistem, znotraj katerega so zabojniki realizirani na nivoju posameznih aplikacij. Iz zabojnika navzven se vidi samo sliko sistema, torej znotraj zabojnika ne moremo direktno vplivati na sam sistem [36].

Za testiranje obeh implementacij smo uporabili že obstoječa orodja za testiranje zmogljivosti sistema. Omejili smo se na testiranje zmogljivosti strojne opreme, jedra operacijskega sistema, diskovja ter omrežnih storitev.



Slika 6.1: Strukturi sistemov [46].

6.2 Predstavitev rešitve

Obstaja več različnih vrst virtualizacije, tako na nivoju operacijskega sistema, kot na nivoju strojne opreme [38].
Na nivoju strojne opreme imamo, "Type 1 hypervisor" ("VMWare ESX"), kjer direktno nad strojno opremo teče namenski operacijski sistem. "Type 2 hypervisor" ("QEMU") se razlikuje od prejšnjega, da med strojno opremo in namenskim operacijskim sistemom teče še nespremenjen operacijski sistem. Nazadnje pa imamo še paravirtualizacijo ("XENServer"), kjer gredo določeni sistemski klici iz virtualnega okolja mimo sistema direktno do strojne opreme, kar zahteva spremenjen operacijski sistem na katerem teče virtualno okolje.

Prav tako imamo na vritualizaciji s programskimi zabojniki več možnih implementacij, ki se med sabo razlikujejo po količini virtualizacije (koliko strojne opreme znajo virtualizirati). "chroot" kot primer zabojnika, ki virtualizira zgolj uporabniško okolje, ter na primer "OpenVZ", ki virtualizira večino sistema.

Za "VMWare ESX" in "OpenVZ" smo se odločili, ker sta si podobna v količini opreme, ki jo virtualizirata in sta oba začela z razvojem približno ob istem času.

6.2.1 Strojne specifikacije testnega sistema

Testni sistem predstavlja zmogljivejša delovna postaja ("workstation") s sledečimi specifikacijami

- Procesor: Intel i7-2600 (3.40GHz),
- Pomnilnik: 16 GB,
- Diskovje: 250 GB SDD,
- Mreža: 100MB/1GB.

6.2.2 Programske specifikacije testnih sistemov

Sistem, ki uporablja najpogostejšo obliko virtualizacije, uporablja kot osnovni operacijski sistem "VMWare ESXi 5.5.0 Hypervisor" [40], z brezplačno licenco in verzijo jedra (kernel) "VMKernel release 2403361". Virtualne delovne postaje, ki tečejo na prej omenjenem sistemu, uporabljajo operacijski sistem "CentOS 6.7 (Final)", z verzijo jedra "Linux 2.6.32-042stab113.21 x86-64" [42].

Sistem z virtualizacijo na nivoju operacijskega sistema uporablja kot osnovni operacijski sistem "CentOS 6.7 (Final)", z verzijo jedra "Linux 2.6.32-042stab113.21 x86-64". Aplikacije tečejo v svojem virtualnem okolju ("software containers") s sistemom "OpenVZ 2.6.32-042stab113.21 x86-64" [39] s predlogo sistema "CentOS 6.7 template x86-64" [42].

Lokacija strežnika in dislocirani testi za našo primerjavo nimajo vpliva, saj primerjamo zmogljivost dejanske implementacije in je v primeru vpliva infrastrukture preveč spremenljivk, ki niso relevantne. Vsi testi se tako izvajajo v lokalnem omrežju.

Pri ocenjevanju zmogljivosti obeh sistemov pri različnih obremenitvah smo se pri testih odločili za testiranje "enouporabniške" (en programski zabojnik v enem primeru in ena virtualna delovna postaja, ki uporablja vse sistemske vire v drugem) in "večuporabniške" implementacije (več uporabnikov na eni virtualni delovni postaji ali na več virtualnih delovnih postajah ter več uporabnikov na en zabojnik in več uporabnikov na več zabojnikih).

Pri testiranju "enouporabniške" implementacije, smo se osredotočili na sledeče teste:

- Test I/O operacij: Testirali smo zmogljivost pri ustvarjanju, kopiranju datotek (različnih velikosti, različnega števila datotek).
- Test mreže: Testiramo učinkovitost in zmogljivost operacij, ki uporabljajo mrežno opremo (dostop do mrežne opreme, diskovnih enot, interneta, drugih sistemov,...).

- Test CPU: Testiramo zmogljivost samega procesorja, v eni in drugi implementaciji (veliko benchmarkov je že implementiranih).
- Test preobremenitve: Testiramo odzivnost sistema, ko zahtevnost bremena presega same specifikacije sistema (enormna količina operacij, preobremenitev sistema z že obstoječimi programi).

Testiranje "večuporabniške" implementacije poteka na naslednje načine:

- Test odzivnosti: testirali smo čas izvedbe ukaza(programa) in čas do pridobitev rezultata (pri različnih obremenitvah sistema; več uporabnikov naenkrat, obremenitev gostitelja,...);
- Test komunikacije: testiramo vpliv komunikacije med uporabniki na delovanje in zmogljivost sistema (prenos podatkov med uporabniki na sistemu); simuliralo se je komunikacijo md več uporabniki, s uporabo opcije paralelnih testov pri izbranih orodjih za testiranje. Komunikacija je potekala znotraj samega virtualnega okolja z več delovnimi postajami (ali programskimi zabojniki), ter tudi preko klienta v lokalnem omrežju.

Nekatera obstoječa orodja za testiranje že zajemajo več kot en tip testov in so na kratko opisani kasneje. Vse teste smo opravili z obstoječimi orodji ter postopek avtomatizirali s preprostimi BASH ali C skriptami.

6.3 Definicije orodij, bremen in metrik

Naša izbira orodij za testiranje vsebuje:

- **UnixBench** [43]: orodje za testiranje operacijskega sistema (eno/več opravil, paralelno izvajanje); vsebuje teste za delo z nizi ("Dhrystone"), delo v plavajoči vejici ("Whetstone"), število sistemskih klicev ("execl throughput"), kopiranje datotek preko predpomnilnika ("FileCopy"), komunikacijo med procesi na sistemu ("pipe-based context switching"), količina "overhead"-a pri sistemskih klicih ("system call overhead");
- **Iperf3** [45]: je uporabljen kot alternativa predhodnem testu (po vsej verjetnosti uporabljen bolj uporaben za naše potrebe); meri zmogljivost in zanesljivost povezave (količina prenosa, hitrost povezave, število ponovnih pošiljanj);
- **Time**: je standardni linux program, ki meri čas izvajanja določenega ukaza (procesorski čas, uporabljen (ali izpeljanke) v večini "benchmarkov" za izračun porabljenega časa (realnega in procesorskih ciklov));

Težave so se pokazale pri testiranju diskovnih pogonov in zmogljivosti diskov, saj nam testi (hdparam, Bonnie++) niso vračali konsistentnih in pravilnih podatkov ali pa se test ni izvedel. Problem pripisujemo implementaciji diska v

virtualnem okolju, kjer particije diska ne vedo kje dejansko se fizično nahajajo (lahko so razpršene čez več sektorjev, ki pa niso nujno zaporedni).

Bremena s katerimi smo testirali naš sistem, so vključevala sekvenčne (na sistemu teče ena kopija testa naenkrat) ter paralelne teste (teče več kopij enakega testa naenkrat).

6.4 UnixBench

6.4.1 Opis testa

Unixbench orodje preverja zmogljivost operacijskega sistema. Njegovi rezultati so bolj odvisni od jedra ("kernel") operacijskega sistema in verzije orodij s katerimi so bili programi prevedeni, tako da strojna oprema nima velikega vpliva na sam test. Orodje uporablja več različnih testov, vsako z drugačno najprimernejšo enoto.
Meritve v tabeli 6.1 predstavljajo sekvenčne teste, v tabeli 6.2 pa so predstavljeni paralelni testi. Obe meritvi uporabljata naslednje enote: lps[1] prestavlja iteracijo zanke na sekundo ("loop iterations per second"), posamezen test se izvaja 10 sekund in se ponovi 7-krat; lpm[2] predstavlja iteracijo zanke na milisekundo ("loop iterations per milisecond"), posamezen test se izvaja 60 sekund in se ponovi 2-krat; lps[3] enako iteracija zanke na sekundo, razlika je v trajanju (20 sekund) in številu ponovitev (2-krat); MWIPS[4] predstavlja milijon operacij v plavajoči vejici na sekundo, posamezen test traja 10 skund in se ponovi 7-krat; KBps[5] predstavlja "KiloBytes" na sekundo, test se izvaja 30 sekund in se ponovi 2-krat.

Tabeli 6.2 in 6.3 sta samo povzetek v obliki indeksov, da se rezultati lažje primerjajo z zmogljivostjo ostalih sistemov (določen sistem se da primerjati s sistemi po svetu [47]).

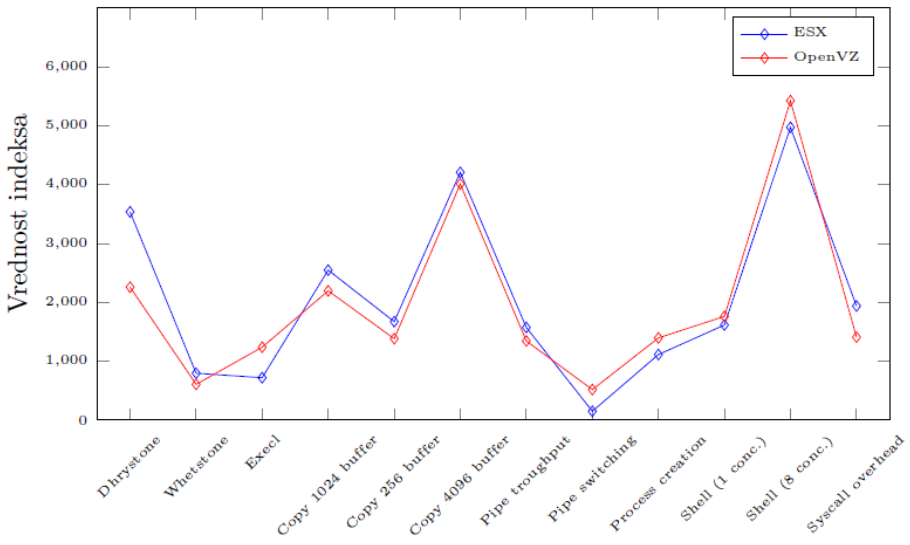
6.4.2 Hipoteza

Predvidevamo, da je klasična virtualizacija hitrejša pri kopiranju datotek, katere je možno v celoti naložiti v pomnilnik, zaradi boljše implementacije samega dostopa do njega. Virtualizacija na nivoju operacijskega sistema ima prednost pri paralelnem izvajanju, saj uporablja večnitenje strojne opreme in ne programske emulacije le te. Večkratna ponovitev testov ne bi spremenila rezultatov, saj že test sam po sebi izvede vsak test večkrat.

6.4.3 Rezultati

"Benchmark name"	ESXi-VM	OpenVZ
Dhrystone 2 using register variables (lps[1])	41298072.5	26378260.0
Double-Precision Whetstone (MWIPS[4])	4384.9	3347.8
Execl Throughput (lps[3])	3101.0	5331.2
File Copy 1024 bufsize 2000 maxblocks (KBps[5])	1009769.0	870075.2
File Copy 256 bufsize 500 maxblocks (KBps[5])	276984.6	229052.6
File Copy 4096 bufsize 8000 maxblocks (KBps[5])	2441731.3	2327838.4
Pipe Throughput (lps[1])	1959466.4	1674236.4
Pipe-based Context Switching (lps[1])	60427.0	207629.1
Process Creation (lps[3])	14048.3	17607.7
Shell Scripts (1 concurrent) (lpm[2])	6855.7	7478.7
Shell Scripts (8 concurrent) (lpm[2])	2985.0	3256.6
System Call Overhead (lps[1])	2910902.4	2115694.5

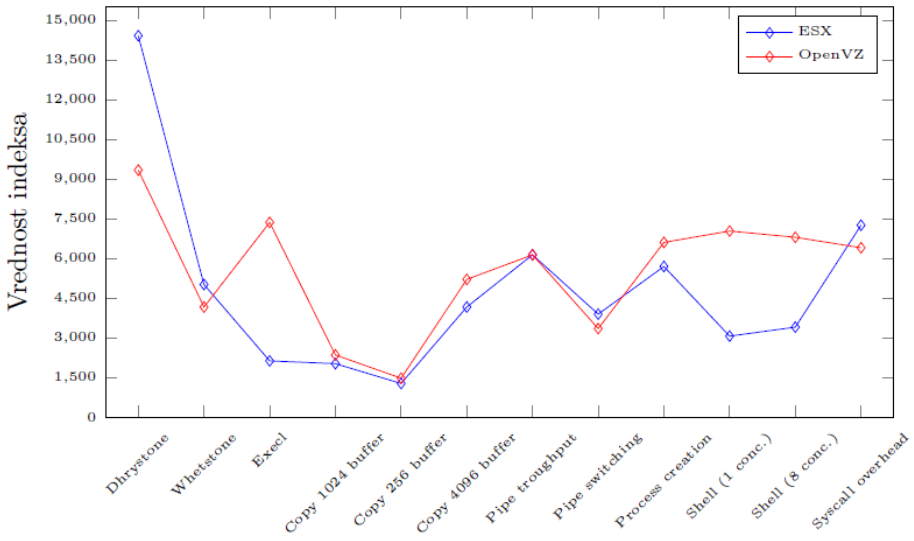
Tabela 6.1: Rezultati meritev na enem procesorju.



Slika 6.2: Primerjave indeksov rezultatov testa na enem procesorju.

"Benchmark name"	ESXi-VM	OpenVZ
Dhrystone 2 using register variables (lps[1])	168214221.6	109049113.5
Double-Precision Whetstone (MWIPS[4])	27648.6	22967.9
Execl Throughput (lps[3])	9217.1	31693.3
File Copy 1024 bufsize 2000 maxblocks (KBps[5])	807812.4	937811.1
File Copy 256 bufsize 500 maxblocks (KBps[5])	213827.6	246762.0
File Copy 4096 bufsize 8000 maxblocks (KBps[5])	2422494.4	3027486.1
Pipe Throughput (lps[1])	7647529.3	7646409.9
Pipe-based Context Switching (lps[1])	1563153.9	1345339.3
Process Creation (lps[3])	71908.1	83359.0
Shell Scripts (1 concurrent) (lpm[2])	13050.8	29866.7
Shell Scripts (8 concurrent) (lpm[2])	2050.4	4085.3
System Call Overhead (lps[1])	10895793.0	9620146.3

Tabela 6.2: Rezultati meritev na več procesorjih.



Slika 6.3: Primerjave indeksov rezultatov testa na več procesorjih.

6.4.4 Komentar

Na podlagi rezultatov lahko potrdimo našo predhodno hipotezo, ker v primeru kopiranja datotek preko pomnilnika, je virtualni sistem hitrejši (testi "File Copy bufsize"). Medtem, ko pa virtualizacija operacijskega sistema bolje izkorišča večnitnost procesorja in so zaradi tega boljši rezultati pri nekaterih ostalih testih. Sekvenčni (v tabeli 6.1) in paralelni (v tabeli 6.2) testi imajo enak zaključek,

se pa pri paralelnih testih vidi večja razlika med sistemoma, kar kaže na boljšo implementacijo v primeru virtualizacije na nivoju operacijskega sistema.
Podobnost rezultatov obeh implementacij pri testih bi lahko pripisali temu, da oba sistema uporabljata enako jedro operacijskega sistema in tečeta na enaki strojni opremi.

6.5 UnixBench - primerjava med sistemoma

6.5.1 Opis testa

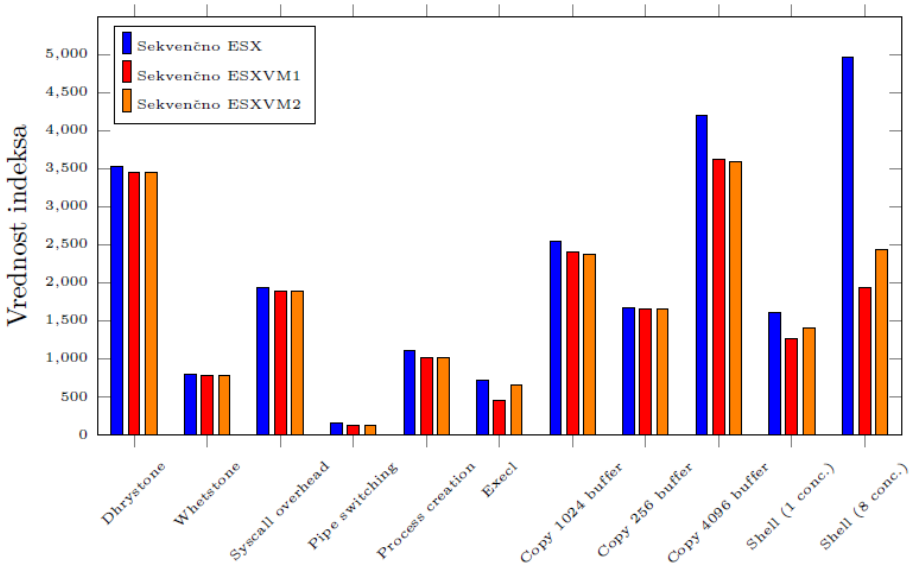
V naslednjem testu z orodjem **UnixBench** naredimo primerjavo med sekvenčno in paralelno zmogljivostjo obeh sistemov. V obeh primerih testov dve virtualni delovni postaji ali dva programska zabojnika uporabljata vsaka polovico vseh sistemih sredstev. Slika 6.4 predstavlja sekvenčne (vsaka delovna postaja poganja samo eno kopijo testov) teste, medtem ko slika 6.5 prestavlja paralelne (vsaka delovna postja pognaja osem kopij testov naenkrat) teste.
Na sliki 6.6 imamo sekvenčne teste "OpenVZ" in na 6.7 paralelne (razlaga je enaka kot pri "ESX", samo namesto delovne postaje imamo programski zabojnik).
Da smo dosegli istočasnost izvajanja testa, smo orodje avtomatizirali z BASH skripto, ki najprej sinhronizira sistemski uri, požene prvi test in nato s 20 minutnim zamikom požene naslednjega.

6.5.2 Hipoteza

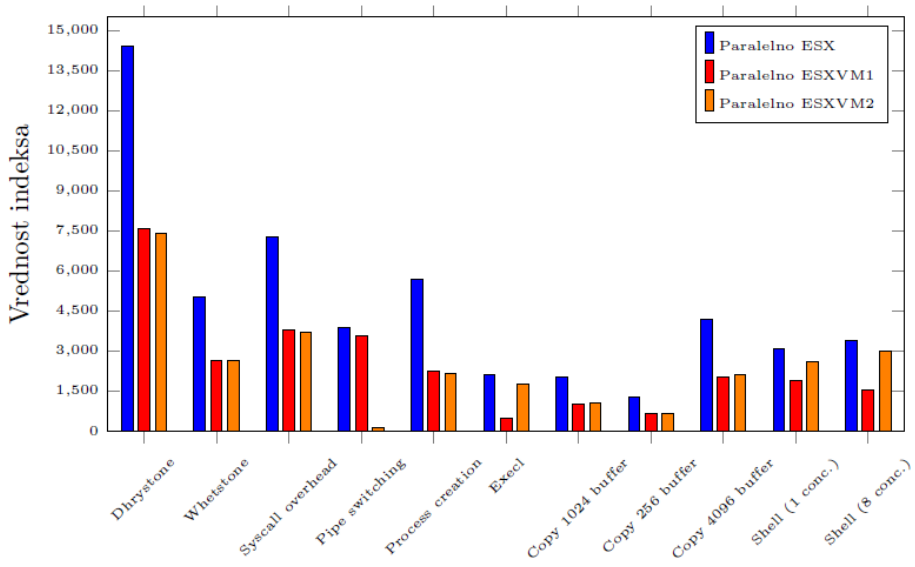
Predvidevali smo, da se zna ponoviti podobnost rezultatov pri prejšnjem testu, saj je sam test enak, vsaka delovna postaja ali programski zabojnik pa ima na voljo vsa sistemska sredstva. Razlika je v tem, da je razporeditev sredstev odvisna od operacijskega sistema ("ESX" ali "OpenVZ"). V najslabšem primeru (da sta obe virtualni delovni postaji ali zabojnika enako obremenjeni) sta sistema nastavljena tako, da se vsakemu virtualnemu sistemu dodeli 50% vseh sredstev. Test zopet teče večkrat.

Oznaki ESX in OpenVZ sta sistema čez vsa sistemska sredstva. Medtem, ko VZ1 in VZ2 (analogno tudi ESXVM1 in ESXVM2) pomenita, da na sistemu vzporedno tečeta dva programska zabojnika (ali virtualni delovni postaji).

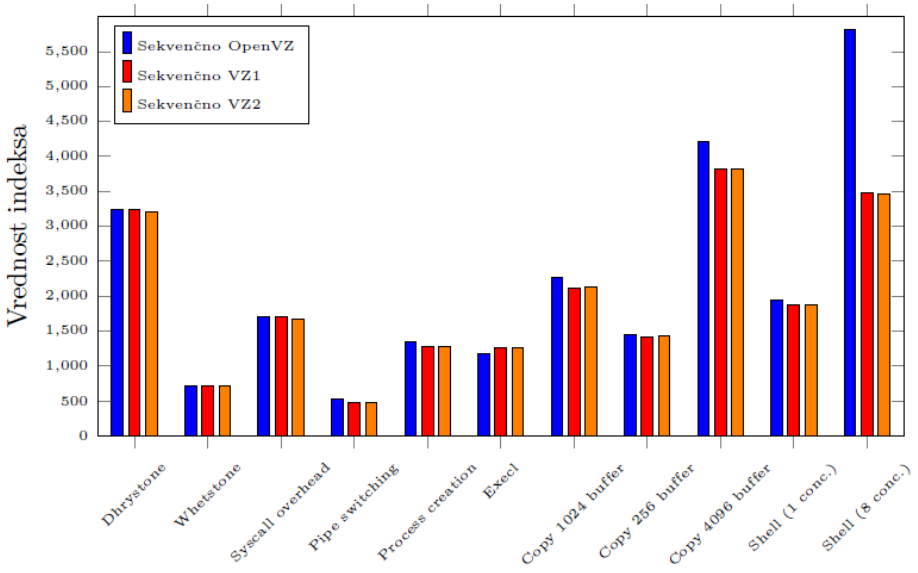
6.5.3 Rezultati



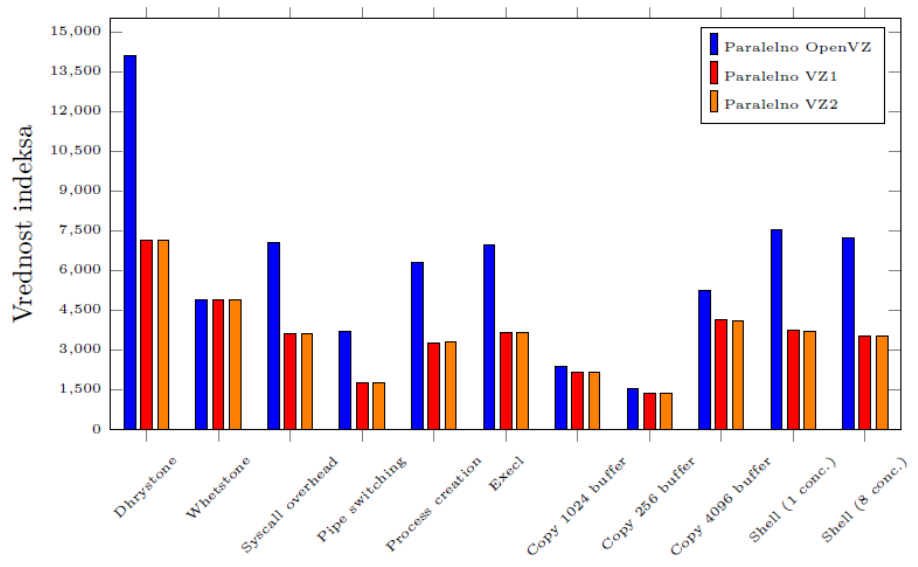
Slika 6.4: Primerjava sekvenčnih testov na ESX sistemu.



Slika 6.5: Primerjava paralelnih testov na ESX sistemu.



Slika 6.6: Primerjava sekvenčnih testov na OpenVZ sistemu.



Slika 6.7: Primerjava paralelnih testov na OpenVZ sistemu.

6.5.4 Komentar

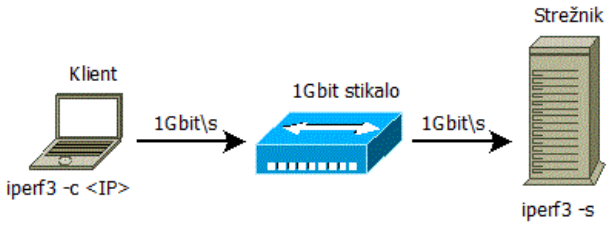
Opazili smo, da se sistemska sredstva razporedijo bolj enakomerno v primeru "OpenVZ". Razlog lahko pripišemo "schedulerju" implementacije, ki razporeja procesor ter strojni paralelnosti procesorja. V primer "ESX" implementacije se lahko zgodi, da je več virtualnih procesorskih jeder preslikanih na isto fizično procesorsko jedro, kar pomeni, da so nekatera jedra bolj obremenjena od drugih in zato pride do neenakosti v razporejanju sistemskih virov (kar se vidi na sliki 6.5).

6.6 Iperf3

6.6.1 Opis testa

Orodje **Iperf3** testira zmogljivost in zanesljivost mrežne opreme sistema (v našem primeru mrežnih kartic). Teste sestavljajo sekvenčno in paralelno pošiljanje podatkov v določenem časovnem okvirju. Rezultati so predstavljeni kot količina podatkov ("transfer"; v enoti "GBytes") v prej omenjenem časovnem okvirju ("interval"; v sekundah), zmogljivost povezave ("bandwidth"; v "Gbits" na sekundo), ter število ponovnih pošiljanj ("retry").
Za naše potrebe smo uporabili sekvenčno pošiljanje (v tabeli 6.3), paralelno pošiljanje 10-ih podatkov (v tabeli 6.4) in paralelno pošiljanje 100-ih podatkov (v tabeli 6.5). Test je bil izveden na istem sistemu, kjer je ena virtualna delovna postaja ali en programski zabojnik služil kot strežnik, drugi pa kot klient. Protokol, ki ga v testu uporabljamo je TCP, ki sam avtomatsko opravlja segmentacijo velikosti podatkov. Pri protokolu UDP pa je velikost privzeta (1024 bytes).

Slika 6.9 predstavlja test, pri katerem je ena virtualna delovna postaja (ali en zabojnik) služila kot strežnik za test **Iperf3**. Kot klienta pa smo uporabili povprečen prenosnik, z mrežno kartico ki podpira 1Gb prenosa. Med klientom in strežnikom smo uporabili 1Gb stikalo (slika 6.8).
Programsko prenosnik uporablja enako distribucijo operacijskega sistema, kot je bila uporabljena za virtualne delovne postaje.



Slika 6.8: Diagram strukture testnega sistema.

6.6.2 Hipoteza

Predvidevali smo, da je sistem "OpenVZ" zmogljivejši pri višjih stopnjah paralelne obremenitve. Testi zajemajo večkratno ponovitev, saj vsak test traja 120 sekund, med katerim se na vsaki 2 sekundi beleži količina prenesenih podatkov in hitrost prenosa. Na koncu pa se količina sešteje, hitrost pa povpreči.

6.6.3 Rezultati

System	Interval	Transfer	Bandwidth	Retry	CPU load
ESXi-VM					
sender	120 sec	467 GBytes	33.5 Gbits/sec	3789	69.8%
receiver	120 sec	467 GBytes	33.5 Gbits/sec	-	61.2%
OpenVZ					
sender	120 sec	25.6 GBytes	1.83 Gbits/sec	0	100.0%
receiver	120 sec	25.6 GBytes	1.83 Gbits/sec	-	56.7%

Tabela 6.3: Rezultati meritev sekvenčnega pošiljanja.

System	Interval	Transfer	Bandwidth	Retry	CPU load
ESXi-VM					
sender	120 sec	187 GBytes	13.4 Gbits/sec	10424	69.2%
receiver	120 sec	187 GBytes	13.4 Gbits/sec	-	19.0%
OpenVZ					
sender	120 sec	28.1 GBytes	2.01 Gbits/sec	0	100.0%
receiver	120 sec	28.1 GBytes	2.01 Gbits/sec	-	76.6%

Tabela 6.4: Rezultati meritev paralelnega (10) pošiljanja.

System	Interval	Transfer	Bandwidth	Retry	CPU load
ESXi-VM					
sender	120 sec	125 GBytes	8.98 Gbits/sec	168627	74.4%
receiver	120 sec	125 GBytes	8.98 Gbits/sec	-	70.1%
OpenVZ					
sender	120 sec	32.7 GBytes	2.34 Gbits/sec	0	100.0%
receiver	120 sec	32.7 GBytes	2.34 Gbits/sec	-	98.8%

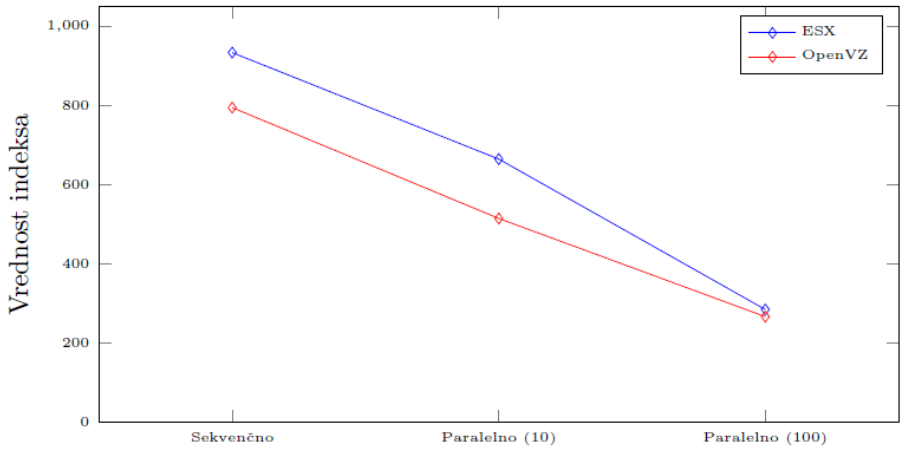
Tabela 6.5: Rezultati meritev paralelnega (100) pošiljanja.

Naslednji graf predstavlja hitrost prenosa pri dostopu iz lokalnega omrežja, ko klient in strežnik nista na isti strojni opremi. Format podatkov v tablebi 6.6 je "Bandwidth / CPU Load". Podatki so samo za klienta, strežnik ima čez celoten

test zasedenost procesorja okoli 20%, nima ponovnih pošiljanj (že zato ker je strežnik), ostali podatki pa so identični klientovim.

	Sequential	Parallel (10)	Parallel (100)
ESXi-VM	934 Mbits / 33.2%	665 Mbits / 91.5%	285 Mbits / 61.4%
OpenVZ	795 Mbits / 87.6%	515 Mbits / 74.4%	267 Mbits / 55.0%

Tabela 6.6: Rezultati meritev testa povezave med klientom in strežnikom.



Slika 6.9: Primerjava hitrosti prenosa sistemov.

6.6.4 Komentar

Glede na rezultate testov implementacije mrežne opreme na naših testnih sistemih, lahko povzamemo, da je bila naša predhodnja trditev napačna, saj virtualni strežnik "ESXi" bolje izrablja povezavo s strojno opremo, medtem ko sistem "OpenVZ" realizira mrežne kartice programsko. Pri slednjem sistemu, je iz testov razvidno, da je ozko grlo in omejevalni faktor procesor, saj je pri vseh testih maksimalno obremenjen. Da je implementacije programska, je razvidno tudi iz podatka, da pri "OpenVZ" ni bilo potrebnih ponovnih pošiljanj datagamov.
Na sliki 6.8, pa lahko opazimo, da hitrosti postajata enaki glede na več paralelnih pošiljanj. Manjše razlike v hitrosti lahko pripišemo različni implementaciji mrežne opreme med sistemoma. V tem testu ni procesor ozko grlo, saj je zasedenost procesorja v najslabšem primeru okoli 90%. V rezultate nismo vključili števila ponovnih pošiljanj, saj jih je v obeh primerih podobno število, kar ni presenetljivo, saj gre ves mrežni promet skozi strojno opremo. Razliko v hitrosti lahko pripišemo programski povezavi med strojno opremo in virtualnim sistemom, saj je strojna oprema in infrastruktura v obeh primerih enaka.

Poleg testa TCP protokola smo naredili tudi test konsistentnosti povezave (UDP protokol), kjer smo opazovali "jitter" (nihanje) na sami povezavi. Tu pride zopet do razlike med "ESX", kjer obstaja nihanje zaradi izrabljanja strojne opreme in "OpenVZ", kjer zaradi programske implementacije tega ni.

6.7 Testiranje preobremenitve sistema

6.7.1 Opis testa

S sledečim testom smo hoteli preizkusiti zmogljivost in odzivnost sistema v primeru, ko ima sistem uporabljene vse sistemske vire. Scenarij testa je izgledal tako, da sta na sistemu tekli dve virtualni delovni postaji ali dva programska zabojnika. Na eni izmed njih smo pognali program forkBomb.c (rekurziven klic funkcije fork(); listing 6.1) napisan v programskem jeziku C, kateri je ustvarjal eksponentno naraščajoče število procesov, dokler ni sistem postal neodziven. Sistem je zavračal oddaljen dostop, kakršnokoli interakcijo in porabil vse razpoložljive sistemske vire. Medtem je bila druga delovna postaja (ali programski zabojnik) nobremenjen. Na sledčem sistemu smo pognali test **UnixBench** in primerjali rezultate s rezultati pridobljenimi v prejšnjih testih.

Listing 6.1: forkBomb.c

```
   #include <unistd.h>

   int main(int argc, char **argv) {
      int forkPID = fork();
      if(forkPID == 0)
         execve(argv[0], argv, NULL) < 0;
      else
         execve(argv[0], argv, NULL) < 0;
   }
%\caption{forkBomb.c}
%\label{lst:forkBombLabel}
```

Omejili smo se na tri teste, ker ostali nimajo takega vpliva pri odzivnosti, to so "Dhrystone", "execl troughput" in "pipe-based context switching" (razlaga posameznih testov pri orodju **UnixBench**). Med vsakim testom se je počakalo dve minuti, da so se sredstva razporedila nazaj.

6.7.2 Hipoteza

Predvidevali smo, da neodzivnost ene delovne postaje ne bi smela imeti vpliva na vse ostale na sistemu, saj mora operacijski sistem direktno nad strojno opremo ("hypervisor") skrbeti za razdeljevanje sistemskih sredstev. Sistem v primeru nezasedenosti sredstev dodeli vsa razpoložljiva, tisti delovni postaji, ki jih potrebuje. V kolikor bi del sredstev potrebovala druga, je naloga "scheduler"-ja,

da sredstva razporedi glede na to koliko odstotkov virov kateri delovni postaji pripada. V našem primeru je bil sistem konfiguriran tako, da vsaka delovna postaja dobi 50% vseh virov, če jih potrebuje.

6.7.3 Rezultati

Primerjava je zgolj v vrednosti indeksov, saj se dosti hitro opazi razliko. Oznaki ESX in OpenVZ pomenita referenčne rezultate (pridobljene v poglavju 1.4). ESX-OL in OpenVZ-OL pa pomenita rezultate testov s preobremenjeno delovno postajo ali programskim zabojnikom.

"Benchmark name"	ESX	ESX-OL	OpenVZ	OpenVZ-OL
Sequential				
Dhrystone (2 register variables)	3538.8	2127.3	2260.3	2272.0
Execl troughput	721.2	254.6	1239.8	156.9
Pipe-based contextswitching	151.1	14.5	519.1	487.3
Parallel				
Dhrystone (2 register variables)	14414.2	9234.6	9344.4	4225.7
Execl troughput	2143.5	1999.6	7370.5	171.9
Pipe-based contextswitching	3907.9	106.0	3363.3	753.9

Tabela 6.7: Rezultati meritev pri preobremenjenem sistemu.

6.7.4 Komentar

Test si zasluži poseben komentar, saj tako obnašanje za pravilno konfiguriran sistem ni tipično. Vsaka delovna postaja na sistemu "ESX" je imela virtualno dodeljeno enako količino sredstev kot jih ima celoten sistem. Pri načrtovanju sistema, se sredstva dodeljuje z nekaj rezerve, tako da do popolne neodzivnosti ne mora priti. Kar je zanimivo je, da se sredstva pri nekaterih testih ne razporedijo pravilno, kar pripisujemo dodelitvi sredstev in konfiguraciji.
Iz rezultatov opazimo, da so nekateri testi občutno slabši kar sicer pomeni, da da bi neodzivnost vplivala samo na določene vrste storitev. Dostop do neobremenjene delovne postaje je deloval brez problema. Medtem ko je bilo treba neodzivno ponovno zagnati preko "hypervisorja".

V primeru sistema "OpenVZ" je možnost, da bi zabojnik postal neodziven pri normalnem delovanju zelo majhna. V vseh poizkusih, da bi sistem postal neodziven, je še preden bi lahko prišlo do tega servis ("deamon") "oom-killer" ("Out-Of-Memory killer"), naš program končal ker je sistemu začelo primanjkovati pomnilnika. Da je sistem postal neodziven, smo morali sami onemogočiti prej omenjeni servis. Rezultati vsega so precej slabi, kar niti ni presenetljivo, saj smo onemogočili obrambni mehanizem sistema, točno proti taki vrsti "napake" [48].

6.8 Zaključek

Tekom seminarske naloge smo testirali sistemsko in omrežno zmogljivost dveh različnih implementacij virtualnega strežnika. Poizkušali smo ugotoviti katera implementacija je bolj primerna za različne vrste aplikacij in storitev.
Za storitve, ki vključujejo komunikacijo znotraj enega sistema, je boljša izbira "VMware ESX", saj uporablja strojno opremo kot pomoč pri komunikaciji. Pri "OpenVZ" pa komunikacija med drugimi zabojniki na istem sistemu poteka preko programsko implementirane opreme.
Storitve, ki uporabljajo visoko stopnjo paralelnosti so bolj smiselne s sistemom "OpenVZ", saj uporablja paralelnost strojne opreme, medtem ko lahko imamo pri "ESX" več emulacij procesorjev kot jih je v dejanski strojni opremi (tej emulacije procesorjev si delijo vsa sredstva fizičnega sistema).
Glede varnosti obeh sistemov ima prednost "OpenVz". Program (aplikacija) znotraj svojega zabojnika vidi samo sliko sistema na katerega nima direktnega vpliva. "ESX" pa nima dostopa do strežniškega operacijskega sistema, vendar ima administratorski (seveda odvisno od uporabniških pravic) dostop do svoje virtualne delovne postaje.
Zaključimo lahko, da nobena od implementacij ni boljša od druge, lahko samo izberemo primernejšo glede na namen storitve, ki jo mislimo nuditi. Prav tako sta obe virtualni implementaciji lahko uporabljeni za sisteme z visoko razpoložljivostjo ("high availability"), saj je zaradi velikega števila mehanizmov redko, da bi sistem v celoti postal neodziven.

Poglavje 7

Analiza zmogljivosti oblačnega sistema za hrambo datotek

Jure Jesenšek, Luka Prijatelj, Tine Šubic

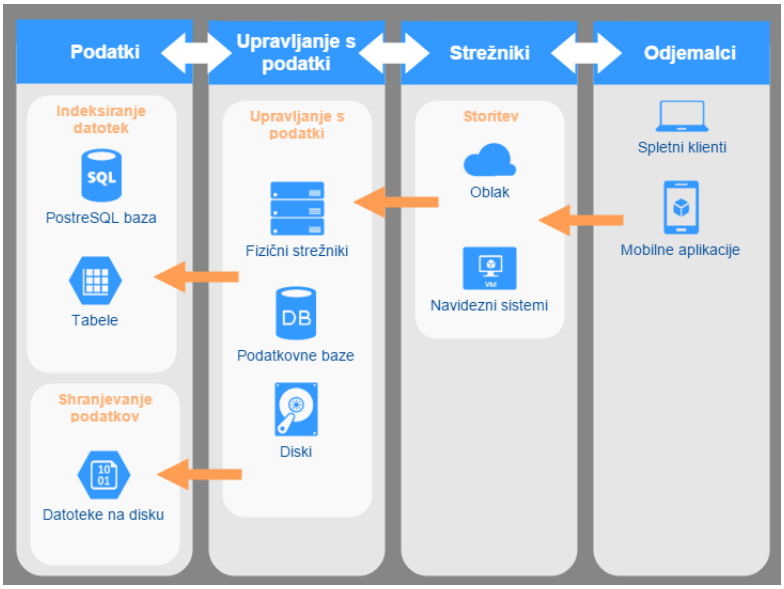
7.1 Predstavitev ideje

Naša skupina si je zadala nalogo izmeriti in stestirati razlike med strežniki v oblaku, razvojnim strežnikom Inštututa Jožef Stefan ter navadnim PC računalnikom, z namenom prenosa datotek od klienta k strežniku ter obratno.

V ta namen smo razvili aplikacijo za hrambo datotek v oblaku, ki omogoča prenos uporabniških datotek na strežnik in s strežnika nazaj na lokalni sistem. Možen je tudi prenos večih datotek hkrati po principu storitve Google Drive, ki ob takem zahtevku datoteke arhivira kot ZIP arhiv.

7.2 Opis delovanja

Interakcija odjemalca in strežnika je implementirana po principu REST s HTTP zahtevki. Ob prejemu POST zahtevka za nalaganje datoteke strežnik prenese datoteko na disk in ji v bazi dodeli unikaten ključ. Če prejme GET zahtevek s ključem za eno samo datoteko, poišče ime datoteke v bazi in najdeno datoteko vrne odjemalcu, če pa je v zahtevku datotečnih identifikatorjev več, strežnik datoteke najprej arhivira v en sam arhiv in ga odpošlje odjemalcu.



Slika 7.1: Diagram realnega sistema.

Na sliki 7.1 je prikazan diagram uporabe realnega sistema, na sliki 7.2 pa diagram simulacijskega okolja.

7.3 Izbira gostiteljskega sistema

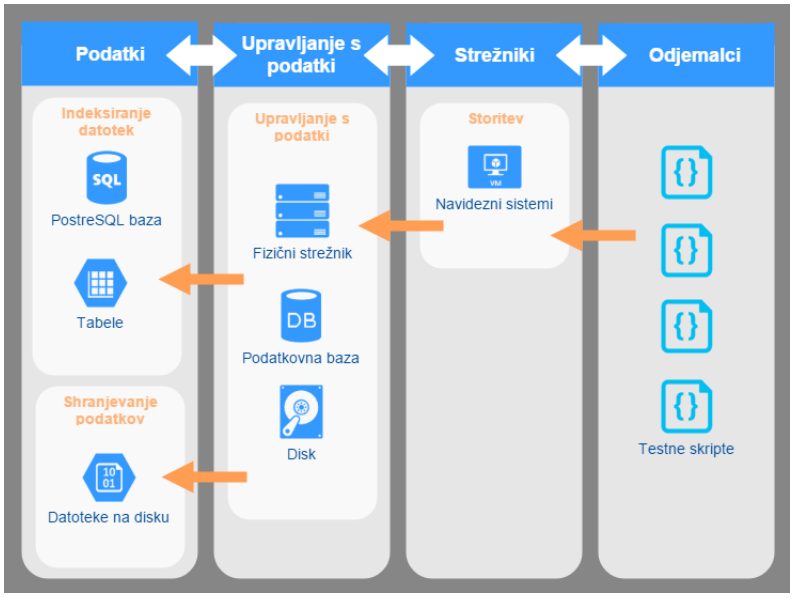
Za testiranje aplikacije smo pripravili tri platforme:

- **Digital Ocean Instance** [49] **:** manjši virtualni strežnik, 1x večjedrni procesor (do 3.3 GHz), 512 MB pomnilnika, Ubuntu Server Linux OS,
- **Domači računalnik običajnih zmogljivosti:** 1x 4-jedrni procesor (do 3 GHz), 12 GB pomnilnika, Windows Server OS,
- **Razvojni strežnik Instituta Jožef Stefan:** 4x 8-jedrni procesor (do 3 GHz), 512 GB pomnilnika, Windows Server OS.

Poleg naštetih sistemov, ki so izvajali vlogo strežnikov, smo za potrebe odjemalca uporabili računalnik srednjega zmogljivostnega razreda - 1x 4-jedrni procesor (do 3,4 GHz), 8 GB pomnilnika, Windows 10 ter optično povezavo s hitrostjo 100/10 Mb/s. Poleg opisanega PCja sta pri testih zmogljivosti posameznega strežnika vlogo odjemalca izvajala tudi preostala dva strežnika.

7.4 Izbira tehnologij

Izbrali smo dve tehnologiji za razvoj strežnika, podatkovne baze in odjemalcev.



Slika 7.2: Diagram simulacijskega okolja.

7.4.1 NodeJS

Strežniška aplikacija je implementirana na NodeJS [50] ogrodju. Ta zaradi vgrajenih orodij in velikega števila dodatnih paketov funkcionalnosti omogoča hiter razvoj spletnih aplikacij, ki potrebujejo skalabilnost in asinhrono izvajanje večjega števila operacij.

Tudi odjemalci so napisani s pomočjo ogrodja NodeJS, saj nam je to omogočilo fleksibilnost in prenosljivost sistema.

7.4.2 PostgreSQL

Za podatkovno bazo v ozadju smo izbrali PostgreSQL [51], odprtokodno rešitev za SQL podatkovne baze, kjer so podatki shranjeni v tabelah. Paket **node-postgres** [52] omogoča hitro asinhrono povezavo med bazo in NodeJS strežnikom.

7.5 Implementacija sistema

7.5.1 Strežniška aplikacija

Pripravili smo strežniško aplikacijo na ogrodju NodeJS, ki implementira REST vmesnik s POST in GET metodami. Odjemalec lahko preko teh metod zahteva datoteke, ali pa jih nalaga na strežnik.

7.5.2 Nalaganje datotek na strežnik

Odjemalec pošlje HTTP POST zahtevek, ki vsebuje datoteko za prenos. Ob prejemu strežnik shrani datoteko na disk in ji v bazi dodeli identifikator, ki ga vrne odjemalcu v odgovoru na zahtevek. Ime datoteke na disku je konkatenacija dejanskega imena, prejetega v zahtevku in unikatnega identifikatorja, kar omogoča nalaganje večih datotek z istim imenom.

7.5.3 Prenos datotek s strežnika k odjemalcu

Odjemalec pošlje HTTP GET zahtevek strežniku. Zahtevek vsebuje seznam datotečnih identifikatorjev. Če je identifikator en sam, sistem poišče datoteko v bazi in jo pošlje odjemalcu. Če je identifikatorjev več, sistem najprej zgradi ZIP arhiv sestavljen iz zahtevanih datotek, ta arhiv pa nato pošlje odjemalcu. Če datoteke ne najde, odjemalec dobi temu ustrezno sporočilo.

Za arhiviranje datotek smo uporabili knjižnico **adm-zip** [54], ki omogoča sestavljanje ZIP arhivov iz več datotek na disku.

7.5.4 Podatkovna baza

Na strežniku smo postavili PostgreSQL podatkovno bazo, ki v tabeli hrani 2 polji - identifikatorje in imena datotek na disku. Strežnik bo z njo povezan s knjižnico node-postgres. Unikatni identifikatorji se ustvarijo s knjižnico **node-uuid** [55].

7.5.5 Odjemalec

Namen odjemalca je pošiljanje GET in POST zahtevkov z vsebino datotek ter merjenje časa potrebnega za odgovor.

7.6 Meritve in testiranje

Meritve so se izvajale tako na strežniku, kot tudi pri odjemalcih. Splošno dostopnost do strežnikov smo testirali tako, da smo v roku 24 ur periodično izvajali ping zahtevke ter si beležili dostopne čase. Iz teh smo lahko ocenili v katerem obdobju dneva so dostopni časi najkrajši. Pri odjemalcih si tudi zapisujemo čase prenosov zahtevkov, saj nas predvsem zanima kakšen vpliv na odzivni čas strežnika imajo dostopi, ki zahtevajo kompresiranje datotek.

Prenos datotek na strežnik in z njega smo testirali na dva načina. Najprej smo uporabili enostaven linearen način z enim klientom, ki je zaporedoma pošiljal zahtevke strežniku in postopoma povečeval velikosti datotek. V drugem delu testa smo uporabili komplesnejši pristop z večjim številom odjemalcev ki vzporedno prenašajo datoteke neodvisno drug od drugega.

7.6.1 Strežniški del

Strežnik ob vsakem API zahtevku zabeleži čas, ki ga je porabil za obdelavo zahtevka (čas od prejema zahtevka do zaključitve odgovora). Njegovo delo je, da shrani poslano datoteko na disk, ter zanjo generira unikaten identifikator in ga shrani v podatkovno bazo. Odjemalcu je identifikator vrnjen v odgovoru na zahtevek. V primeru, da želi odjemalec prenesti določeno datoteko, ki jo je predhodno že naložil na strežnik, potem potrebuje njen identifikator.

7.6.2 Odjemalec

Odjemalec beleži čas, ki je potreben za potek celotnega zahtevka, poleg tega pa ima še naslednje osnovne funkcije:

- ping strežnika - merjenje odzivnega časa strežnika,
- spremljanje časa odgovora strežnika od trenutka, ko odjemalec pošlje API zahtevek,
- nalaganje ene binarne datoteke na strežnik (slika JPG, TXT besedilo, format BIN...),
- zahteva za prenos ene binarne datoteke s strežnika (JPG slika, TXT besedilo, BIN format...),
- zahteva za prenos več datotek v formatu ZIP s strežnika.

7.6.3 Bremena

Prenose na strežnik simuliramo z bremeni velikosti od 10 kB do 5 MB. To so datoteke različnih formatov: TXT, PNG, ZIP in PDF. Za vsak velikostni razred smo naključno generirali 10 datotek vsakega tipa, izmed katerih ob izvedbi testiranja za vsak zahtevek naključno izberemo breme. Na ta način se izognemo anomalijam robnih primerov, predvsem v primeru testiranja kompresije, da dobimo reprezentativen vzorec bremen.

7.6.4 Uporabljene skripte

Skripta za testiranje časa dostopnosti

Skripta za teste časa dostopnosti je napisana v jeziku Javascript. Kot argumente v ukazni vrstici prejme IP naslov strežnika in mrežna vrata ki jih testiramo. Skripta inicializira datoteke za beleženje podatkov in nato s 15 sekundim intervalom pošilja PING paket na zahtevan strežnik ter si beleži odzivne čase.

```
/**
 * Created by subic on 18. 04. 2016.
 */
```

```
var fs = require('fs');
var path = require('path');

//Install these with NPM!
var ArgumentParser = require('argparse').ArgumentParser
var tcpp = require('tcp-ping');

var requestNumber = 0;

var parser = new ArgumentParser({
    version: '0.0.1', addHelp:true,
    description: 'ZZRS PING script'
});
parser.addArgument(
    [ '-s', '--server' ],{help: 'Remote server'}
);

parser.addArgument(
    [ '-p', '--port' ], {help: 'Remote port'}
);

var args = parser.parseArgs();
if(!args.server) {
    console.log("Missing server.")
    process.exit(1);
}

if(!args.port) args.port = 3000;

console.log("Pinging: " + args.server + ":" + args.port);
var logName = './ping_results_' + args.server + "_" + new Date().getTime() +
    ".txt";
var columnNames = "ID\tday_time\tresponse_time\r\n"

 fs.writeFileSync(logName, columnNames, 'utf8', function (err) {
     if (err) console.log(err);
 });


setInterval(function() {
    if (requestNumber < 60*24*4) {
        requestNumber++;

        tcpp.ping({address: args.server, port: args.port, attempts: 1}, function
            (err, data) {
            if (err) console.log(err);

            var data = requestNumber + "\t" + new Date() + "\t" + data.avg +
                "\r\n";
            fs.appendFileSync(logName, data, 'utf8', function (err) {
```

```
                if (err) console.log(err);
            });
        });
    }

}, 15000);
\label{Test.}
```

Skripte za prenos datotek

Tudi te skripte so napisane v jeziku Javascript. Same kode ne prilagamo v tem poročilu, saj posamezna skripta obsega med 200 in 400 vrstic kode - obdelava argumentov, izbiranje, branje in pošiljanje datotek, beleženje podatkov, itd.

Vse skripte kot argumente v ukazni vrstici sprejmejo naslednje podatke: IP naslov strežnika in mrežna vrata ki jih za HTTP zahtevke, tip datotek (Besedilo, slike, binarno), in število vzorcev datotek za testiranje. Po nastavitvi parametrov skripte inicializirajo datoteke za beleženje podatkov in naložijo seznam datotek za prenos - lokalno ali preko API zahtevka na strežniku. V glavni fazi delovanja skripte izvedejo HTTP GET ali POST zahtevek in prenesejo datoteko. Ime datoteke, čas za prenos, in velikost datoteke se zapišejo v log.

7.7 Rezultati

7.7.1 Testi dostopnosti

Pristop

Pri testih dostopnosti smo poskušali ugotoviti ali za dan gostiteljski sistem obstaja časovni interval, ki bi bil s stališča dostopnega časa neobremenjenega strežnika najugodnejši.

Teste smo vršili tako, da smo v intervalih dolžine 15 sekund pošiljali PING zahtevke na ustrezen strežnik in beležili čase, potrebne za prejem odgovora. To smo počeli 24 ur in s tem dobili 5760 rezultatov. Tak test smo za vsak strežnik ponovili trikrat in nato rezultate povprečili da smo se znebili anomalij.

Test: virtualni strežnik

Prvi test je bil izveden z virtualnim strežnikom podjetja Digital Ocean, lociranem v strežniškem centru v New Yorku. Odjemalec se je nahajal v Kranju, za dostop pa je bila uporabljena optična povezava z zmogljivostjo 100/10 Mb/s, meritev pa je bila izvedena trikrat. Rezultati meritve so vidni na sliki 7.3. Postavili smo si hipotezo, da bo tu odzivni čas višji kot pri ostalih sistemih zaradi lokacije strežnika, pa tudi da bomo opazili višja nihanja odzivnih časov med

Slika 7.3: Testiranje odziva strežnika, Digital Ocean Virtualni strežnik.

dnevnim časom evropskih časovnih pasov.

Rezultati potrjujejo naši hipotezi. Povprečni odzivni časi se gibajo med 105 in 115 milisekund z občasnimi skoki do 135 milisekund. Iz grafa 7.3 je razvidno da se število konic z visokimi odzivnimi časi močno poveča okrog devete ure zjutraj (po lokalnem času, sicer 7:00 po UTC). Ta trend se nadaljuje do polnoči (po lokalnem času, sicer 22:00 UTC), nato pa se število konic zmanjša. Gre za del dneva, v katerem je na internetu aktivna večina populacije, tako v okviru službenih dolžnosti kot prostočasnih aktivnosti, kar povzroča večjo gostoto prometa.

Test: strežnik IJS

V drugem testu je bil strežnik postavljen na razvojnem sistemu Inštituta Jožef Stefan v Ljubljani. Metodologije meritev je bila tu enaka ko pri prvem testu, prav tako tudi lokacija odjemalca. Naša hipoteza je bila, da bomo zaradi bližje geografske lokacije opazili znatno nižje dostopne čase kot pri oddaljenem virtualnem strežniku.

Tudi v tem primeru so rezultati meritev (Vidni na sliki 7.4)potrdili našo hipotezo. Povprečni dostopni časi so za cel velikostni red manjši od časov, izmerjenih pri oddaljenem strežniku, in sicer med 1 in 10 milisekund, z občasnimi nihanji do 40 milisekund. Tudi tu so večja nihanja lokalizirana v dnevnem, predvsem popoldanskem času. Razlika je tu še posebej očitna zaradi že tako nižjih dostopnih časov. V eni od meritev smo opazili smo tudi močen skok med

Slika 7.4: Testiranje odziva strežnika, razvojni strežnik IJS.

osmo in deveto uro (nekaj 1000 milisekund), za kar predvidevamo da je posledica povečane aktivnosti ali motnje v omrežju odjemalca, saj se podoben skok pojavlja ob istem času tudi na grafih drugih dveh testov, izvedenih tisti dan.

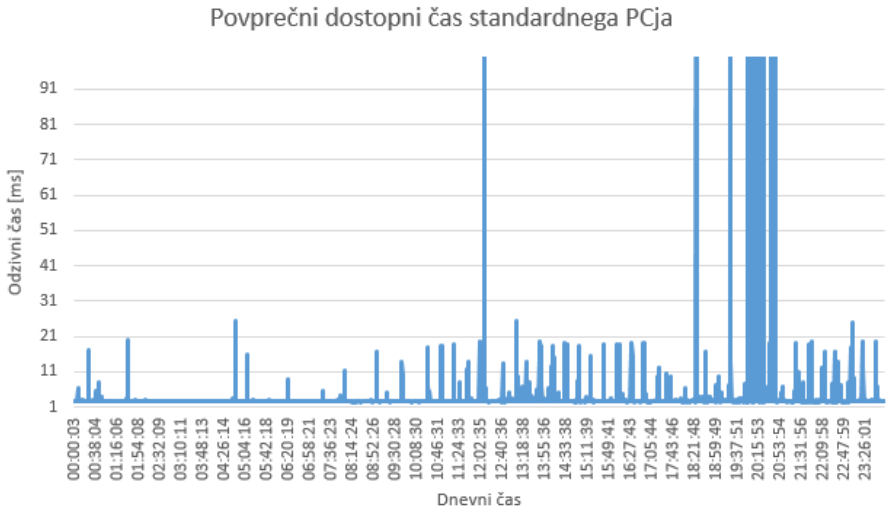
Test: standardni PC

Zadnji strežnik je bil postavljen na običajnem namiznem računalniku v Ljubljani in povezan na omrežje z optično povezavo. Tu smo predvidevali, da bodo rezultati zelo podobni kot pri strežniku IJS, saj sta si sistema geografsko zelo blizu, na test pa strojna oprema (razen omrežne povezave) ne vpliva. Rezultati so vidni na sliki 7.5. Povprečni dostopni čas se giba med 1 in 5 milisekund, s skoki do 25 milisekund. Malenkost višje vrednosti v prejšnjem eksperimentu pripisujemo predvsem omrežju IJS, ki vsebuje notranje požarne zidove in filtriranje mrežnega prometa, kar najverjetneje povzroči nekaj milisekundni zamik pri odzivu. Na grafu 7.5 ponovno opazimo nekaj zalo velikih skokov, ki so verjetno posledica aktivnosti v odjemalčevem omrežju.

7.7.2 Testi zaporednega nalaganja datotek na strežnik

Pristop

Tu smo testirali kako se spreminja čas, potreben za obdelavo zahtevka v odvisnosti od velikosti datoteke, ki jo je odjemalec poslal na strežnik. Uporabili smo zaporedno pošiljanje datotek s 50 zahtevki v vsakem velikostnem razredu (50kB,



Slika 7.5: Testiranje odziva strežnika, navadni PC.

100kB, 500kB, 1000kB, 2500kB, 5000kB). Testi so bili izvedeni preko optične povezave na sistemih standardne zmogljivosti, ki predstavljajo povprečnega odjemalca take storitve. Test vsakega sistema je bil izveden trikrat, rezultati pa nato povprečeni da smo se znebili anomalij.

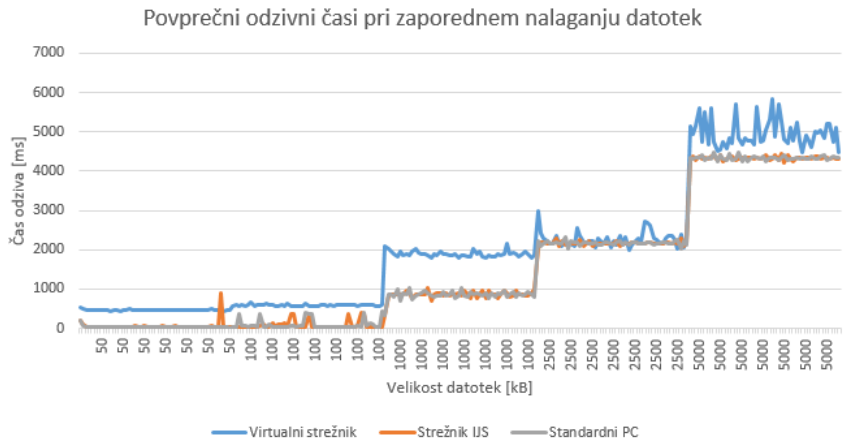
Hipoteze

Pred začetkom testiranja smo postavili nekaj osnovnih hipotez. Predpostavili smo, da se bo najslabše odrezal virtualni strežnik, saj je po strojnih specifikacijah najslabši, hkrati pa je geografsko zelo oddaljen od odjemalcev. Nadalje smo predvidevali, da se bo najbolje odrezal IJS strežnik zaradi velike strojne zmogljivosti, navadni PC pa se bo umestil nekje vmes.

Rezultati so vidni na grafu 7.6.

Rezultati

Naša predvidevanja se niso uresničila popolnoma. Iz grafa 7.6 je razvidno da smo potrdili svojo prvo hipotezo, saj je virtualni strežnik res pokazal najslabše rezultate. Poleg tega, da je bil povprečni čas procesiranja za vsak velikostni razred precej višji kot pri ostalih sistemih, so se pri večjih velikostih pojavljala tudi velika nihanja odzivnega časa, kar je posledica same narave virtualnega strežnika v oblaku.



Slika 7.6: Testiranje odziva na zaporedne zahtevke za nalaganje datotek, vsi sistemi.

Poleg tega da je bil izmed uporabljenih sistemov le-ta najmanj zmogljiv, si mora tak strežnik deliti strojne in omrežne vire ostalimi virtualnimi sistemi na gostitelju. To privede do ozkih grl, med drugim tudi pri bralnih dostopih do diska in podatkovne baze za katere predvidevamo, da so bili poleg geografske lokacije drugi kritični vzrok nihanja odzivnih časov.

Na naše presenečenje so si bili rezultati na standardnem PCju in strežniku IJS zelo podobni. Vzrok tega je naš način testiranja. Ker smo izvajali enostaven test z enim klientom in zaporednimi prenosi, dejanske zmogljivosti niso posebej vplivale na ta dva sistema. Domnevamo da bomo videli večje spremembe v testu ko bomo simulirali več vzporednih prenosov naenkrat.

7.7.3 Testi vzporednih nalaganj datotek na strežnik

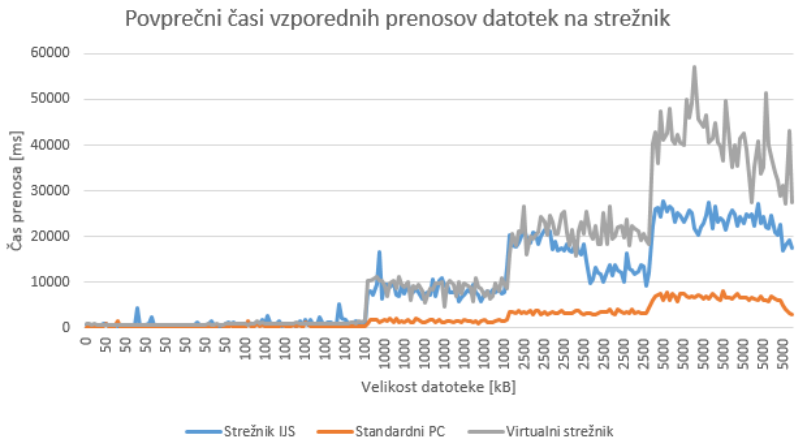
Pristop

Pri tem testu smo ugotavljali, kako se spreminja čas, potreben za prenos datotek na strežnik če sočasno poženemo večje število odjemalcev. Vsak od odjemalcev je uporabljal zaporedno pošiljanje s 50 zahtevki (poslanimi datotekami) za vsak velikostni razred (50 kB, 100 kB, 500 kB, 1000 kB, 2500 kB, 5000 kB). Odjemalske skripte so bile porazdeljene na večih fizičnih sistemih v različnih omrežjih, ki so bili na omrežje povezani z optično povezavo. Test za vsak sistem je bil izveden trikrat, rezultati pa nato povprečeni za čimboljšo reprezentativnost podatkov.

Hipoteze

Postavili smo si naslednje hipoteze: Sistemi ki se zanašanjo na trde diske (HDD) se bodo odrezali slabše kot tisti s solid state diski (SSD). Ker je NodeJS strežnik enonitna aplikacija, smo domnevali tudi da bodo sistemi s hitrejšimi jedri v CPE lažje servirali večje število zahtevkov.

Rezultati so vidni na grafu 7.7.



Slika 7.7: Testiranje odziva vzporednega nalaganja datotek na strežnik, vsi sistemi.

Rezultati

Našo prvo hipotezo smo delno potrdili. Naš testni PC in virtualni strežnik sta oba uporabljala SSD, medtem ko je IJS strežnik Mustang za hrambo podatkov uporabljal HDD. Glede na to da se je pri večjih datotekah najslabše odrezal virtualni strežnik v oblaku, lahko domnevamo da je na ta čas še vedno močno vplivala latenca čezoceanske povezave in nizka frekvenca procesorja.

Bolj primerljiva pa sta rezultata PCja in IJS strežnika, saj se sistema nahajata na približno enako razdalji od odjemalcev (oz. je razlika razdalje zanemarljiva glede na hitrost povezave). Za večje datoteke se je PC odrezal precej bolje, kar je najverjetneje posledica razlike v hitrosti med SSD in HDD.

7.7.4 Testi zaporednega prenosa datotek s strežnika

Pristop

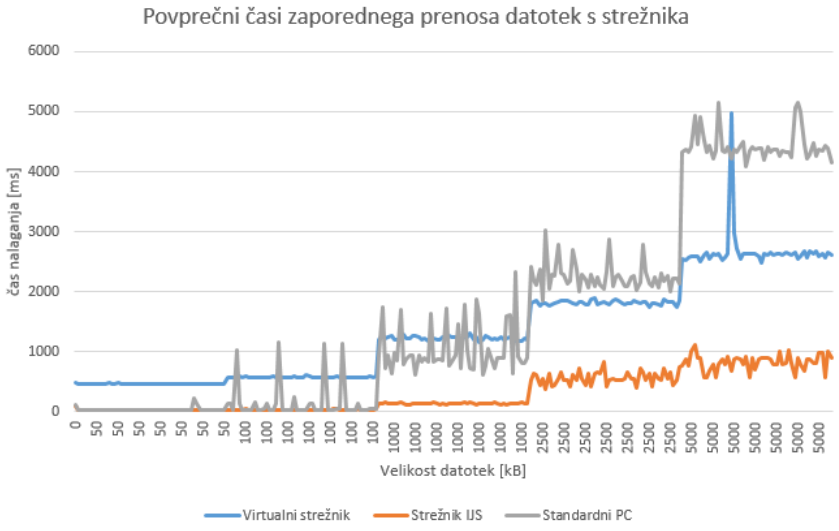
Testirali smo kako se spreminja čas, potreben za obdelavo zahtevka v odvisnosti od velikosti datoteke, ki jo je odjemalec zahteval od strežnika. V testu odjema-

lec najprej od strežnika pridobi seznam datotek na strežniku - za testne namene smo uporabili po 10 naključno generiranih datotek za vsak velikostni red (50kB, 100kB, 500kB, 1000kB, 2500kB, 5000kB). Odjemalec nato zaporedno zahteva po eno naključno izbrano datoteko ter to stori po petdesetkrat za vsako kategorijo velikosti. Z naključnim izbiranjem poskušamo čimbolj izničiti vpliv robnih primerov naključnih datotek in predpomnjenja. Za vsako datoteko si odjemalec zabeleži njeno velikost, čas za odziv in prenos s strežnika. Vsi testi so bili izvedeni z enega odjemalca z optično povezavo, ponovljeni trikrat, rezultati pa nato povprečeni za čimboljšo reprezentativnost podatkov.

Hipoteze

Pred testiranjem smo si postavili naslednje hipoteze: domnevali smo, da se bo ponovno najslabše odrezal virtualni strežnik zaradi manjše strojne zmogljivosti in da bomo pri tem opazili tudi največja nihanja časa obdelave podatkov za večje datoteke. Predvidevali smo tudi, da se bo IJS strežnik odrezal najbolje, navadni PC pa nekje vmes.

Rezultati so vidni na grafu 7.8.



Slika 7.8: Testiranje odziva na zaporedne zahtevke za nalaganje datotek, vsi sistemi.

Rezultati

Naša predvidevanja se niso uresničila popolnoma. Iz grafa 7.8 je razvidno, da smo potrdili hipotezo o strežniku IJS. Zaradi geografske lokacije in večje strojne

zmogljivosti je ta sistem deloval najbolje. Pri majhnih datotekah je bilo delovanje podobno navadnemu PCju, a se je očitna razlika pokazala pri večjih datotekah.

Virtualni strežnik Digital Ocean se je odrezal presenetljivo dobro. Pri manjših datotekah so bili zaradi oddaljenosti sicer odzivni časi višji kot pri ostalih sistemih, a pri velikih datotekah se je ta strežnik odzival hitreje kot navadni PC, hkrati pa so bili časi bolj stabilni, z manjšimi nihanji kot ostala dva sistema. Vzrok za stabilnost je najverjetneje uporaba hitrih SSD diskov v virtualnih sistemih podjetja Digital Ocean, medtem ko sta IJS Mustang in navadni PC uporabljala navadne trde diske za hrambo datotek.

Odzivi navadnega PCja so bili slabši kot smo pričakovali. Kljub nizkim odzivnim časom za majhne datoteke je sistem deloval slabo pri večjih velikostih. Na grafu so vidna tudi večja nihanja kot pri ostalih sistemih.

7.7.5 Testi vzporednih prenosov datotek iz strežnika

Pristop

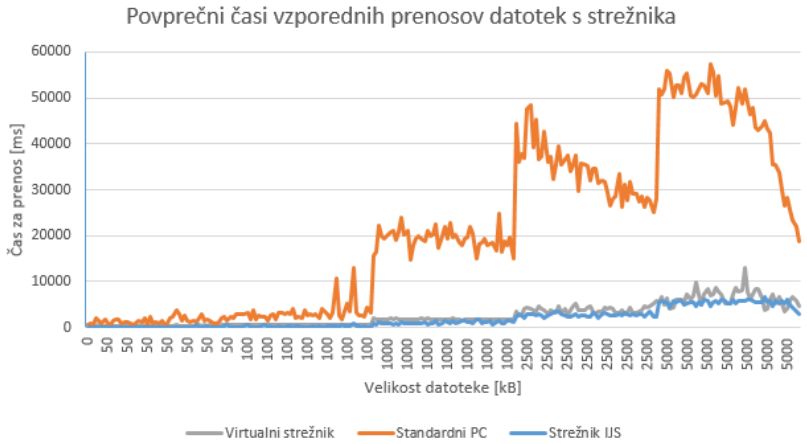
Podobno kot pri testu nalaganja datotek na strežnik, smo podoben test izvedli tudi z vzporednimi prenosi datotek iz strežnika k klientom. Testirali smo kako večje število sočasnih zahtevkov vpliva na čase prenosov. Vsak od klientov je uporabljal zaporedno pošiljanje 50 zahtevkov za vsak velikostni razred (50kB, 100kB, 500kB, 1000kB, 2500kB, 5000kB). Funkcijo skupno 30 odjemalcev so vršili trije sistemi na različnih lokacijah - vsi pa so bili povezani z optično povezavo. Testi so bili izvedeni trikrat, ob različnih časih, rezultati meritev pa so povprečeni.

Hipoteze

Podobno kot pri vzporednem nalaganju na strežnik, smo tudi tukaj predpostavili da se bodo strežniki s trdimi diski slabše odrezali kot tisti, ki vsebujejo diske SSD. Poleg tega smo predvidevali da bo na čas vplivala tudi hitrost internetne povezave pri internetnemu ponudniku strežnika (upload hitrost strežnika).

Rezultati

Drugo hipotezo smo potrdili, saj se je najslabše odrezal standardni PC (slika 7.9) s standardno optično povezavo (100Mb/s download, 10Mb/s upload), kjer se je za ozko grlo izkazala nižja hitrost do strežnika proti odjemalcem. Pri virtualnem strežniku smo pričakovali dobre rezultate, saj so sistemi na katerih smo gostili naš strežnik, v večji meri namenjeni pošiljanju datotek proti odjemalcem. Podobno predivdevamo za strežnik IJS Mustang, pri katerem sta verjetno hitrosti povezave visoki v obe smeri.



Slika 7.9: Rezultati testiranja odziva na vzporedne zahtevke, vsi sistemi.

Pri standardnem PCju druga hipoteza glede tipa diskov ne pride v poštev, saj je največja razlika nastala zaradi počasnejše internetne povezave. Pri ostalih dveh strežnikih pa tudi ne zaznamo večjega odstopanja na ta račun, saj bi lahko nekoliko daljše čase odziva virtualnega strežnika pripisali geografski oddaljenosti strežnika (ZDA).

Pri grafu na sliki 7.9 bi radi še omenili, da je vzrok za padajočo krivuljo pri standardnem PCju ta, da je strežnik nekaterim odjemalcem že odgovoril na vse njihove zahtevke in je z njimi prekinil povezavo, posledično se je odzivni čas za ostale odjemalce zmanjšal.

7.8 Zaključek

Sistem ki smo ga razvili deluje zanesljivo in je dovolj skalabilen za širšo uporabo. V testih je sistem vzporedno serviral 30 različnih odjemalcev (višjega števila odjemalcev nismo uporabili zaradi finančnih omejitev), brez kritičnih težav.

Med testiranjem smo odkrili tri glavna ozka grla sistema, in sicer: omrežna povezava, hitrost diskov in frekvenca jeder v CPE. PostgreSQL podatkovna baza je ves čas testiranja delovala brez večjih sprememb v porabi sistemskih virov, saj so vsi testi uporabljali zgolj nekaj tisoč vrstic tabele, kar je za bazo minimalna obremenitev.

V realnem produkcijskem okolju bi te težave lahko rešili na več načinov. V kolikor bi aplikacija tekla v oblaku lahko običajno privzamemo da ima podatkovni center oblačnega ponudnika zadostno pasovno širino za naše potrebe oz. lahko za zadostno pasovno širino doplačamo, medtem ko smo bili mi omejeni

na zmogljivost ki so nam jo nudili naši ponudniki interneta.

V idealnem primeru bi naš sistem tekel v večjem številu strežniških centrov po svetu, saj smo videli da geografska lokacije močno vpliva na prenosne čase in bi na ta način znižali latenco. Večina ponudnikov virtualnih stežnikov ponuja izbiro regije kjer želi uporabnik gostiti aplikacijo. Za večje število uporabnikov bi uporabili t.i. load balancing [56] - sisteme ki so sposobni optimalno porazdeliti promet aplikacije na tak način, da en sam sistem ni preobremenjen. Uporabiti je mogoče tudi samodejno skaliranje sistemov in tako zagotoviti visoko zanesljivost in dostopnost sistemov iz katerekoli regije. S povečanjem števila instanc aplikacije se znebimo tudi potrebe po sistemih z visokimi frekvencami jeder v CPE. Za najboljše delovanje bi morali biti podatki shranjeni na SSD diskih ali na klasičnih trdih diskih, povezanih v RAID 0 [57] načinu.

Med testiranjem smo uporabili vse kredite za storitev Digital Ocean ki smo jih imeli na voljo in doplačali 4 USD.

Poglavje 8

Analiza zmogljivosti odprtokodne storitve Icecast2

Matic Tkalec, Klavdij Starman

8.1 Opis problema

Cilj poglavja je testiranje zmogljivosti 'open source' storitve Icecast2 [60], ki omogoča 'streaming' glasbe z oblaka na skoraj katero koli napravo. Najin namen je bil izvedeti, kje je meja pri obremenjenosti aplikacije glede na dane zmogljivosti strežnika, kjer je oblačna storitev nameščena.

8.2 Uporabljene tehnologije

8.2.1 Icecast2

Icecast2 [60] je strežniška storitev, ki omogoča 'streamanje' zvočnih medijev preko strežnika. Trenutno podpira Ogg, Opus, WebM in MP3 audio 'streame'. Uporablja se za ustvarjanje internetnih radiev ali privatno uporabo in tvrobo poljubnih glasbenih list, ki jih potem lahko poslušamo preko interneta na našem strežniku. Icecast2 [60] je prosta programska oprema, tako da lahko k njenem razvoju prispeva vsak in je izdan pod licenco GNU GPL v2.

8.2.2 Ices

Za to, da sploh lahko predvajamo glasbo pa potrebujemo še odjemalca. V najinem primeru sva uporabila Ices [61], ki je odjemalec za audio streame za Icecast2

[60] server. Njegova naloga je omogočanje zvočnega 'streama' do Icecasta, da lahko do tega dostopa več uporabnikov. Za pravilno konfiguracijo posameznega odjemalca moramo zanj napisati ustrezno konfiguracijo. Ta je shranjena v XML datoteki, ki jo potem z ukazom v terminalu zaženemo in začnemo s 'streamanjem'. Primer ene izmed uporabljenih konfiguracij je sledeč:

```
<?xml version="1.0"?>
<ices:Configuration
xmlns:ices="http://www.icecast.org/projects/ices">
<Playlist>
  <File>/playliste/playlist2.m3u</File>
  <Randomize>1</Randomize>
  <Type>builtin</Type>
  <Module>ices</Module>
</Playlist>

<Execution>
  <Background>1</Background>
  <Verbose>0</Verbose>
  <BaseDirectory>/var/log/ices</BaseDirectory>
</Execution>

<Stream>
  <Server>
    <Hostname>localhost</Hostname>
    <Port>8000</Port>
    <Password>test</Password>
    <Protocol>http</Protocol>
  </Server>

  <Mountpoint>/stream</Mountpoint>
  <Dumpfile>ices.dump</Dumpfile>
  <Name>Electro Radio FM</Name>
  <Genre>Electro</Genre>
  <Description>Electro</Description>
  <URL>http://188.166.166.187:8000/stream</URL>
  <Public>0</Public>
  <Bitrate>128</Bitrate>
  <Reencode>0</Reencode>
  <Samplerate>44100</Samplerate>
  <Channels>2</Channels>
</Stream>
</ices:Configuration>
```

8.2.3 DigitalOcean

V najinem primeru sva za oblačno infrastrukturo izbrala ponudnika DigitalOcean [59] , ki je izredno lahek za uporabo. Nov 'droplet' (strežnik) lahko nare-

dimo v manj kot minuti, njihova tehnična podpora je izredna in na vsak tvoj problem odgovorijo v manj kot 30 minutah. Ponujajo veliko različno zmogljivih strežnikov, katerim se ob povečanju zmogljivosti tudi ustrezno zvišuje cena. Na voljo imamo tudi že same 'droplete' z že naloženim operacijskim sistemom oz. zahtevano tehnologijo (npr. LAMP ali NodeJS). Ponujajo tudi veliko različnih lokacij, tako da lahko brez težav postavimo strežnik kjerkoli na svetu. S pomočjo GitHub developer packa sva dobila vsak po 50 $ kredita za DigitalOcean [59], kar je zadostovalo za vsa najina testiranja.

8.3 Rešitev problema

Testni sistem je nameščen v Frankfurtu. Storitev sva testirala z različnimi bremeni ter na ta način prišla do ustreznih zaključkov.
Pri izbranem ponudniku sva izbrala strežnik, ki ga lahko dobimo za 10 $ na mesec s sledečimi specifikacijami :

- Ubuntu 14.04,
- Intel(R) Xeon(R) CPU E5-2650L v3 @ 1.80GHz,
- 1GB RAM,
- 30GB SSD.

Za poganjanje aplikacije oz. 'streama' na gostitelju uporabljava sledečo programsko opremo:

- Icecast2
- Ices

8.4 Breme

Definirala sva več različnih bremen. Vsa bremena sva definirala z bash skripto, ki uporablja ukaz curl in s tem definira odjemalca, ki posluša naš 'stream'. Skripta za poljubno število odjemalcev:

```
#!/bin/bash
#
# run concurrent curls which download from URL to /dev/null. output total
# and average counts to results directory.
#

# max concurrent curls to kick off


while getopts "n:m:d:s:" opt ; do
    case "${opt}" in
        n)
            nflag="${OPTARG}"
            ;;
        m)
            mflag="${OPTARG}"
            ;;
        d)
            dflag="${OPTARG}"
            ;;
        s)
            sflag="${OPTARG}"
            ;;
    esac
done

max=$mflag
# how long to stay connected (in seconds)
duration=$nflag
# how long to sleep between each curl, can be decimal 0.5
delay=$dflag
# url to request from
URL=http://139.59.137.119:8000/stream$sflag

#####
#mkdir -p results
echo > results
while true
do
count=0
while [ $count -le $max ]
do
   #echo $count
   curl -o /dev/null -m $duration -s -w "bytes %{size_download} avg
```

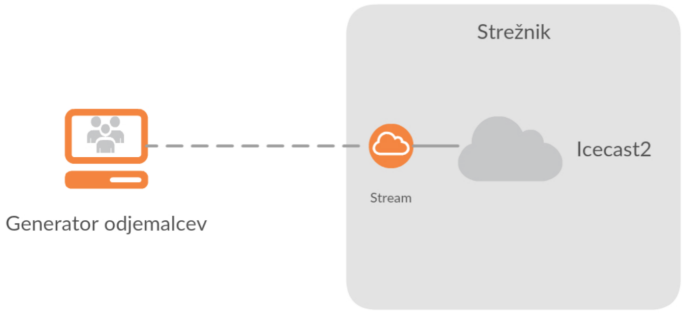
```
        %{speed_download} " "$URL" >> results &
    curl -o /dev/null -m $duration -s -w "bytes %{size_download} avg
        %{speed_download} " "$URL" >> results &
    curl -o /dev/null -m $duration -s -w "bytes %{size_download} avg
        %{speed_download} " "$URL" >> results &
    curl -o /dev/null -m $duration -s -w "bytes %{size_download} avg
        %{speed_download} " "$URL" >> results &
    curl -o /dev/null -m $duration -s -w "bytes %{size_download} avg
        %{speed_download} " "$URL" >> results &
    curl -o /dev/null -m $duration -s -w "bytes %{size_download} avg
        %{speed_download} " "$URL" >> results &
    curl -o /dev/null -m $duration -s -w "bytes %{size_download} avg
        %{speed_download} " "$URL" >> results &
    curl -o /dev/null -m $duration -s -w "bytes %{size_download} avg
        %{speed_download} " "$URL" >> results &
    curl -o /dev/null -m $duration -s -w "bytes %{size_download} avg
        %{speed_download} " "$URL" >> results &
    curl -o /dev/null -m $duration -s -w "bytes %{size_download} avg
        %{speed_download} " "$URL" >> results &
    [ "$delay" != "" ] && sleep $delay
    let count=$count+10
done
wait
done
echo "done"
```

8.4.1 Prvo breme

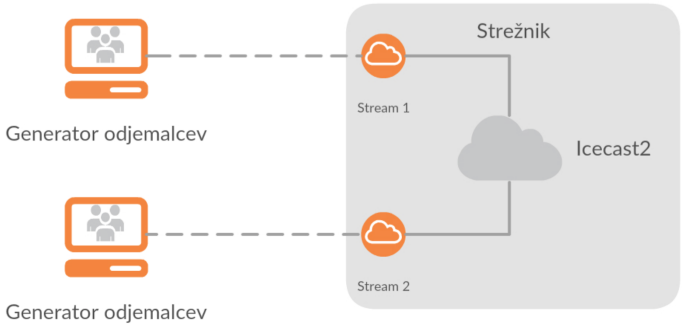
Pri prvem bremenu sva pognala samo eno skripto, ki je vse odjemalce usmerila na isti 'stream'. Vsakih 10 sekund je skripta ustvarila 10 novih odjemalcev in jih povezala na 'stream'. Število odjemalcev sva povečevala, dokler ni zmanjkalo pomnilnika (8GB), in prišla do številke 6000. Čas izvajanja je bil 1h 40min. Ko je zmanjkalo pomnilnika je sistem preprosto opozoril, da je temu tako in naprej deloval nemoteno, če pa je skripta še naprej ustvarjala nove procese, pa je prišlo do popolne zrušitve strežnika, tako da se do njega ni bilo več moč povezati - potreben je bil 'reboot' preko vmesnika ki ga ponuja DigitalOcean [59]. Strukturo prvega bremena lahko vidite na sliki 8.1.

8.4.2 Drugo breme

Pri drugem bremenu sva pognala 2 različni skripti iz dveh različnih računalnikov, ki sta se povezala na različna 'streama', ki pa jih je serviral isti strežnik. Skupno število odjemalcev je bilo seveda enako kot pri prvem bremenu, na vsak 'stream' se je povezalo 3000 odjemalcev. Obe skripti sta število odjemalcev povečevali za 10 vsakih 10 sekund, tako kot pri prvem bremenu, zato se je čas izvajanja prepolovil. Strukturo prvega bremena lahko vidite na sliki 8.2.



Slika 8.1: Struktura testiranja pri prvem bremenu.



Slika 8.2: Struktura testiranja pri drugem bremenu.

8.4.3 Tretje breme

Pri tretjem bremenu sva ustvarila 100 'streamov', pri čemer je bilo na vsak 'stream' povezanih 90 odjemalcev, torej je bilo skupno število odjemalcev 9000. Odjemalcev nisva povezovala postopno, vendar se je vseh 9000 odjemalcev na svoje pripadajoče 'streame' povezalo naenkrat.

8.4.4 Četrto breme

Pri tem bremenu sva pognala skripto, ki je takoj ustvarila veliko število odjemalcev (10.000 in 12.000) in jih povezalo v enem primeru na dva 'streama', v drugem primeru pa na isti 'stream'.

8.5 Metrike

Merila sva obremenjenost centralno procesne enote v odvisnosti od časa oz. števila odjemalcev, merila sva tudi zasedenost pomnilnika RAM, ter s poslušanjem 'streamov' poizkušala preveriti, kako dobra je uporabniška izkušnja (preverjala sva, če je 'stream' neprekinjen).

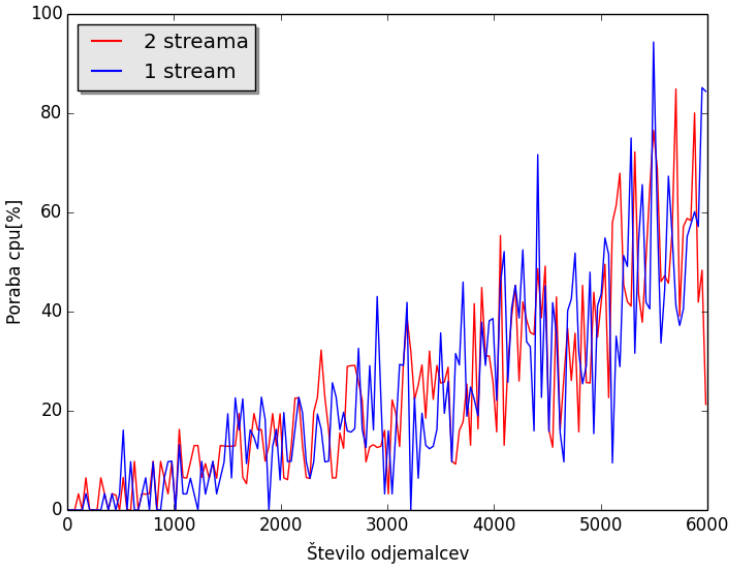
8.6 Meritve

Za meritve sva uporabila skripto, ki je vsakih 5 sekund zajemala podatke o obremenjenosti procesorja. Podatke sva nato obdelala in predstavila na ustreznih grafih. Izvorna koda skripte :

```
#!/bin/bash

# default flags
nflag="1"

# read flags
while getopts "n:" opt ; do
    case "${opt}" in
        n)
            nflag="${OPTARG}"
            ;;
    esac
done

while : ; do
    # timestamp
    ts=$( date +"%d/%m/%Y %T" )

    # CPU usage
    result=$( top -bn 1 | grep 'icecast2' )

    # display text
    echo "${ts} ${result}"

    # sleep for N seconds
    sleep "${nflag}"
done
```

8.6.1 Postopno povečevanje odjemalcev

Ugotovila sva, da se obremenjenost procesorja približno linearno povečuje s številom odjemalcev. Merila sva torej obremenjenost centralno procesne enote če imamo samo en 'stream' in če imamo dva streama. Rezultati so združeni in jih lahko vidimo na sliki 8.3. Eksperiment je bil ponovljen dvakrat, rezultati na grafu so povprečeni. Prvi eksperiment se je izvajal v soboto, 30.4.2016 od 14.00 naprej, drugi pa je bil izveden v petek, 13.5.2016 od 22.00 naprej. Iz rezultatov je vidno, da je pri dveh 'streamih' CPU malce bolj obremenjen kot zgolj pri enem. Zasedenost pomnilnika RAM je tako pri prvem kot pri drugem poizkusu skorajda ostala enaka, zasedenost je sicer naraščala, vendar za zanemarljive vrednosti. Uporabniška izkušnja je bila pri obeh eksperimentih ves čas nemotena.



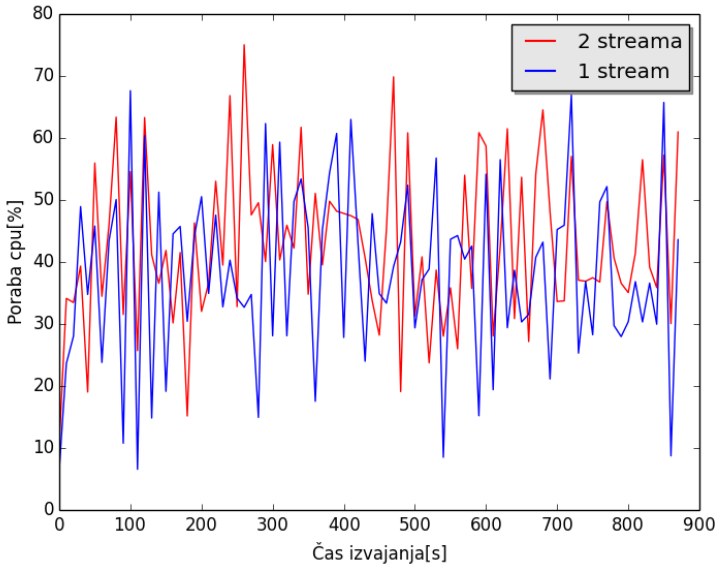
Slika 8.3: Primerjava obremenjenosti CPU pri enem 'streamu' ter pri dveh 'streamih'.

8.6.2 Obremenjenost CPU pri konstantnem številu odjemalcev

Merjenje pri enem in dveh 'streamih' in 6000 odjemalcih

Za to meritev sva vzela primer, ko na začetku povežemo veliko odjemalcev naenkrat (v prvem primeru 6000) na en 'stream' ter potem še po 3.000 odjemalcev na dva različna 'streama', da vidimo kakšna je razlika v obremenjenosti CPU-ja v teh primerih. Pognala sva skripto ter 15 minut izvajala meritev obremenjenosti CPU-ja. Rezultati so vidni na sliki 8.4.

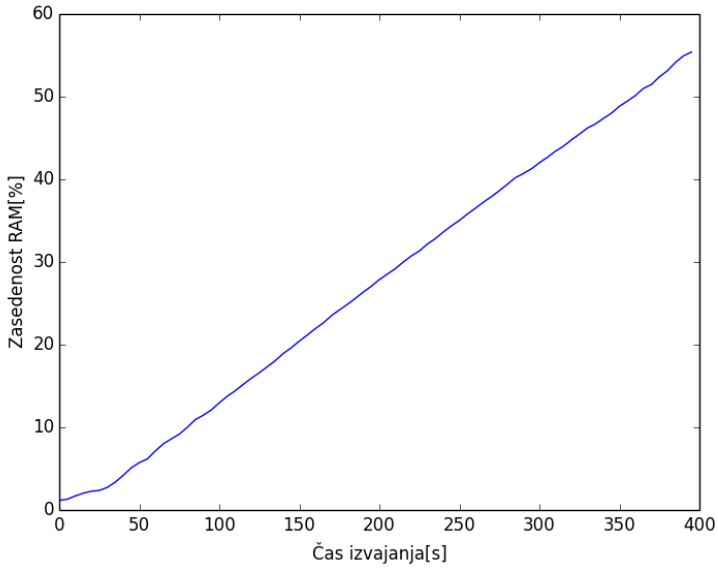
Vidimo, da se rezultati pri enem ali dveh 'streamih' ne razlikujejo preveč. Pri večjem številu odjemalcev na enem 'streamu' je mogoče malce več nihanj, kot pri dveh 'streamih'. Pri povprečni porabi CPU-ja razlike skorajda ni. Pri enem 'streamu' je namreč povprečna poraba v času 900 sekund : 31%, pri dveh 'streamih' pa 33%. Iz tega sledi, da bi pri več streamih in odjemalcih, porazdeljenih na različne streame bila ta razlika tudi večja, kar je potrjeno pri naslednji meritvi. Uporabniška izkušnja je bila pri obeh eksperimentih nemotena, zasedenost pomnilnika RAM je ostala na zelo majhni vrednosti. Eksperimenti so se izvajali v petek, 13.5.2016 od 7.00 naprej, izvedeni so bili trikrat. Rezultati na sliki 8.4 so povprečeni.



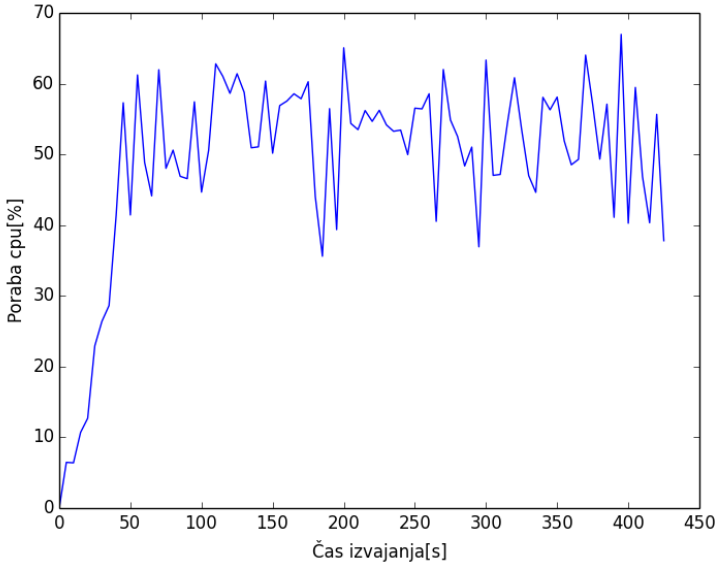
Slika 8.4: Primerjava obremenjenosti CPU med enim 'streamom' in dvema 'streamoma' ter konstantnem številu odjemalcev (6000).

Merjenje pri 100 'streamih' z 9000 odjemalci

Za to meritev sva ustvarila 100 različnih 'streamov', ki so servirali različne datoteke, na vsak 'stream' pa se je povezalo 90 odjemalcev. Ravno zaradi ogromnega števila 'streamov' je bila ta meritev zelo zanimiva s pomnilniškega vidika, saj je zasedenost pomnilnika RAM naraščala hitro in linearno, kar je razvidno iz slike 8.5. Ko je zasedenost pomnilnika dosegla v povprečju 54,3%, je operacijski sistem preprosto ubil proces, ki je serviral 'streame', ker je zasedal preveč pomnilniškega prostora. Strežnik je naprej potem deloval nemoteno. Zasedenost centralno procesne enote je bila dokaj konstantna in je v povprečju znašala 49,1%. Gibanje zasedenosti CPU lahko vidimo na sliki 8.6. Eksperiment se je izvajal v četrtek, 12.5.2016 od 10.00 naprej, s trikratno ponovitvijo. Rezultati na slikah 8.5 in 8.6 so povprečeni. Uporabniška izkušnja je bila pri tej meritvi nezadostna, saj je bil 'stream' konstantno prekinjan.



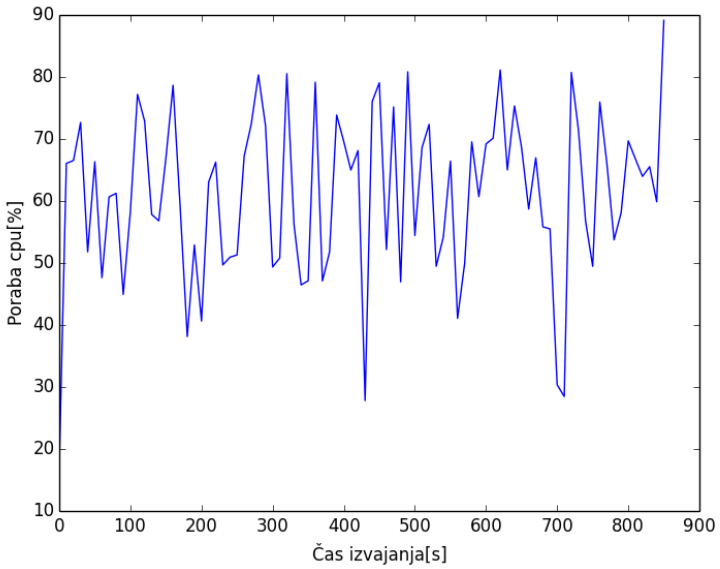
Slika 8.5: Zasedenost pomnilnika RAM pri 100 'streamih' in 9000 odjemalcih.



Slika 8.6: Obremenjenost CPU pri 100 'streamih' in 9000 odjemalcih.

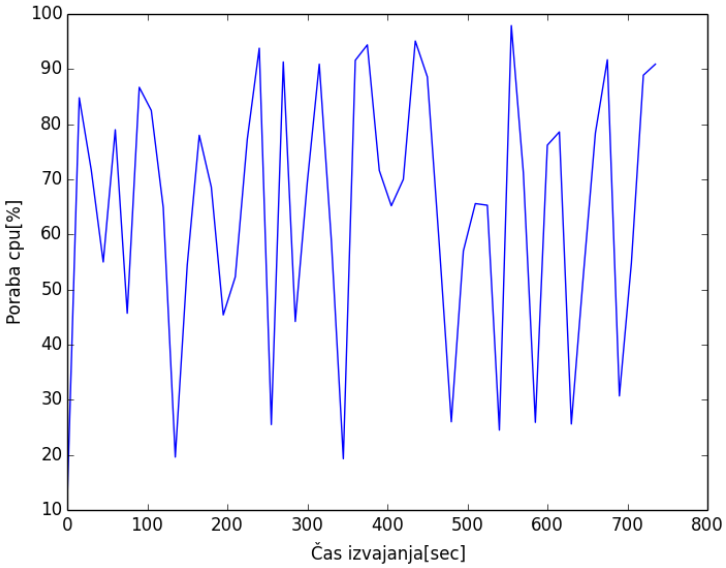
Merjenje pri enem 'streamu' z 10.000 ter 12.000 odjemalci

V nadaljevanju sva naredila še eno meritev in spremenila breme ter pognala skripto za merjenje obremenjenosti, da bi videla pri koliko odjemalcih strežnik dejansko počepne. Meritev sva izvedla z 10 tisoč odjemalci ter ponovila z 12 tisoč odjemalci, ki sva jih poganjala 15 minut ter poslušala 'stream'. Po nekaj sekundah je 'stream' že začel prekinjati, posamezne prekinitve so se pojavljale občasno. Rezultati so vidni na slikah 8.7 in 8.8.



Slika 8.7: Meritev obremenjenost CPU pri enem streamu in 10000 odjemalcih.

Povprečna poraba je zaradi načina delovanja procesorja in nihanj sicer 48,5% v obdobju petnajstih minut za deset tisoč odjemalcev (slika 8.7). Potem pa sva meritev ponovila še za 12000 odjemalcev. Stream je začel prav tako kot pri prejšnji meritvi prekinjati, vendar so bile prekinitve bolj pogoste. Vidimo, da je obremenjenost CPU-ja na sliki 8.8 večja kot na sliki 8.7. Popvrečna obremenjenost je narasla na 65%. Tudi pri teh dveh meritvah se zasedenost pomnilnika RAM ni veliko spreminjala in je ostala na zelo majhni vrednosti. Zaznala pa sva motnje pri uporabniški izkušnji, ki je bila nezadostna že pri 10 tisoč odjemalcih. Eksperimenti so bili za meritev 8.7 ponovljeni trikrat in sicer prvič 9.5.2016 od 11.00 naprej, naslednjič pa 13.5.2016 od 9.00 naprej. Rezultati na grafu so povprečeni. Meritev na sliki 8.8 sva izvedla zgolj enkrat.



Slika 8.8: Meritev obremenjenost CPU pri enem 'streamu' in 12000 odjemalcih.

8.7 Zaključek

Uspelo nama je postaviti sistem, ki deluje zanesljivo, dokler ima na voljo dovolj računalniških virov. Pri velikem številu odjemalcev je problem predvsem zasedenost centralno procesne enote, medtem ko je pri povečevanju števila 'streamov' problem predvsem zasedenost pomnilnika RAM. Zasedenost CPU torej raste približno linearno s povečevanjem števila odjemalcev, zasedenost pomnilnika RAM pa raste linearno s povečevanjem števila 'streamov'. Pri tem je treba poudariti, da morajo 'streami' servirati različne in nikakor ne iste datoteke, če želimo, da ta teza velja. Če želimo uporabnikom zagotoviti dobro uporabniško izkušnjo, lahko pri specifikacijah, kot jih je imel ta strežnik, en 'stream' serviramo največ 9500 odjemalcem.

Poglavje 9

Analiza zmogljivosti oblačnih storitev za hranjenje podatkov

Aleksandar Gogov, Staš Hvala, Matic Repše

9.1 Predstavitev ideje

Lokalni prostor izgublja na pomembnosti, ker se uporabniki vedno pogosteje poslužujejo oblačnega hranjenja podatkov. Osnovne oz. minimalne storitve so po večini brezplačne, možno pa je tudi nadgraditi prostor in uporabljati plačljive storitve, ki uporabniku omogočajo uporabo dodatnih funkcionalnosti. Namen naše raziskovalne naloge je bil testirati najpopularnejše ponudnike (Google Cloud Storage, Amazon S3, Microsoft Azure) in bralcu predstaviti rezultate in ugotovitve testiranj. Naša želja je tudi odkriti, kdo ima zanesljivejšo ter zmogljivejšo performanco oblaka.

9.2 Rešitev problema

Testiranja smo se najprej lotili z CloudHarmony benchmark-om [63], da lahko postavimo začetne hipoteze in dobimo okvirno sliko rezultatov. Ker pa nam je konec koncev najpomembnejša neposredna izkušnja uporabnika, smo se odločili, da preverimo kakovost storitev tudi z lastnimi testi (glej poglavje 9.4) in tako pridemo do pristnejših rezultatov vsakdanje uporabe. Prav tako nam dovoljujejo tudi večjo manipulacijo z bremenom (tip, velikost, število).

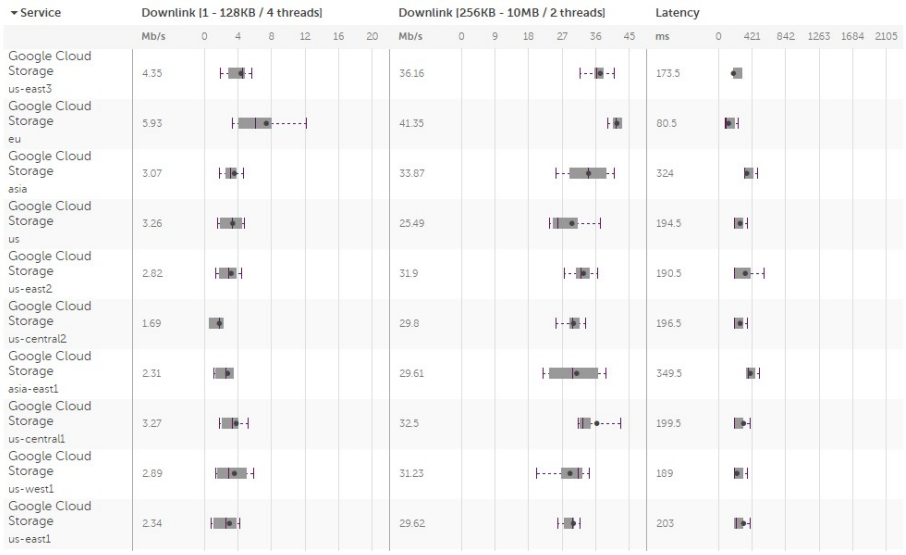
9.3 CloudHarmony Benchmark

CloudHarmony zagotavlja objektivno, nepristransko in zanesljivo analizo uspešnosti za primerjavo storitev računalništva v oblaku. Testirali smo Google Cloud Storage, Amazon S3 in Microsoft Azure v različnih regijah.
Osredotočili smo se na dva različna testa:

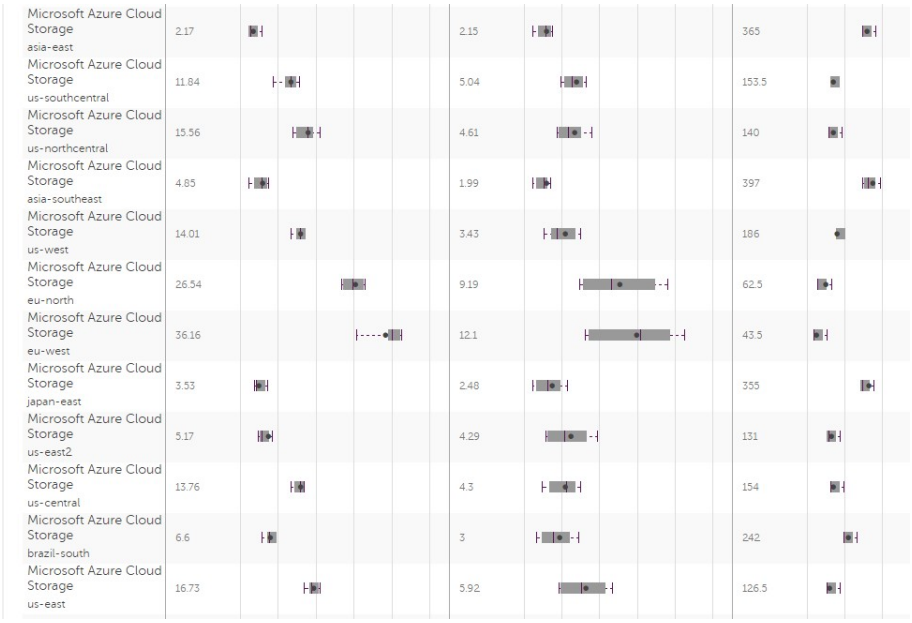
- Downlink
  - Downlink [256KB - 10MB / 2 threads],
  - Downlink [1 - 128KB / 4 threads],
- Latency.

Na vsako uro smo pognali 6 testov na HTTP protokolu, ker je hitrejši brez dodatnih varnostnih rokovanj. Kot je bilo pričakovano glede na naš geografski položaj, naši rezultati kažejo najboljšo odzivnost in hitrost prenosa pri evropskih strežnikih. Naše meritve so bile opravljene na 33 različnih točkah, ki so bile porazdeljene na treh kontinentah (US, EU, ASIA).

Slika 9.1: Amazon S3 CloudHarmony benchmark rezultat.



Slika 9.2: Google Cloud Storage CloudHarmony benchmark rezultat.



Slika 9.3: Microsoft Azure CloudHarmony benchmark rezultat.

Pri Amazon S3 je iz škatel z brki (angl. *box plot*) razvidno, da je downlink najhitrejši na eu-central strežniku. Ista zgodba je tudi pri latenci, najnižja je v centralni Evropi (glej sliko 9.1).

Google Cloud Storage ima v Evropi samo en strežnik, in sicer na zahodu. Tudi ta je najhitrejši po odzivnosti in prenosu podatkov glede na ostale kontinente (glej sliko 9.2).

Zadnji je še Microsoft Azure, pri katerem ponovno prevladuje evropski strežnik, je bilo pa velika podobnost med Severno in Zahodno Evropo. Po večjem številu testov je prevladal zahod (glej sliko 9.3).

Za vse tri ponudnike je očitno, da imajo najhitrejše in najodzivnejše strežnike v Evropi. Pričakovali smo, da pri vseh treh do tega ne bo prišlo, ker se ponavadi pretakajo informacije po celemu svetu, a očitno ima geografska razdalja večji vpliv kot smo mislili. Ob kratkem pregledu na internetu smo ugotovili [67], da je odvisno kje je postavljen naš ISP (angl. *Internet service provider*) oz. v tem primeru CloudHarmony strežnik, ki izvaja teste. Očitno je CloudHarmony testni strežnik lociran nekje v Evropi in ker smo geografsko tudi mi najbližji ponudnikom v Evropi, smo se dokončno odločili za zgoraj navedene lokacije.

Ob kategoričnem pregledu rezultatov je Microsoft Azure z najkrajšo latenco in največjim downlinkom zmagovalec na izbranem benchmarku. Drugo mesto je zasedel Google. Njegov downlink se zelo približa Microsoftovemu, a je latenca podvojena. Amazonova latenca je malo večja od Azure-a, vendar je downlink približno dvakrat manjši kot pri ostalih dveh ponudnikih.

Uplinka in DNS-ja kljub možnosti izbire ni bilo mogoče testirati, saj očitno CloudHarmony pri teh storitvah tega ne podpira. Predpostavili pa smo, da so hitrosti za uplink glede na lokacijo podobne downlinku.

9.4 Lastni testi

Naši lastni testi so usmerjeni v čas upload-a in download-a posamezne storitve. Opravili smo več različnih testov, vsakega s svojim določenim tipom bremena, kjer smo za vsak test ločili bremena glede na posamezno lastnost (velikost datoteke, tip datoteke, število datotek, itd.)
Testirali smo torej upload/download čas:

- posamezne datoteke,
- večih datotek,
- datotek različnih velikosti,
- datoteke pred vstavitvijo in po vstavitvi v predpomnilniku na oblaku,
- ...

Za dostop in testiranje storitev smo uporabljali unix-ov terminal, v katerem se avtomatizirano izvaja *bash* skripta.
Za posamezno storitev smo uporabili naslednja bash orodja:

- `azure cli` za Microsoft Azure [64],
- `gsutil` za Google Cloud Storage [65],
- `aws cli` za Amazon S3 [66].

9.4.1 Breme

Z določitvijo bremena smo testirali latenco in odzivnost ponudnika - za sekvenčno ter paralelno prenašanje. Glede na pridobljene rezultate smo poskušali ugotoviti ali uporabljajo *cache* (za majhne datoteke) in kompresijo (za določen tip datoteke, velike datoteke, itd.). Poskušali smo se izogniti zunanjim vplivom in pridobiti čimbolj pristne rezultate. Meritve smo zato izvedli večkrat - vsako uro, tekom enega dneva - a smo bili s številom testiranj omejeni, ker se poslužujemo brezplačne verzije pri vseh ponudnikih. Testiranje smo izvajali na sledečih bremenih:

- Posamezne datoteke:
  - 1kB,
  - 10kB,
  - 100kB,
  - 1MB,
  - 10MB,
  - 100MB,
  - 500MB,
  - 1GB.
- Več datotek:
  - 1kB x 2, 5, 10, 50, 100,
  - 10kB x 2, 5, 10, 50, 100,
  - 100kB x 2, 5, 10, 50, 100,
  - 1MB x 2, 5, 10, 50, 100,
  - 10MB x 2, 5, 10, 50, 100,
  - 100MB x 2, 5, 10,
  - 500MB x 2, 5,
  - 1GB x 2.

Testne datoteke so končnice *.txt*, napolnjene z naključnimi podatki. Za testiranje download/upload pri načinu "več datotek" potrebujemo pripravljeno okolje in sicer smo zgradili drevesno strukturo datotek. V korenskem imeniku se nahajajo sinovi, ki so razdeljeni po velikosti datotek, ki jih vsebujejo. Znotraj so datoteke istih velikosti razdeljene še po kvantiteti, torej vsak imenik vsebuje število datotek, ki se bo paralelno preneslo.
Za ustvarjanje in kopiranje datotek smo uporabili skripto 9.1.

```
#!/bin/bash

SIZE=$1
NUMBER=$2
PREFIX=$3
DESTINATION=$4

# Ustvarjanje tekstovne datoteke poljubne velikosti
base64 /dev/urandom | head -c $SIZE > $PREFIX.txt

# Kopiranje datoteke poljubno krat
for file in 'seq 1 $NUMBER'; do
    filename=${PREFIX}_${file}
    cp $PREFIX.txt $DESTINATION/$filename.txt
done

# Odstranitev originalne datoteke
rm $PREFIX.txt
```

Listing 9.1: Skripta za ustvarjanje in kopiranje datotek.

9.4.2 Samodejno izvajanje testov

Ker se izogibamo ročnemu poganjanju in beleženju rezultatov, smo napisali skripto 9.2, ki bo to opravljala namesto nas. Skripte smo poganjali na *eduroam* omrežju na serverju *XeonPhi* (z dovoljenjem as. uni. dipl. ing. Davor Sluga), saj server teče 24/7, sploh pa bo zaradi 24-ih jeder in 64GB RAM-a prenašanje datotek nemoteno. Obremenitev omrežja na žalost nismo mogli preveriti, ker na strežniku nimamo `sudo` (angl. kratica za *superuser do*) pravic, da bi nameščali dodatna orodja, zato smo teste pognali večkrat in z izračunom povprečja ter standardne deviacije zmanjšali vpliv iregularnih rezultatov. V primeru plačljivih verzij, bi za zanesljivejše rezultate testirali še na drugih omrežjih. Pri brezplačnemu načinu pa to ni možno, ker nam omejujejo število operacij, več računov pa ne moremo ustvariti, ker za registracijo zahtevajo številko kreditne kartice.

```
#!/bin/bash

printf "%s\n" "arguments:"
printf "%s\n" "-m ----> multiple files"
printf "%s\n" "-t ----> followed by int time in hours \
        (execute every H hours)"
printf "%s\n" "-n ----> execute N times"

SOURCE=single_files
H=1
```

```
N=24
S3Bucket=s3://zzrs2016mf/
GSbucket=gs://zzrs2016/
iter=0

while [ "$#" -gt 0 ]
do
   case $1 in
   -m)
      SOURCE=multiple_files
      shift 1
      continue
      ;;
    -t)
        H=$2
        shift 2
        continue
        ;;
    -n)
        N=$2
        shift 2
        continue
        ;;
   *)
      ;;
   esac
   shift
done

for iter in $(seq 1 $H);
do
    for map in $HOME/$SOURCE/*
    do
      for file in $map/*
      do
        ./amazonS3.sh -s $file -d $S3Bucket # upload
        ./amazonS3.sh -D # download
        aws s3 rm --recursive $S3Bucket

        ./azure.sh -s $file
        ./azure.sh -D
        azure storage blob delete zzrs zzrs

        ./google.sh -s $file -d $GSbucket # upload
        ./google.sh -D
        gsutil rm $GSbucket**

        rm -rf ~/downloads/
      done
    done
```

```
  sleep $Hh
done
```

Listing 9.2: Skripta za samodejno izvajanje testov.

9.4.3 Amazon S3

Najprej smo na Amazonu ustvarili račun z osebnimi podatki in prestali avtorizacijo. Izbrali smo brezplačni paket, ki traja 12 mesecev. Ponudnik obljublja skalabilen in zanesljiv sistem z nizko latenco ter:

- 5GB prostora,
- 20.000 *GET* zahtev,
- 2.000 *PUT* zahtev.

Sledila je kreacija *bucketa* (odlagališče), dodelitev pravic (trenutno *public*) in zapis *credentionals* v `.bashrc` kot okoljske spremenljivke (`AWS_ACCESS_KEY_ID` in
`AWS_SECRET_ACCESS_KEY`).
S pomočjo *AWS CLI* (angl. *Command Line Interface*) ukaza smo napisali skripto 9.3, ki obdela argumente in jih posreduje aws-ju, medtem pa meri čas izvajanja uploada ali downloada. Za prenos in pošiljanje se uporablja bash komande `cp`, kjer samo zamenjujemo izvor (angl. *source*) in cilj (angl. *destination*) za željeno operacijo (download/upload). S stikalom `-recursive` prenašamo celotne imenike, avtor *API*-ja (angl. *Application programming interface*) pa nam zagotavlja, da se prenašanje izvaja paralelno.

```
#!/bin/bash

printf "%s\n" "arguments:"
printf "%s\n" "-s ----> source file (path)"
printf "%s\n" "-d ----> destination file (path)"
printf "%s\n" "-D ----> download (upload is the default\
        action)"

DESTINATION=s3://zzrs2016/test
SOURCE=upload_test
UPLOAD=upload
HOME=~/downloads/

while [ "$#" -gt 0 ]
do
    case $1 in
    -s)
       SOURCE=$2
       shift 2
```

```
      continue
      ;;
    -d)
      DESTINATION=$2
      shift 2
      continue
      ;;
    -D)
      UPLOAD=download
      SOURCE=s3://zzrs2016mf/
      ;;
    *)
      ;;
    esac
    shift
done

  timestamp=$( date +"at: %d-%m-%Y %H:%M:%S" )
  START=`date +%s%N`

  if [[ $UPLOAD == "upload" ]]
  then
    aws s3 cp $SOURCE $DESTINATION --recursive --quiet
  else
    aws s3 sync $SOURCE $HOME --quiet
  fi

  END=`date +%s%N`
  ELAPSED=`echo "scale=5; \
          $(($END - $START)) / 1000000000" | bc`
  echo "File: $SOURCE, Started $UPLOAD" \
       "$timestamp, lasted: [$ELAPSED] seconds." \
        >> "rezultatiAmazonS3.txt"
```

Listing 9.3: Skripta za testiranje Amazon S3 storitve.

9.4.4 Google Cloud Storage

Pri GCS je brezplačna ponudba malo drugačna. Ob registraciji so nam dali **300$** virtualnega denarja, ki ga lahko mirne vesti zapravljamo na njihovi strani. Cene storitev, ki smo jih uporabljali, so sledeče:

- **$0.10** na 10.000 operacij za *GET* ali *PUT* ukaze,
- Standard storage (tip bucketa, najnižja latenca) **$0.026** na GB podatkov na mesec,
- *DELETE* je brezplačen.

Lokalni terminal smo z bucketom povezali preko ukaza `gsutil config`. Ta nam vrne *url* (angl. *Uniform Resource Locator*) naslov na katerem dobimo aktivacijsko kodo, ki jo nato vpišemo v terminal.

Skripta za prenašanje datotek je sila podobna Amazonovi, saj za prenos tudi uporablja ukaz `cp`. S kombinacijo stikal `-r` (*recursive*) prenašamo imenike, z `-m` pa paralelno prenašanje. Primer uporabe: `gsutil -m cp -r dir gs://zzrs`. Skripte zaradi podobnosti ne bomo prilagali.

9.4.5 Microsoft Azure

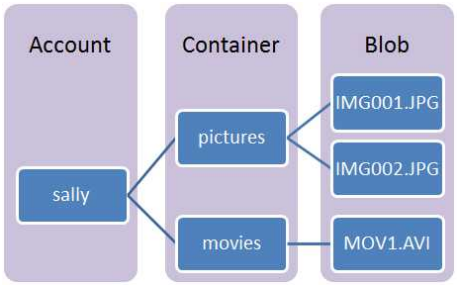
Azure CLI nam prav tako ponuja širok spekter ukazov s katerimi delamo na Azure platformi, kjer je velik poudarek na operiranju z *Blobi*. Za dostopanje do storitve smo najprej kreirali uporabniški račun in ob registraciji dobili **200$** virtualnega denarja, ki se troši s hranjenjem in prenašanjem datotek (blobov). Cene storitev, ki smo jih uporabljali, so sledeče:

- za *Standard storage (Block blobs)* na mesec za prvi TB podatkov hranjenje stane **$0.024** na GB podatkov,
- na 100.000 transakcij (branje/pisanje) je **$0.0036** tarife,
- brisanje je brezplačno.

Že takoj opazimo, da je ta storitev cenovno najugodnejša in je ne glede na odzivnost storitve najboljša izbira za uporabnika, ki hoče koristiti brezplačno verzijo.

Po registraciji smo se preko terminala povezali na *container* tako, da smo se najprej indentificirali z vključitvijo `AZURE_STORAGE_ACCOUNT` in `AZURE_STORAGE_ACCESS_KEY` v datoteko `.bashrc`. Za sam prenos datotek nam je spet ponujen intuitiven *API*, a se od ostalih dveh ponudnikov ločuje z uporabo dveh različnih funkcij za upload in download. Primer uporabe:
`azure storage blob upload source_to_upload container_name blob_name`.
`azure storage blob download container_name blob_name destination_folder`.

Značilnost tega ponudnika je, da se podatki prenašajo v *blob-ih*. Azure blob je storitev za shranjevanje velike količine nestrukturiranih podatkov, kot so besedilo ali binarni podatki, do katerih lahko dostopamo prek HTTP ali HTTPS. Na sliki 9.4 je razvidno, kakšno arhitekturo uporablja Azure.



Slika 9.4: Primer Azure Blob storitve.

Obstajajo tri vrste blob-ov: *block blobs*, *page blobs* in *append blobs*. Block blobs so namenjenji hranjenju tekstovnih in binarnih datotek (npr. dokumenti, medijske datoteke, itd.). Append blobi se uporabljajo za isti namen, razlika je le, da so sestavljeni iz blokov in so optimizirani za hitro dodajanje (uporablja se jih lahko npr. za logiranje). Page blobi so namenjeni za večje datoteke (do 1 TB) in so uporabni, kadar potrebujemo hitre dostope za branje/pisanje. Ker uporabljamo brezplačno verzijo in imamo omejene kredite, uporabljamo Block blobe, saj ne bomo prenašali ogromnih datotek, poleg tega pa so tudi tekstovne narave.

Struktura napisane skripte se je ohranila v primerjavi z ostalima dvema, razlikujejo se le ukazi za prenos datotek. Nažalost pa to orodje in ostale alternative ne ponujajo sočasnega prenašanja večjega števila datotek, zato smo pri Microsoft Azure testirali in primerjali samo rezultate za prenašanje ene datoteke naenkrat.

9.4.6 Primerjava rezultatov

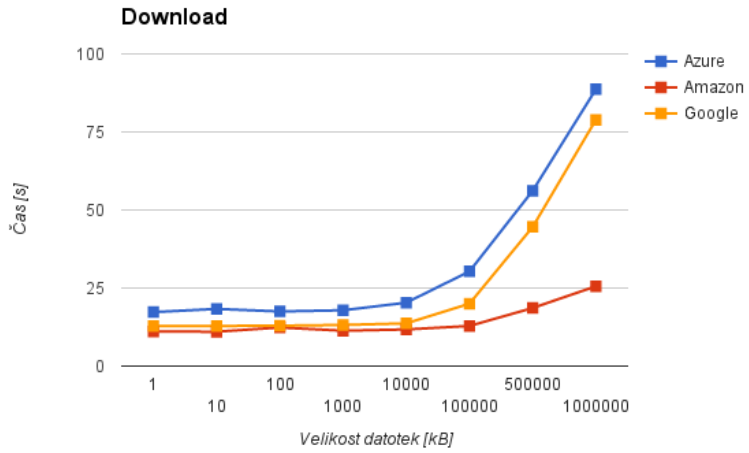
Pri testiranju za eno datoteko naenkrat smo izvedli 24 iteracij za vse 3 ponudnike naenkrat in enournim razmakom med iteracijami. Pri testiranju za več datotek (paralelni prenos) naenkrat pa smo zaradi omejenih resursov izvedli 12 iteracij z dvournim razmakom samo za Google in Amazon, ker Azure ne podpira paralelnega prenosa. Prav tako je vredno omeniti, da smo teste izvajali čez vikend, ker je po analizi lanske 6. skupine [68] takrat obremenitev strežnika Amazon S3 v Evropi najmanjša. Enako smo predpostavili za ostala dva ponudnika.

Za vse iteracije smo izračunali povprečje prenosa, ker pa smo nekajkrat dobili iregularne rezultate, smo vključili v izračun tudi standardno deviacijo.

Testiranje enkratnega prenosa

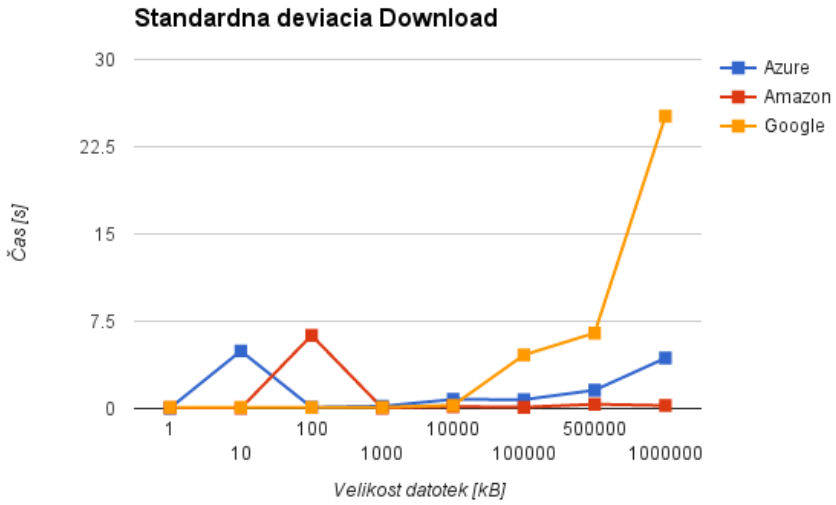
Na sliki 9.5 za download se takoj opazi, da so jasni hitrostni pasovi za vse tri ponudnike, ne glede na velikost datoteke. Najhitrejši je Amazon, drugi Google in tretji Azure. Uvrstitev je ravno obratna kot pri CloudHarmony (glej poglavje 9.3), a težko ugotovimo kje in zakaj je do tega prišlo, ker na njihovi strani ne opisujejo prav posebej kako in kakšne teste izvajajo. Je pa zanimivo, da je uvrstitev skladna z radodarnostjo ponudbe za brezplačno verzijo. Amazon nudi najmanj, a to opravi z največjo hitrostjo, Google ima srednjo drago storitev ter srednjo hitrost, Azure pa ponuja najcenejšo storitev, a se to odraža tudi v hitrosti prenosa.

Pri vseh treh ponudnikih je za downloadiranje vidno, da je čas prenosa do velikosti 10000kB skoraj zanemarljiv in do izraza pride le latenca, ki pa med ponudniki raste v enakem zaporedju kot sam download. Za velike datoteke (nad 100000kB) čas prenosa raste precej linearno, torej nobeden ne uporablja kompresije ali pa paralelnega prenosa po koščkih. Googlova krivulja se po obliki prilagaja Azurovi in za velike datoteke obe strmo naraščata, medtem, ko pri Amazonu narašča z manjšim naklonom in za 1GB veliko datoteko prinaša že do 50 sekund razlike v prenosu.



Slika 9.5: Graf povprečnega downloada ene datoteke.

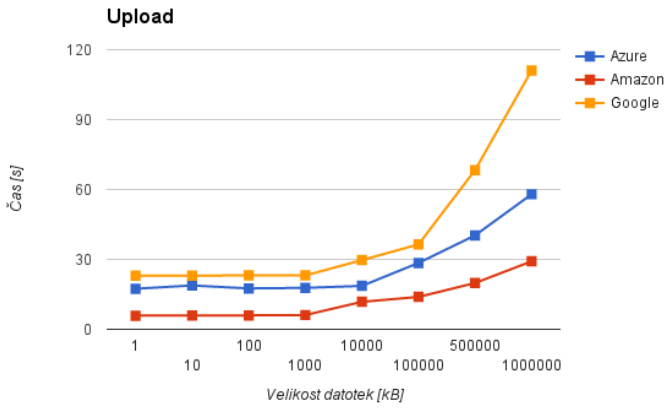
Do odstopanj pri Azure-u in Amazonu (glej sliko 9.6) pride le za manjše datoteke, pri večjih pa se je čas prenosa večinoma držal povprečja. Pri Googleu pa je bilo ravno obratno, za manjše datoteke je bila standardna deviacija izredno majhna, medtem ko je za velike datoteke (nad 10000kB) eksponentno naraščala. To nakazuje na izredno nestabilno storitev, a mogoče smo imeli le smolo in naleteli na obdobje vzdrževalnega dela na serverju.



Slika 9.6: Graf standardne deviacije downloada ene datoteke.

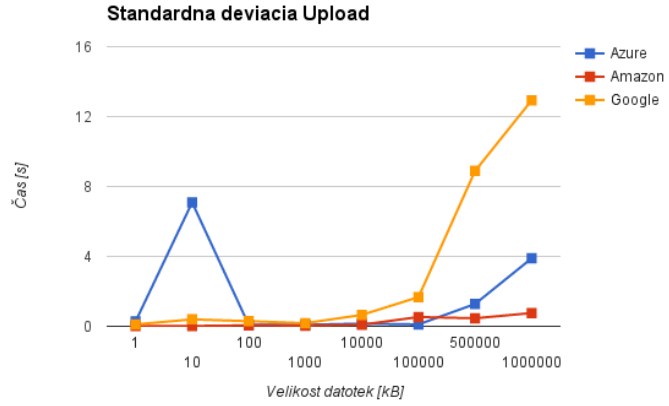
Za upload je s slike 9.7 razvidna drugačna razvrstitev. Na prvem mestu je še vedno Amazon, sta se pa zamenjala Google in Azure. Med ponudniki in njihovimi krivuljami uploada glede na velikost datoteke ni več očitnih podobnosti, pač pa vsaka narašča s svojim naklonom. Prav tako je razlika med ponudniki večja za upload kot za download, torej imata Azure in Google pred sabo še precej optimizacije za nalaganje datotek.

Prag latence za Azure ostaja pri do 10000kB, pri Googleu in Amazonu pa so večje časovne razlike opazne že za datoteke večje od 1000kB. Tudi tukaj je razvidno, da sta hitrost uploada in višina latence povezana, torej nobena storitev ne potrati preveč časa za rokovanje (angl. *handshake*).



Slika 9.7: Graf povprečnega uploada ene datoteke.

Pri standardnem odklonu za upload (glej sliko 9.8) se zgodba ponovi, nekaj manjših razlik je prišlo pri Azureu, Google pa je ponovno variiral za velike datoteke (nad 100000kB). Konsistenca je pri prenašanju lahko izredno pomembna in se je vsekakor splačalo izvesti teste večkrat, ker smo le tako dobili bolj točno oceno ponudnikov. Če bi imeli neomejeno resursov, bi seveda testirali večkrat v enem tednu in mogoče dobili lepše rezultate (manj deviacije), ampak razvrstitev ponudnikov bi verjetno ostala kar ista.

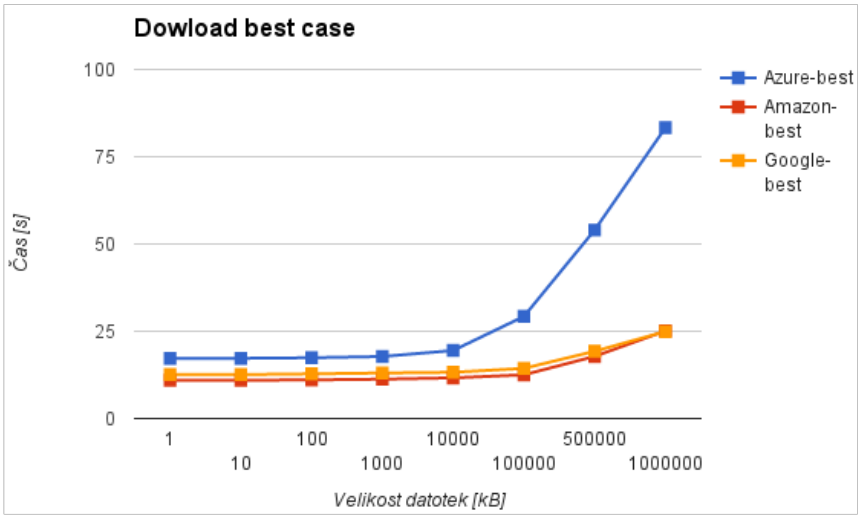


Slika 9.8: Graf standardne deviacije uploada ene datoteke.

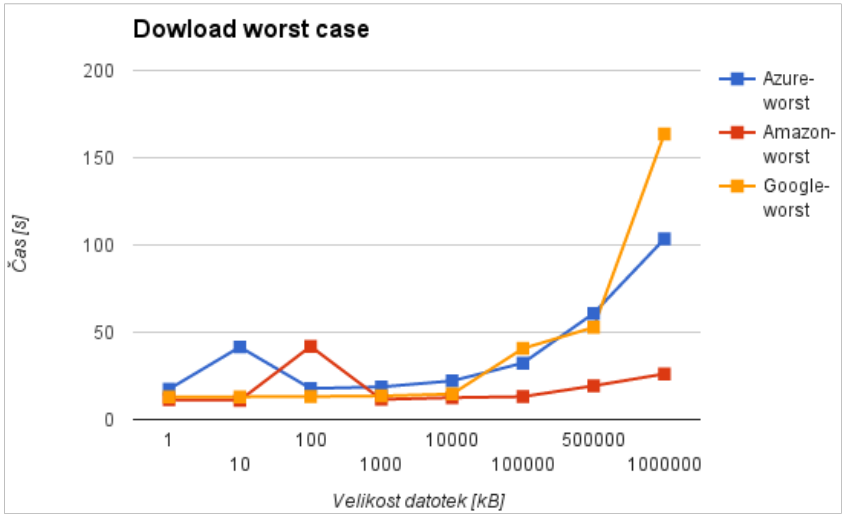
Če pri download operaciji pogledamo najboljše primere testiranja za npr. 1GB datoteke (glej sliko 9.9), lahko opazimo, da je Amazonov praktično enak

kot povprečje, kar je precej presenetljivo, ker sta se za to breme ostala dva uporabnika izkazala precej slabše. Pri Azure je najboljši primer le za nekaj sekund hitrejši od povprečja. Googlov najboljši primer pa se zelo razlikuje od povprečja, in sicer kar za dobrih 50 sekund. To je znova dokaz Googlove nekonsistentnosti, ki se izraža tudi pri najslabšem primeru.

Za najslabše primere download testiranja za 1GB velike datoteke na sliki 9.10 opazimo, da je Amazon znova tik ob povprečju, kar glede na najboljši primer ni presenečenje. Azure je nekoliko slabši, ampak še vseeno ni tako daleč od povprečja (približno 10-15 sekund). Google je znova precej zgrešil povprečje. Ker smo teste opravljali čez vikend (zaradi manj prometa - ugotovitev lanske 6. skupine [68], težko določimo razlog za tako veliko odstopanje. Mogoče so takrat opravljali kakšna vzdrževalna dela na strežniku, ali pa je to posledica kakšnega drugega dogodka na katerega nismo računali. Dodatna možna razlaga je, da Google enostavno ponuja nekonsistenten sistem za svoje brezplačne storitve in je mogoče stanje pri plačljivi verziji drugačno. Standardna deviacija močno potrjuje naš sklep o Googlovi nezanesljivosti.



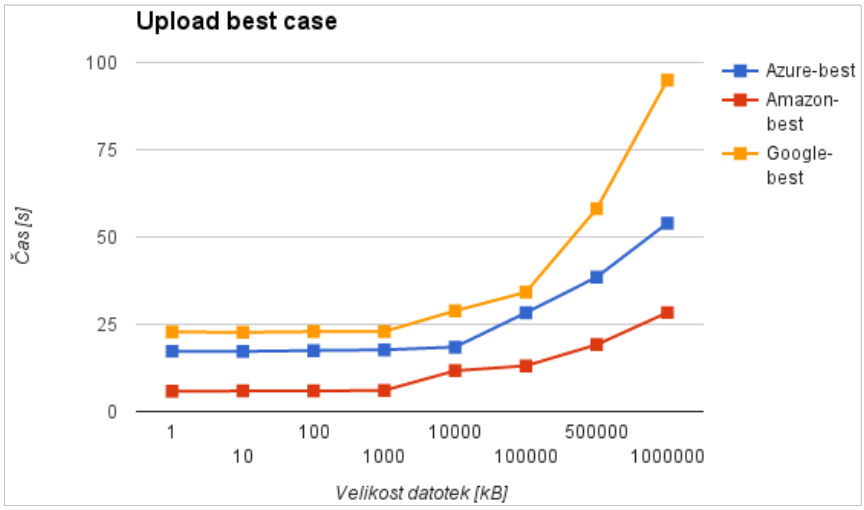
Slika 9.9: Graf najboljših primerov download-a.



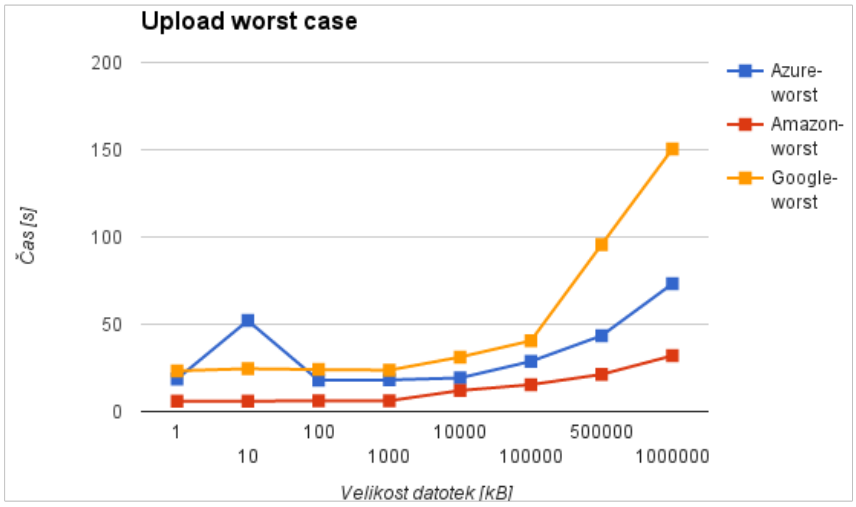
Slika 9.10: Graf najslabših primerov download-a.

Pri najboljših primerih upload operacije (glej sliko 9.11) za datoteke velike 1GB, lahko pri vseh potegnemo enake ugotovitve kot pri najboljših primerih download operacije. Trend se že ves čas ponavlja. Amazon zelo konsistenten, Azure malo manj, Google zelo nekonsistenten.

Za najslabše primere upload operacije (glej sliko 9.12) lahko prav tako opazimo podobnost z našo analizo najslabših primerov download operacije. Tukaj lahko potegnemo končno črto, da Googlova storitev enostavno ne ponuja zanesljivosti, saj je enako kot pri download operaciji tudi tukaj standardna deviacija ogromna v primerjavi z Azure in Amazon. To nam pokažeta tudi najboljši in najslabši čas Googlovega uploada, saj se razlikujeta za dobrih 50 sekund.



Slika 9.11: Graf najboljših primerov upload-a.

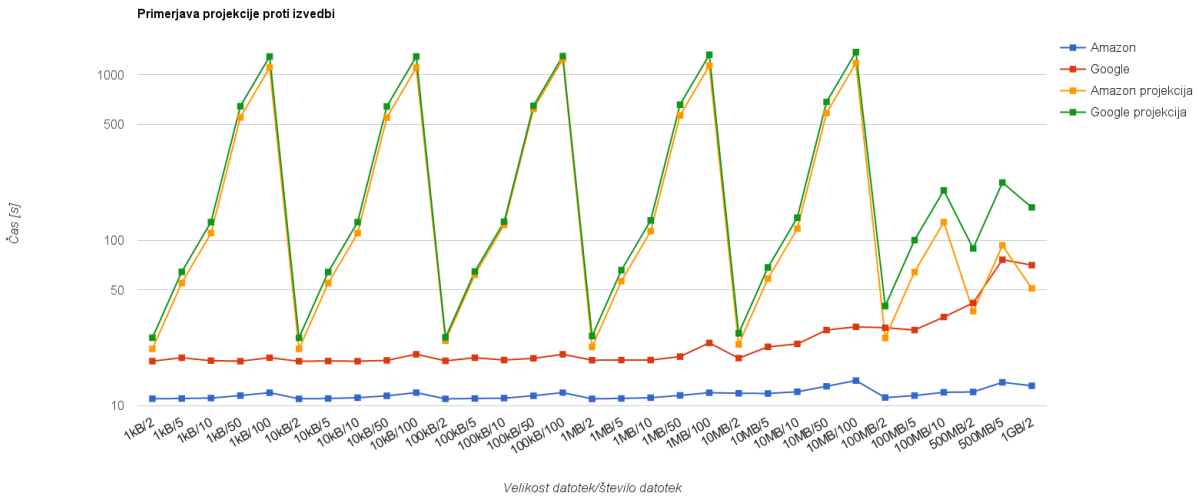


Slika 9.12: Graf najslabših primerov upload-a.

Testiranje paralelnega prenosa

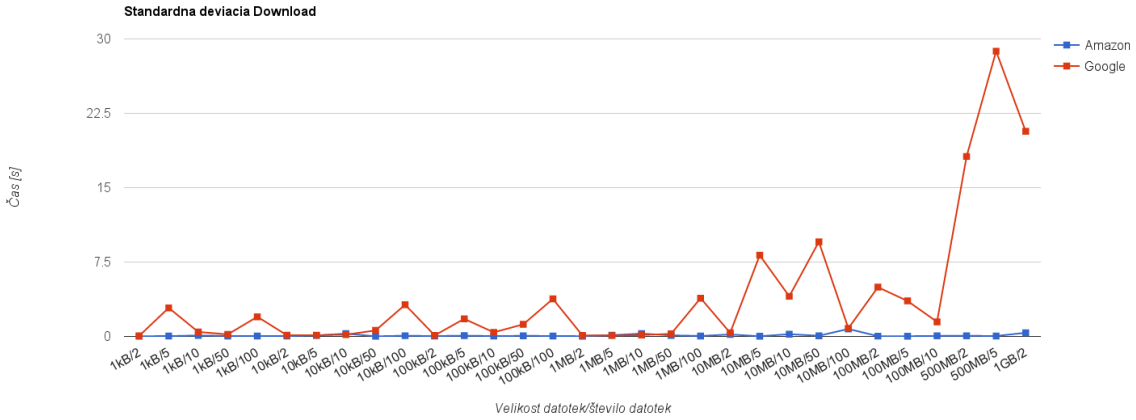
Najprej smo izvedli testiranje na eni datoteki naenkrat, zato smo ob pričetku testiranja paralelnega prenosa postavili hipotezo, da bo Amazon ponovno prevzel vodstvo in smo predvidevali da bo čas prenosa linearno naraščal s številom datotek. Za konec smo naredili še analizo vseh paralelnih testov skupaj, ker nas je zanimalo, koliko bi to vplivalo na časovno skalo. Tudi pri paralelnemu prenosu se je izkazalo, da je Amazon hitrejši od Googla, kar pomeni da smo potrdili prvi del hipoteze.

Pri povprečnem downloadu večih datotek (glej sliko 9.13), nas je Amazon presenetil z skoraj vodoravno krivuljo ne glede na velikost. Tukaj je najverjetneje problem pri *AWS CLI* orodju in ukazu `sync` [69].



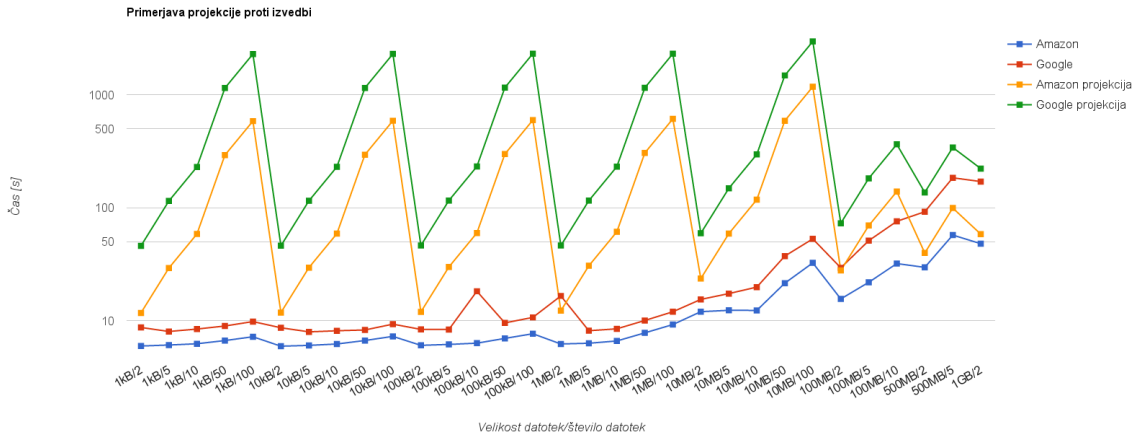
Slika 9.13: Graf povprečnega downloada večih datotek naenkrat in napoved rezultatov.

Pri standardni deviaciji za download večih datotek (glej sliko 9.14), ponovno pride do izraza nekonsistentnost download časov Googla, podobno kot pri testih enkratnega prenosa. Medtem, ko lahko pri Amazonu opazimo kako enovit je bil download čas ukaza `sync` in odstopanja praktično ni.



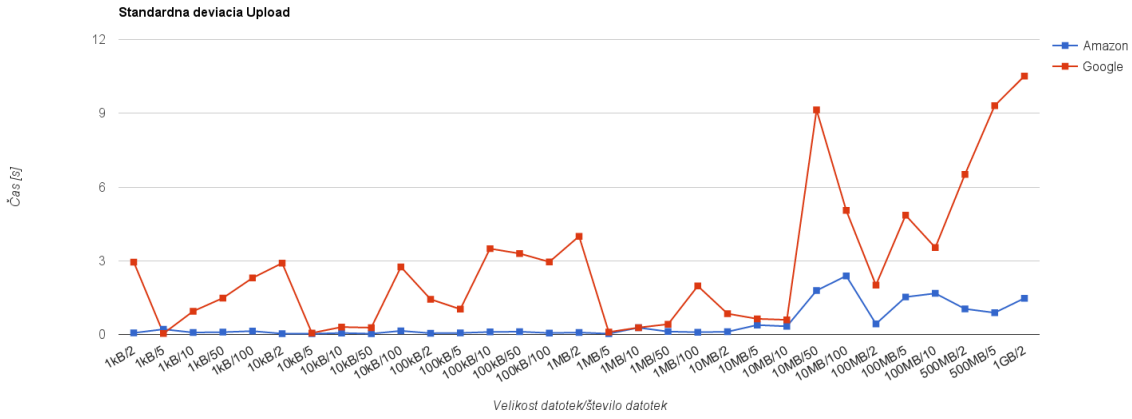
Slika 9.14: Graf standardne deviacije downloada večih datoteke naenkrat.

Pri testiranju drugega dela hipoteze (linearna povezanost med časom in številom datotek) se je izkazalo, da oba ponudnika izkoriščata paralelni prenos. Iz slike 9.15 je razvidno da naša predpostavka o sekvenčnem prenosu ne drži, pravzaprav naši ponudniki svoje delo opravljajo mnogokrat hitreje kot smo napovedali.



Slika 9.15: Graf povprečnega uploada večih datotek naenkrat in napoved rezultatov.

Zgodba se pri standardni deviaciji uploada večih datotek ponovi (glej sliko 9.16), saj spet pride do nihanj pri Googlovi storitvi. Pri Amazonu je do malo večjega odstopanja prišlo le pri večjih datotekah, a le pri večjem številu uploadanih datotek.



Slika 9.16: Graf standardne deviacije uploada večih datoteke naenkrat.

Metrika zmogljivosti

Iz naših grafov in analiz je razvidno, da je ponudnik Microsoft Azure performančno najbolj povprečen ter cenovno najugodnejši, zato smo se odločili, da se za analizo metrike zmogljivosti osredetočimo le na tega ponudnika.

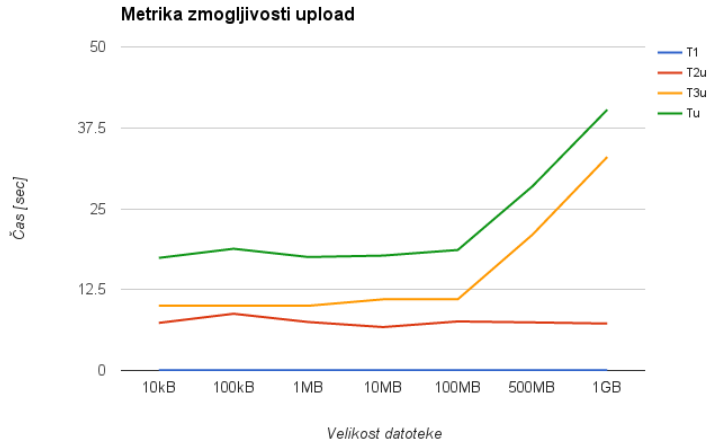
Merili smo čas prenosa datoteke med oblakom ter uporabnikom, ki ga lahko predstavimo z enačbo $T = T_1 + T_2 + T_3$. Sestavljajo jo sledeči parametri:

- T_1: čas dostopa do storitve,
- T_2: čas procesiranja zahteve,
- T_3: čas prenosa datoteke med oblakom in uporabnikom (upload / download).

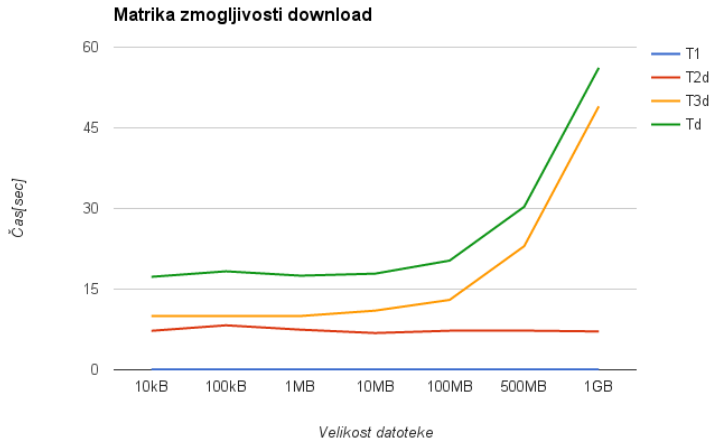
Za testna bremena smo vzeli enake datoteke kot pri ostalih testiranjih in za njih izmerili povprečni čas prenosa v obeh smereh.

T_1 smo v grobem ocenili glede na povprečno latenco, ki smo jo dobili iz Microsoft Azure CloudHarmony benchmark rezultatov (glej sliko 9.3). Rezultat je bila izredno majhna vrednost in sicer $T_1 = 43.5ms$. Bolj bi bilo zanesljivo, če bi čas latence pridobili sami, a Azure zaradi strožje varnosti ne odgovarja na *ICMP* (angl. *Internet Control Message Protocol*) pakete, zato smo tukaj prisiljeni zaupati CloudHarmony-ju.

Pri prenosu datotek nam ponudnik pošlje oceno časa T_3. Ker lahko merimo le skupni čas T_D (download) in T_U (upload), smo morali izračunati čas T_2 po enačbi $T_2 = T - (T_1 - T_3)$. Iz grafov na sliki 9.17 in 9.18 je razvidno, da sta časa T_1 in T_2 bolj ali ne konstantna, medtem, ko čas T_3 pri povečevanju velikost datotek (a šele pri 100MB) eksponentno narašča.



Slika 9.17: Graf metrike zmogljivosti za upload



Slika 9.18: Graf metrike zmogljivosti za download

9.5 Zaključek

Iz vseh rezultatov testiranja je jasno razvidno, da je Amazon vsekakor na prvem mestu, kar se tiče hitrosti. Za uporabnika, ki pri oblačnih storitvah za hranjenje podatkov potrebuje predvsem hitrost in časovno konsistentnost, denar pa ni ovira, je prava izbira Amazon. Če pa nam je vseeno, koliko časa se datoteke prenašajo, ampak nas zanima samo koliko časa lahko storitev uporabljamo brezplačno, potem bi izbrali Azure, ki je najbolj radodaren z brezplačno verzijo.

Vsekakor bi lahko našo raziskavo zelo izboljšali, če bi razširili število ponudnikov (DropBox, Mega, ...) in pognali več raznolikih testov. Zanimivo bi pa bilo poizkusiti tudi plačljive verzije, ker vsi trije ponudniki ponujajo hitrejše opcije prenosa, a seveda z večjo tarifo.

Poglavje 10

Zmogljivostna analiza Raspberry Pi v funkciji spletnega strežnika

Gašper Kolar, Matjaž Mav

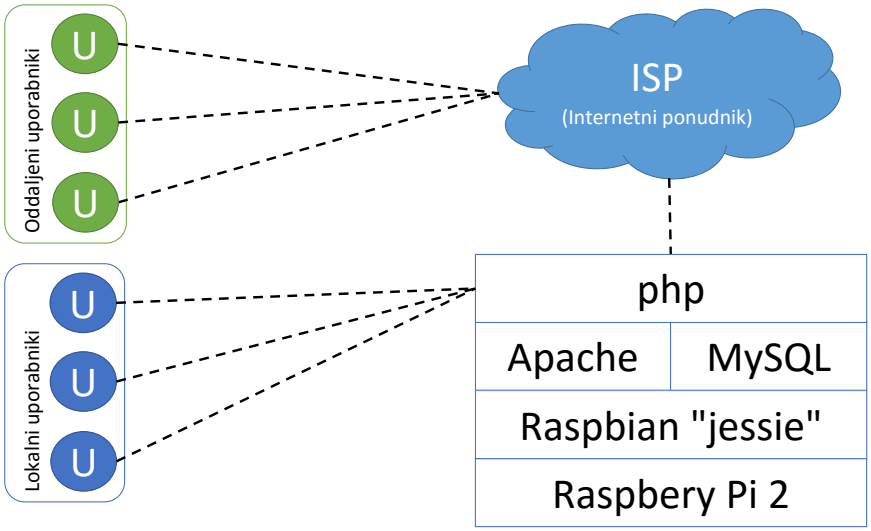
10.1 Predstavitev ideje

Raspberry Pi [58] predstavlja ugodno cenovno alternativo za gostovanje preprostih spletnih strani. V pričujočem prispevku sva ugotavljala kakšne so zmogljivosti preprostega LAMP spletnega strežnika na Raspberry Pi računalniku. Zanimal naju je predvsem povprečni odzivni čas ob različnih bremenih.

10.2 Definicija strežnika

Kot že rečeno sva uporabila računalnik Raspberry Pi, natančneje Raspberry Pi 2 model B [70]. Ta model uporablja štiri jedrni ARM Cortex-A7 procesor (Broadcom BCM2836 SoC) s taktom ure 900 MHz. Ima 1 GB delavnega pomnilnika tipa LPDDR [71]. Uporablja tudi 10/100 BaseT Ethernet priključek, kar pomeni, da lahko dosega hitrosti prenosa do 100 Mbit/s.

Kot OS je na Raspberry Pi nameščen Raspbian [72], ki bazira na Debian "jessie"distribuciji. Na ta Linux sva namestila LAMP spletni strežnik. Vlogo spletnega strežnika je prevzel Apache. Za podatkovno bazo sva uporabila MySQL. Uporabljeni programski jezik na strežniški strani je bil PHP. Shema strežnika



Slika 10.1: Osnovna shema strežnika.

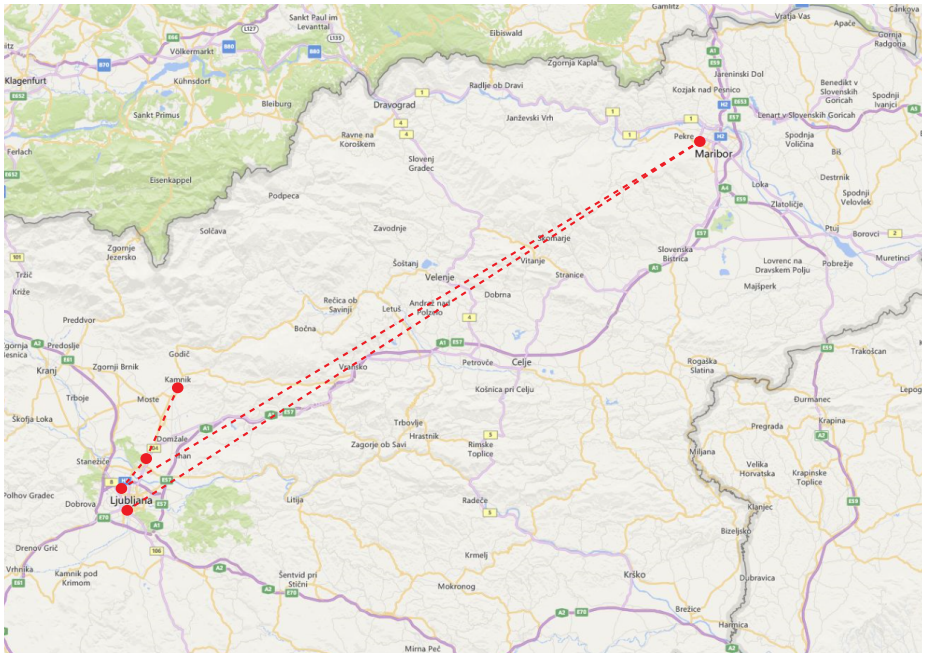
je predstavljena na sliki 10.1.

10.3 Definicija odjemalca

Ločila sva dva tipa odjemalcev. Prvi tip odjemalca je bil na istem lokalnem omrežju kot strežnik. Drugi tip ne bo na istem območju, tako, da bo promet tekel preko IP ponudnika (ISP). Strežnik se je nahajal v Kamniku (download 20 Mb/s, upload 1 Mb/s), ne lokalni odjemalci, tisti ki zahtevke pošiljajo preko ISP pa so se nahajali v Ljubljani (download 60 Mb/s, upload 2 Mb/s). Med odjemalcem in strežnikom je tako 21 km zračne razdalje, vendar je dejanska razdalja bistveno daljša. Orodje **tracert** je pokazalo, da komunikacija poteka po naslednjih korakih Ljubljana -> Maribor -> Ljubljana -> Trzin -> Kamnik, kar je vidno na sliki 10.2. V obeh primerih je odjemalec pošiljal preproste HTTP zahtevke s katerimi je zahteval spletno stran.

10.4 Breme

Celotni Raspberry Pi je bil namenjen LAMP strežniku tako, da so glavni vir bremena predstavljali odjemalci s svojimi HTTP zahtevki. Odjemalce sva si-



Slika 10.2: Grafični prikaz orodja **tracert**.

mulirala z **ab (Apache Benchmark)** [74, 73]. To je orodje za ocenjevanje in primerjanje zmogljivosti HTTP strežnikov. Glavni namen tega orodja je prikazati, kako se obnaša strežnik pod različnimi bremeni. Poleg tega sva uporabila še **JMeter** [75]. To je orodje za testiranje zmogljivosti aplikacij, prvotno namenjeno za testiranje spletnih aplikacij. Z njim lahko testiramo tako statične, kot tudi dinamične vire pod raznimi obremenitvami.

10.5 Metrike

Vsako metriko sva merila za oba tipa odjemalcev (lokalnega in oddaljenega):

- **Število zahtevkov na sekundo**: Ocenjuje koliko zahtevkov je bilo poslanih ciljnemu strežniku v eni sekundi (ang. *Request per Second* oziroma *Average Load*). Povprečje se po navadi izračuna v intervalu od 1 do 5 minut.
- **Intenzivnost napak**: Ob velikih bremenih so napake bolj pogoste. Sem spadajo HTTP napake s statusnimi številkami 4xx ter 5xx. Ta metrika se meri v procentih, cilj pa je, da je procent čim manjši.
- **Povprečni odzivni čas**: Je ena glavnih metrik, ki meri povprečni čas od trenutka ko odjemalec pošlje zahtevek do prejetja odgovora (vključuje

tudi čas, ki ga strežnik potrebuje za pripravo odgovora). Meri se v milisekundah.

- **Čas neprekinjenega delovanja**: Meri se v procentih. Pove nam kakšen odstotek vsega časa v nekem časovnem intervalu je strežnik deloval. Pričakuje se vsaj 99% dostopnost.
- **Poraba procesorskih virov**: Meri se odstotek procesorskega časa, ki ga potrebuje strežnik za serviranje zahtevkov.
- **Poraba pomnilnika**: Meri se odstotek celotnega pomnilnika, ki ga uporablja strežnik za serviranje zahtevkov.
- **Število niti**: Spletna stran generira več niti, te pa so omejene s strojno opremo ter operacijskim sistemom.
- **Delež odprtih deskriptorjev**: Datoteke so lahko odprte za branje ali pa za pisanje. Sem spadajo tudi mrežni vtičniki. Ker ima operacijski sistem omejeno število odprtih deskriptorjev, nam to predstavlja problem. Metrika se meri v procentih (število odprtih deskriptorjev glede na število vseh možnih deskriptorjev, ki jih OS dovoljuje).

10.6 Rezultati meritev

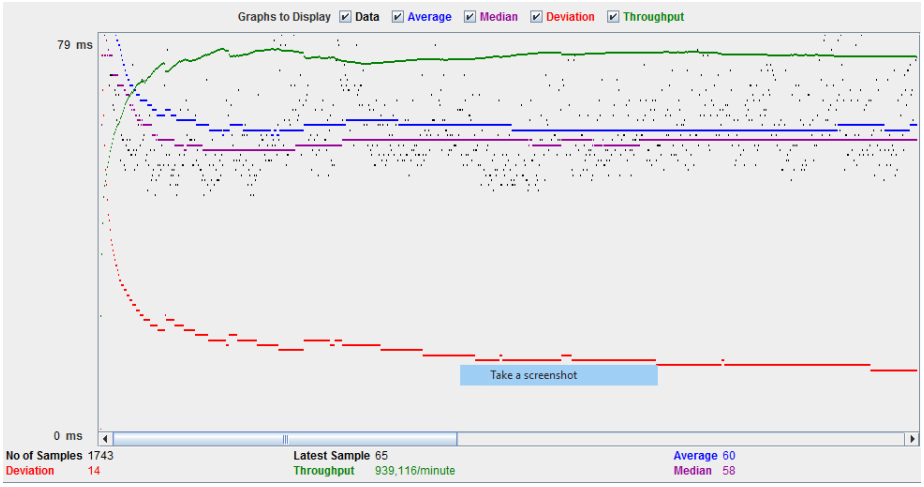
Vse meritve sva izvedla preko lokalnega omrežja ter preko internetnega ponudnika. Strežnik se je nahajal v Kamniku z naslednjimi internetnimi hitrostmi: download 20 Mb/s, upload 1 Mb/s, ping med ponudnikom in strežnikom pa je 36 ms (izmerjeno s pomočjo Speedtest). Uporabila sva tri različna orodja: ukaz **ab**, **JMeter** in **loader.io**, ki je spletna storitev za testiranje zmogljivosti spletnih aplikacij. Vsa orodja so dala zelo podoben rezultat. Opazi pa se velika razlika med lokalnimi in oddaljenimi zahtevki. V nadaljevanju so predstavljeni rezultati posameznih testov.

10.6.1 Poraba procesorja in pomnilnika

Poraba procesorja in pomnilnika je bila vseskozi zelo nizka. Procesorska poraba se je gibala okoli 0.7% na eno nit apache strežnika. Pomnilniška poraba je bil povsem zanemarljiva.

10.6.2 JMeter v lokalnem omrežju

Najprej sva uporabila orodje **JMeter** znotraj lokalnega omrežja (prvi tip odjemalca). Rezultati so vizualizirani na sliki 10.3. Vidi se, da se je povprečni odzivni čas ustalil pri 60 ms. To je čas od trenutka ko zahtevek zapusti računalnik pa do prejetja odgovora.



Slika 10.3: Rezultati testiranja z JMeter-om v lokalnem omrežju.

10.6.3 JMeter preko spletnega ponudnika

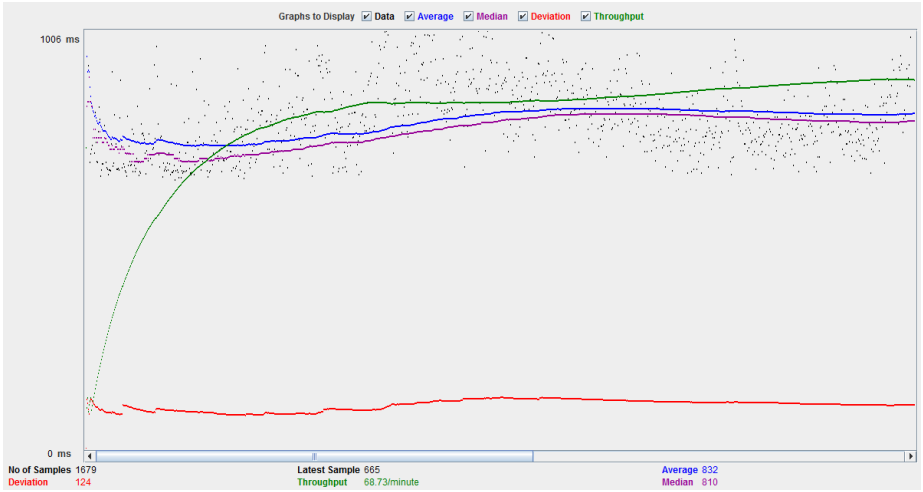
Nato sva orodje **JMeter** uporabila še preko spletnega ponudnika (drugi tip odjemalca). Rezultati so prikazani na sliki 10.4 in so grafično zelo podobni rezultatom **JMeter-a** v lokalnem omrežju. Glavna razlika je, da se je tu povprečni odzivni čas ustalil pri 832 ms, kar je približno štirinajstkrat daljši odzivni čas. Čas se spet mero od odpošiljanja zahtevka pa do prejetja odgovora. Sklepava da je razlog za tako večji odzivni čas slaba internetna hitrost na strani spletnega strežnika.

10.6.4 Primerjava povprečnih odzivnih časov

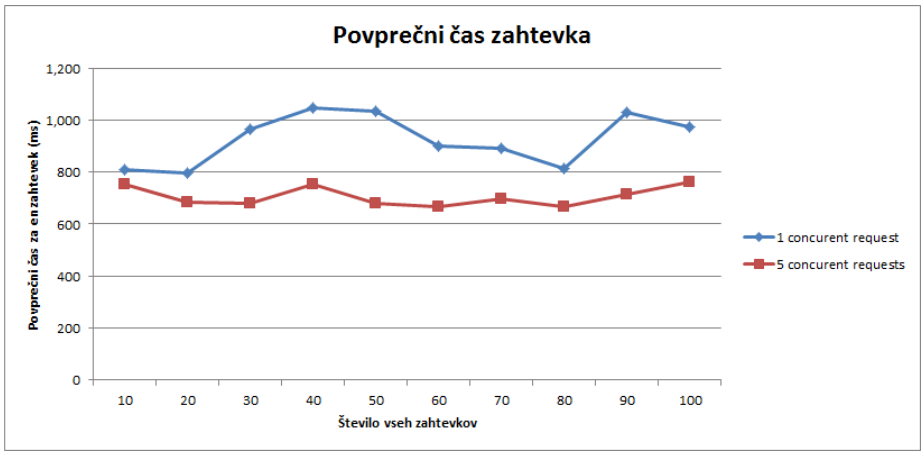
Nato sva uporabila **ab** orodja, da sva primerjala povprečne odzivne čase zahtevkov, če sva pošiljala le po en zahtevek na enkrat, ali pa 5 zahtevkov na enkrat. Rezultati so grafično prikazani na sliki 10.5. Razvidno je, da so povprečni odzivni časi manjši, če pošiljamo po 5 sočasnih zahtevkov, saj takrat pride do boljše izkoriščenosti procesorja Izmerjeni čas se še vedno meri od trenutka odpošiljanja zahtevka pa do prejetja odgovora.

10.6.5 Primerjava števila neuspelih zahtevkov

Na sliki 10.6 je prikazana primerjava števila neuspešno serviranih zahtevkov med testom, ki sočasno pošlje en zahtevek in testom, ki sočasno pošlje pet zahtevkov. Testi so bili izvedeni z **ab** orodjem. Z slike se vidi, da ima test s petimi sočasnimi zahtevki nekoliko več neuspešnih zahtevkov, kar je normalno, saj je strežnik pod večjo obremenitvijo.



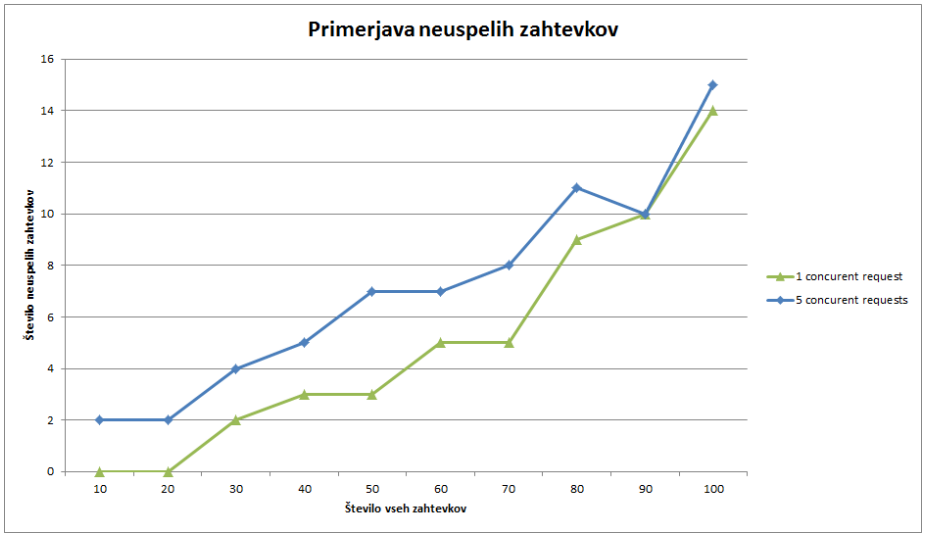
Slika 10.4: Rezultati testiranje z JMeter-om preko ISP ponudnika.



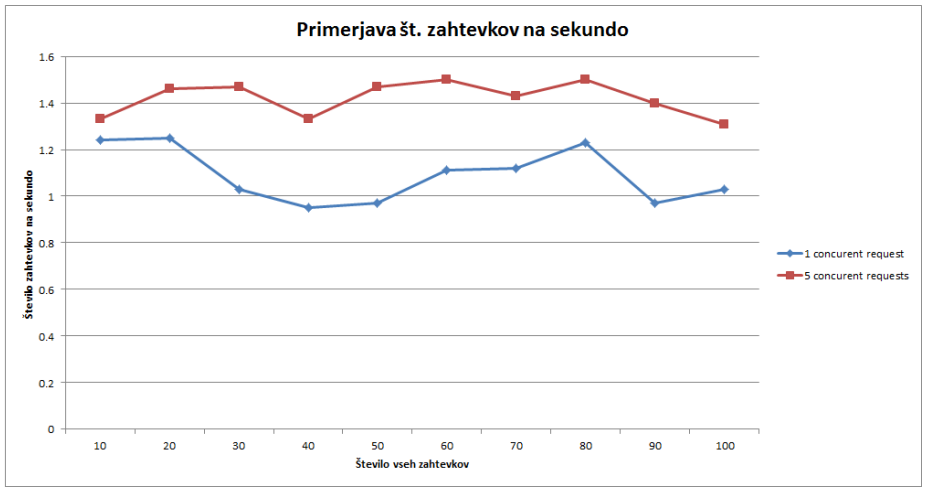
Slika 10.5: Primerjava povprečnih odzivnih časov zahtevkov.

10.6.6 Primerjava števila uspešno serviranih zahtevkov na sekundo

Slika 10.7 prikazuje koliko uspešnih zahtevkov, servira strežnik na sekundo. Med uspešno servirane zahtevke se štejejo samo tisti, ki jih je strežnik uspešno obdelal in poslal odgovor. Test s petimi sočasnimi zahtevki se tukaj obnaša dosti boljše, kot pa test z enim sočasnim zahtevkom. Test je bil opravljen z **ab** orodjem.



Slika 10.6: Primerjava števila neuspelih zahtevkov.

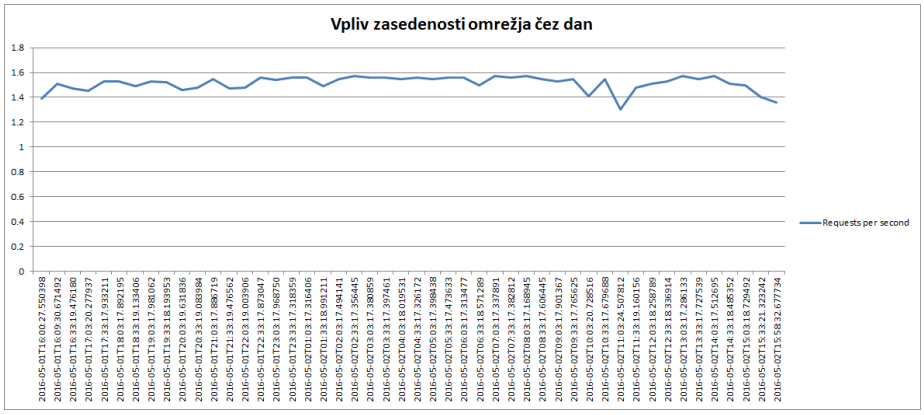


Slika 10.7: Primerjava števila uspešno serviranih zahtevkov na sekundo

10.6.7 Prikaz povprečnega števila zahtevkov v obdobju enega dne

Na sliki 10.8 je prikazano povprečno število izvedenih zahtevkov na sekundo. Meritve so bile izvedene med prvim in drugim majem. Vidi se, da tekom dneva zasedenost omrežja ni imela velikega vpliva. Meritve sva izvedla s pomočjo

skripte 10.8, ki se nahaja v prilogi.



Slika 10.8: Prikaz povprečnega števila zahtevkov v obdobju enega dne

10.6.8 Prikaz povprečnega števila zahtevkov v obdobju enega tedna

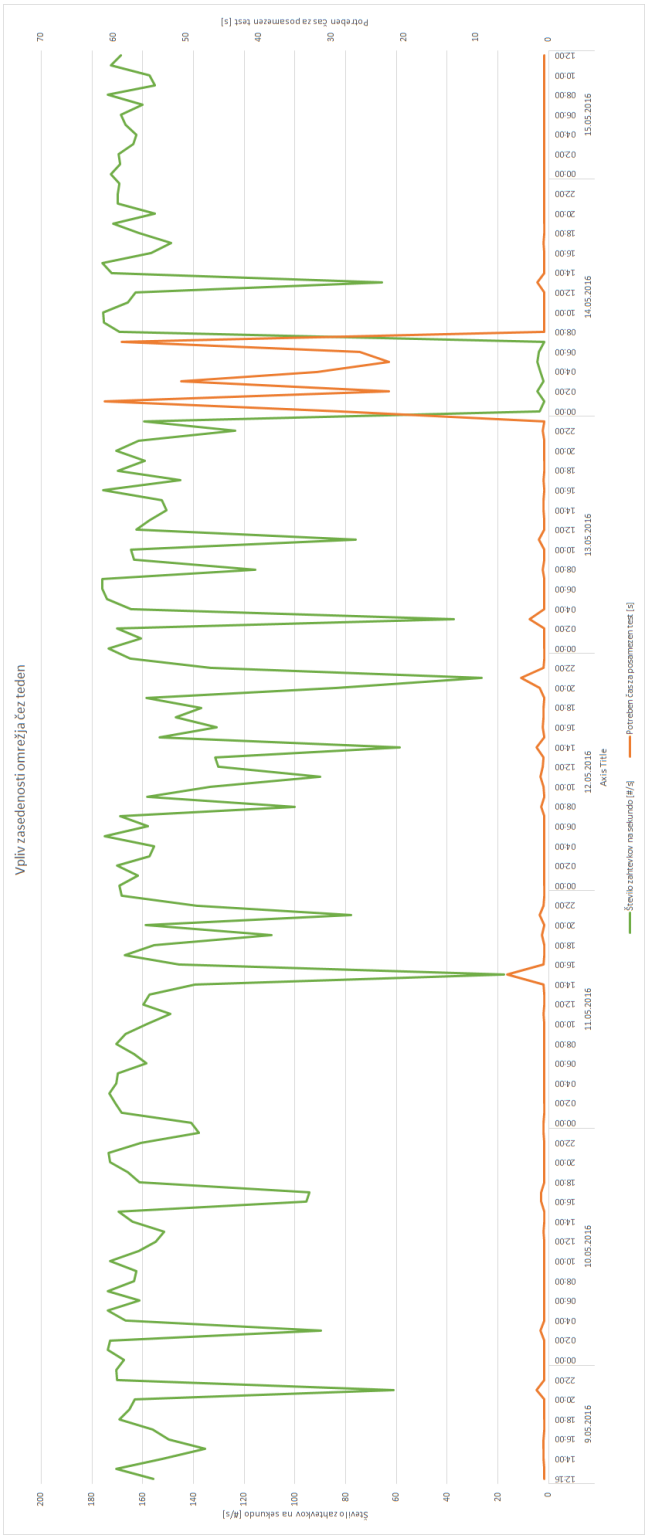
Ker v času enega dne nisva pridobila nobenih relevantnih podatkov sva test ponovila v času enega tedna. Meritve sva izvajala med 9.05.2016 in 15.05.2016. Rezultati so prikazani na sliki 10.9. Zasedenost omrežja spet ni imela nobenega vpliva na število poprečnih zahtevkov na sekundo. Sva pa tekom testa uporabljala izboljšane nastavitve, ki so izboljšale zmogljivost strežnika. Povprečni odzivni čas se je tako spustil na 280 ms kar je približno trikrat hitreje kot z starimi nastavitvami. Meritve sva izvedla s pomočjo skripte 10.8, ki se nahaja med prilogami.

V datoteki */etc/security/limits.conf* je bilo potrebno ročno nastaviti maksimalno število procesov in maksimalno število odprtih datotek.

```
# /etc/security/limits.conf
# ...
# <domain> <type> <item> <value>
   * soft nofile 10240
   * soft nproc 10240
# ...
```

10.7 Zaključek

Cilj testiranja je bil preveriti, če lahko računalnik Raspberry Pi služi kot cenovno ugodna alternativa spletnega strežnika za majhna podjetja z malo spletnega prometa. Glede na rezultate meritev sva mnenja da ni ugodna alternativa. Cene spletnega gostovanja se začnejo že pri 12 evrih na leto medtem ko je najin Raspberry Pi stal okoli 40 evrov. Nato pa je potrebno imeti še hitro internetno povezavo, kar še dodatno poveča stroške, medtem ko nam pri spletnem gostovanju za to ni potrebno skrbeti. Poleg cenovnih razlogov pa so tu še tehnični. Na Raspberry Pi je potrebno ročno namestiti vso programsko opremo, ki jo potrebujemo (Apache, PHP, ...). Nato je po vsej verjetnosti potrebno nastaviti statičen IP naslove ter *port forwarding*. Poleg tega je potrebno konfigurirati spletni strežnik, da doseže optimalno zmogljivost.



Slika 10.9: Prikaz povprečnega števila zahtevkov v obdobju enega tedna

10.8 Python 'ab' skripta

```
import subprocess
import datetime
import re

# UREDI - Pot do ukaza 'ab', pride z 'apache' streznikom
AB_PATH = "C:/wamp/bin/apache/apache2.4.17/bin/ab.exe"
# UREDI - Pot do root mape te skripte
ROOT_PATH = "E:/matja/Google Drive/Fri/Fri 2015_2016/S02/ZZRS/ab"

# UREDI
AB_N = "10" # <- koliko odjemalcev
AB_C = "2" # <- koliko socasnih odjemalcev
AB_URL = "http://mavpi.ddns.net/"

AB_OUT = datetime.datetime.today().___format___("%Y%m%dT%H%M%S")
AB_OUT_PATH = ROOT_PATH + "/log/" + AB_OUT + ".log"
AB_CSV_PATH = ROOT_PATH + "/data.csv"

# IZVRSI AB UKAZ
AB = AB_PATH + " -n " + AB_N + " -c " + AB_C + " " + AB_URL + " >
    \"" + AB_OUT_PATH + "\""
subprocess.call(AB, shell=True)

# ODPRI DATOTEKO
ab_out = open(AB_OUT_PATH, "r+")
# PRIDOBI KLJUCNE VRSTICE
lines = ab_out.readlines()[14:25]
lines.pop(4)
ab_out.seek(0)
ab_out.truncate()
ab_out.writelines(lines)
# ZAPRI DATOTEKO
ab_out.close()

ab_csv = open(AB_CSV_PATH, "a")
# PRECISTI PODATKE
csv = []
for line in lines:
    match = re.search(r"[\d.]+", line)
    if match:
        csv.append(match.group())
csv = ",".join(csv)
ab_csv.write(datetime.datetime.today().isoformat()+","+csv+"\n")
ab_csv.close()

# PRIMER LOG DATOTEKE
# ====================
```

```
#
# Concurrency Level: 2
# Time taken for tests: 1.085 seconds
# Complete requests: 10
# Failed requests: 9
# Total transferred: 113696 bytes
# HTML transferred: 107606 bytes
# Requests per second: 9.21 [#/sec] (mean)
# Time per request: 217.059 [ms] (mean)
# Time per request: 108.530 [ms] (mean, across all concurrent requests)
# Transfer rate: 102.31 [Kbytes/sec] received
```

Literatura

[1] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 6, pp. 931–945, 2011.

[2] "Digitalocean: Simple cloud computing for developers." https://www.digitalocean.com/, Marec 2016.

[3] "Amazon web services (aws) - cloud computing services." https://aws.amazon.com/, Marec 2016.

[4] "Google cloud computing, hosting services & apis." https://cloud.google.com/, April 2016.

[5] "Perfkit benchmarker." http://googlecloudplatform.github.io/PerfKitBenchmarker/, Marec 2016.

[6] "Welcome to python.org." https://www.python.org/, Marec 2016.

[7] "iperf - the network bandwidth measurement tool." https://iperf.fr/iperf-doc.php, April 2016.

[8] "Industry-standard benchmarks for embedded systems." http://www.eembc.org/coremark/faq.php, April 2016.

[9] "Fio - flexible i/o tester synthetic benchmark." http://www.storagereview.com/fio_flexible_i_o_tester_synthetic_benchmark, April 2016.

[10] "How the scp protocol works." https://blogs.oracle.com/janp/entry/how_the_scp_protocol_works, April 2016.

[11] "Digital Ocean." `https://www.digitalocean.com/`, March 2016.

[12] "Amazon AWS." `http://aws.amazon.com/`, March 2016.

[13] "Transip." `https://www.transip.eu/`, March 2016.

[14] "Benchmarking guides." `https://openbenchmarking.org/suites/pts`, March 2016.

[15] "Sysbench." https://www.howtoforge.com/how-to-benchmark-your-system-cpu-file-io-mysql-with-sysbench, April 2016.

[16] "How to use sysbench." https://wiki.mikejung.biz/Sysbench, April 2016.

[17] "Phoronix Test Suite." http://www.phoronix-test-suite.com/, April 2016.

[18] "Phoronix Test Suite on Ubuntu." https://wiki.ubuntu.com/PhoronixTestSuite, April 2016.

[19] "Digital Trends." http://www.digitaltrends.com/computing/raspberry-pi-vs-banana-pi/, March 2015.

[20] "Open maniak." http://openmaniak.com/iperf.php, 2010.

[21] "How To Benchmark Your System." https://www.howtoforge.com/how-to-benchmark-your-system-cpu-file-io-mysql-with-sysbench.

[22] "RAID." https://en.wikipedia.org/wiki/RAID.

[23] "ftpbench." https://github.com/selectel/ftpbench.

[24] "Cloud9 IDE." https://c9.io/, Marec 2016.

[25] "HTML." https://en.wikipedia.org/wiki/HTML/, Marec 2016.

[26] "CSS." https://en.wikipedia.org/wiki/Cascading_Style_Sheets, Marec 2016.

[27] "JavaScript." https://www.javascript.com/, Marec 2016.

[28] "Node.js." https://nodejs.org/, Marec 2016.

[29] "Firefox Network Monitor." https://developer.mozilla.org/en-US/docs/Tools/Network_Monitor, Marec 2016.

[30] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful web services vs. big'web services: making the right architectural decision," in *Proceedings of the 17th international conference on World Wide Web*, pp. 805–814, ACM, 2008.

[31] F. F.-H. Nah, "A study on tolerable waiting time: how long are web users willing to wait?," *Behaviour & Information Technology*, vol. 23, no. 3, pp. 153–163, 2004.

[32] K. D. Lee, *Python Programming Fundamentals*, ch. Event-Driven Programming, pp. 149–165. London: Springer London, 2011.

[33] "Bananapi - home page." http://bananapi.com/index.php/2-uncategorised/59-what-is-bpi-r1-2.

[34] "What is a reverse proxy server?." https://www.nginx.com/resources/glossary/reverse-proxy-server/.

[35] "Reverse proxy example." https://www.nginx.com/resources/wiki/start/topics/examples/reverseproxycachingexample/.

[36] "Operating system virtualization." https://en.wikipedia.org/wiki/Operating-system-level_virtualization, Maj 2016.

[37] "Virtualization." https://en.wikipedia.org/wiki/Virtualization, Maj 2016.

[38] "Virtualization comparison." https://en.wikipedia.org/wiki/Comparison_of_platform_virtualization_software, Maj 2016.

[39] "OpenVZ." https://en.wikipedia.org/wiki/OpenVZ, Maj 2016.

[40] "VMWare ESX." https://en.wikipedia.org/wiki/VMware_ESX, Maj 2016.

[41] "VMWare Server." https://en.wikipedia.org/wiki/VMware_Server, Maj 2016.

[42] "CentOS." https://www.centos.org/, April 2016.

[43] "UnixBench." https://github.com/kdlucas/byte-unixbench, Maj 2016.

[44] "interbench." http://users.tpg.com.au/ckolivas/interbench/, Oktober 2015.

[45] "iperf." https://github.com/esnet/iperf, Maj 2016.

[46] "System comparison." http://cloudacademy.com/blog/container-virtualization/, Maj 2016.

[47] "ServerBear." http://serverbear.com/benchmarks, Maj 2016.

[48] "User beancounters." https://openvz.org/UBC, Julij 2015.

[49] "Digital Ocean." https://www.digitalocean.com/, April 2016.

[50] "NodeJS." https://nodejs.org/, March 2016.

[51] "PostgreSQL." http://www.postgresql.org/, March 2016.

[52] "Node-Postgres library." https://github.com/brianc/node-postgres, March 2016.

[53] "Python." https://www.python.org/, March 2016.

[54] "ADM Zip." https://github.com/cthackers/adm-zip, March 2016.

[55] "UUID Generator." https://github.com/broofa/node-uuid, March 2016.

[56] "Load balancing." https://en.wikipedia.org/wiki/Load_balancing_(computing), May 2016.

[57] "Raid 0." https://en.wikipedia.org/wiki/Standard_RAID_levels#RAID_0, May 2016.

[58] "Rasperry Pi." https://www.raspberrypi.org/, March 2016.

[59] "DigitalOcean." http://www.digitalocean.com, March 2016.

[60] "Icecast2." http://www.icecast.org/docs/, March 2016.

[61] "Ices." http://icecast.org/ices/docs/ices-2.0.2/, March 2016.

[62] "HowToForge." https://www.howtoforge.com/linux_webradio_with_icecast2_ices2, March 2016.

[63] G. Inc., "Cloudharmony." https://cloudharmony.com/. [Dosegljivo; dostop 19 Maj, 2016].

[64] M. T. Myers), "Azure cli." https://azure.microsoft.com/en-us/documentation/articles/storage-azure-cli/. [Dosegljivo; dostop 19 Maj, 2016].

[65] Google, "Gsutil." https://cloud.google.com/storage/docs/gsutil. [Dosegljivo; dostop 19 Maj, 2016].

[66] Amazon, "Aws cli." https://aws.amazon.com/cli/. [Dosegljivo; dostop 19 Maj, 2016].

[67] S. (user606723), "Effect of distance from server on page load speed." http://serverfault.com/questions/346906/effect-of-distance-from-server-on-page-load-speed, 2012. [Dosegljivo; dostop 19 Maj, 2016].

[68] A. L. Andrej Česen, Denis Božović, "Analiza hitrosti oblačne storitve amazon s3." http://lrss.fri.uni-lj.si/sl/teaching/zzrs/lectures/mro.pdf, 2015. [Dosegljivo; dostop 19 Maj, 2016], stran: 85-97.

[69] Amazon, "Aws cli - sync." http://docs.aws.amazon.com/cli/latest/reference/s3/sync.html. [Dosegljivo; dostop 19 Maj, 2016].

[70] "Rasperry Pi hardware." http://akizukidenshi.com/download/ds/rs/raspberrypi2b.pdf, March 2016.

[71] "LPDDR." https://en.wikipedia.org/wiki/Mobile_DDR, March 2016.

[72] "Raspbian OS." https://www.raspbian.org/, March 2016.

[73] "Howto: Performance Benchmarks a Webserver." http://www.cyberciti.biz/tips/howto-performance-benchmarks-a-web-server.html, November 2008.

[74] "ab - Apache HTTP server benchmarking tool." https://httpd.apache.org/docs/2.4/programs/ab.html, April 2016.

[75] "Apache JMeter." http://jmeter.apache.org/, April 2016.