Analiza zmogljivosti oblačnih in strežniških storitev

Uredil prof. dr. Miha Mraz

Maj 2015

Kazalo

Predgovor

Pričujoče delo je razdeljeno v devet poglavij, ki predstavljajo različne analize zmogljivosti nekaterih oblačnih izvedenk računalniških sistemov in njihovih storitev. Avtorji posameznih poglavij so slušatelji predmeta *Zanesljivost in zmogljivost računalniških sistemov*, ki se je v štud.letu 2014/2015 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 2015

Poglavje 1

Zmogljivostna analiza oglaševalskega sistema

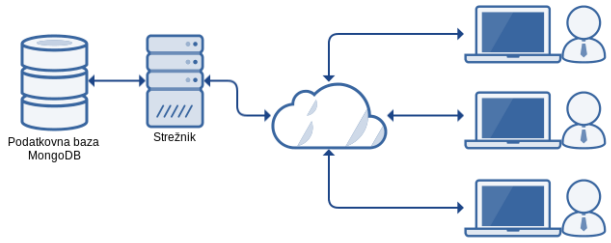
Ajda Lampe, Dejan Štepec, Anže Medved

1.1 Predstavitev ideje

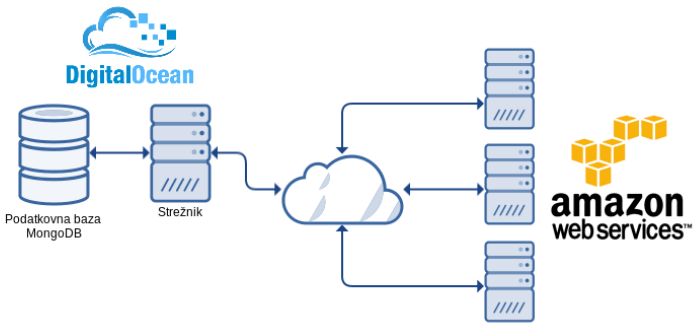
V pričujočem poglavju opišemo izvedbo simulacije sistema za spletno oglaševanje. Sistem prejme preko HTTP POST zahtevka sporočilo v obliki JSON, v katerem je zapisan IP naslov uporabnika, ki si ogleduje spletno stran. Na podlagi IP naslova izvede geolociranje, s katerim pridobi državo uporabnika, ki služi kot ključ za poizvedbo v bazi oglasov. Sistem odgovori na zahtevek z JSON sporočilom, ki vsebuje HTML kodo oglasa.

Za namene testiranja smo pripravili strežniško aplikacijo in odjemalca, ki bo pošiljal zahteve na strežnik. Pri testiranju smo merili odzivni čas sistema, torej čas od poslane zahteve, do prejetja HTML kode oglasa. Zahteve, ki smo jih pošiljali proti strežniku, so bile večinoma enolične in majhne, vendar smo spreminjali frekvenco poslanih zahtev. Odgovore, ki smo jih prejemali od strežnika, smo po velikosti spreminjali (od nekaj kB do nekaj 10kB). Za bolj nadzorovano testiranje so IP naslovi v odjemalcu določeni vnaprej (naslovi različnih DNS strežnikov), tako da smo v naprej vedeli, kakšen "oglas"nam mora sistem vrniti.

Na sliki 1.1 je prikazan diagram uporabe realnega sistema, na sliki 1.2 pa diagram simulacijskega okolja.



Slika 1.1: Diagram realnega sistema.



Slika 1.2: Diagram simulacijskega okolja.

1.2 Izbira ponudnikov oblačnih storitev

Aplikacijo gostujemo pri dveh ponudnikih oblačnih storitev, in sicer DigitalOcean [1] in Amazon [2]. Pri obeh ponudnikih so osnovne storitve za nas na voljo brezplačno - pri DigitalOcean dobimo kot študenti 100$ kredita [3], pri Amazonu pa imamo za prvo leto uporabe brezplačnih 750 ur delovanja na mesec [4]. Amazonova gruča nam omogoča, da lahko hkrati poganjamo več instanc odjemalcev in s tem spreminjamo frekvenco poizvedb.

1.3 Izbira tehnologij

Z namenom enostavne implementacije sistema, smo uporabili več različnih tehnologij.

1.3.1 NodeJS

Kot osnovo za aplikacijo, ki bo tekla na strežniku, smo izbrali NodeJS [5], ki temelji na Googlovem V8 Javascript engine-u [6]. Izbrali smo ga, ker omogoča izdelavo hitrih in skalabilnih omrežnih aplikacij. Zaradi dogodkovno vodenega modela in neblokirajočih vhodno-izhodnih operacij je najbolj primeren za podatkovno intenzivne realnočasovne aplikacije. Velika prednost pa je tudi obsežna zbirka knjižnjic, ki je dostopna preko upravljalca paketov NPM (več o upravljalcu paketov najdete v viru [7]).

1.3.2 MongoDB

Bazo smo implementirali v MongoDB [8], ki omogoča delo z dokumentno usmerjenimi podatkovnimi bazami. Podatki so shranjeni v bazi v obliki JSON dokumenta, tako da je pridobivanje iz baze z uporabo NodeJS hitro.

1.3.3 Python

Odjemalca smo napisali v Pythonu, saj omogoča najenostavnejšo pošiljanje HTTP zahtev brez uporabe dodatnih knjižnjic. Ker bo odjemalec napisan v skriptnem jeziku, ga ne bo potrebno prevajati, temveč le zagnati.

1.4 Implementacija sistema

Implementacije sistema smo se lotili modularno - najprej smo pripravili strežniško aplikacijo, nato podatkovno bazo in na koncu še odjemalca.

1.4.1 Strežniška aplikacija

Pripravili smo strežniško aplikacijo napisano v NodeJS, ki deluje po arhitekturnem stilu REST storitev. Aplikacija navzven ponuja POST storitev, ki jo kliče

odjemalec, ko želi uporabiti funkcionalnost strežniške aplikacije.

Odjemalec ob klicu storitve pošlje HTTP POST zahtevek, ki v zaglavju vsebuje sporočilo v obliki JSON formata. Sporočilo lahko vsebuje več polj, med katerimi mora vsebovati polje myIP v katerem je zapisan IP naslov odjemalca. Strežniška aplikacija sporočilo razčleni in pridobi vrednost polja myIP, ki ga nato posreduje knjižnjici za geolociranje.

Za izvedbo geolociranja smo uporabili odprtokodno knjižnico geoip-lite, ki preslika dani IP naslov v geolokacijske podatke (država, regija, mesto, ...). Knjižnica trenutno podpira standarda IPv4 in IPv6, vendar za naslove iz standarda IPv6 vsebuje samo podatke o državi. Več informacij o knjižnici je na voljo v povezavi vira [9].

Vrnjene geolokacijske podatke aplikacija nato uporabi za izvedbo poizvedbe na podatkovni bazi. V bazo smo za namene prvega testiranja aplikacije dodali samo dva ključa (US,DE) in pripadajoča oglasa v obliki HTMLja. Bazo smo nato dopolnili, tako da vsebuje večje število ključev, ki vračajo različno velike rezultate poizvedb, testna ključa, ki sta bila namenjena zgolj za testiranje povezljivosti aplikacije in baze, pa smo odstranili.

Rezultat poizvedbe se nato vključi v zaglavje odgovora aplikacije na POST zahtevek in pošlje odjemalcu.

Aplikacijo smo namestili v oblak ponudnika DigitalOcean, in sicer v regijo Amsterdam 1.

1.4.2 Podatkovna baza

Na strežniku smo pripravili podatkovno bazo oglasov, ki so služili kot bremena pri testiranju delovanja in zanesljivosti strežnika. Kot je bilo omenjeno v uvodnih razdelkih smo bazo realizirali v nerelacijski bazi (noSQL) MongoDB. Nerelacijsko bazo smo izbrali, ker v primerjavi z relacijsko bazo ponuja večje frekvence poizvedb in manjši čas procesiranja, kar vodi do hitrejšega pridobivanja podatkov.
Podatki so shranjeni v obliki dokumentov, ki vsebujejo po dve polji - kratico države po standardu ISO-3166-1, ki služi kot ključ za poizvedbo in podatkovnem polju, ki vsebuje navidezno vsebino oglasa. Baza vsebuje oglase petih različnih velikostih, in sicer 10kB, 25kB, 50kB, 75kB, 100kB. Pred testiranjem smo si pripravili širok razpon bremen, vendar smo po prvih testih ugotovili, da so dovolj bremena v velikosti od 10 do 75kB.

1.4.3 Odjemalec

Napisali smo odjemalca v jeziku Python, ki na strežnik pošilja POST zahtevke in meri čas od trenutka poslanega zahtevka do konca prenosa odgovora. Pri

delovanju izpiše čas posameznega zahtevka, po koncu pa še skupni čas, povprečni čas, maksimalni čas in minimalni čas.
Odjemalca smo dopolnili tako, da izpisuje rezultate v datoteko, katere ime nam pove katero breme se je testiralo. Posamezne meritve zakasnitev so zapisane vsaka v svoji vrstici, kar nam je omogočilo enostavnejšo analizo in vizualizacijo meritev. Za procesiranje rezultatov smo uporabili Matlab, saj je najenostavnejši za procesiranje velike količine numeričnih podatkov. Več o analizi smo napisali v ločenem razdelku.

1.5 Nadzor delovanja

Ker imamo pri Amazonu na voljo omejeno število ur za testiranje (750 procesorskih ur), jih želimo, kar se da optimalno izkoristiti. V ta namen smo namestili na strežnik aplikacijo, ki nam preko spletnega brskalnika omogoča realnočasovni vpogled v stanje strežnika.

1.5.1 PyDash

Na sliki 1.3 je prikazana zaslonska slika vmesnika PyDash v1.4.6, ki smo ga namestili na naš podatkovni strežnik pri ponudniku DigitalOcean. Vmesnik prikazuje osnovne podatke o sistemu, porabo CPE, porabo pomnilnika, obremenitev, omrežni promet in druge uporabne podatke. S pomočjo vmesnika smo med testiranjem opazovali odzivanje strežnika in ob morebitni napaki samo testiranje prekinili. Zaslonska slika je bila narejena nekaj sekund po testiranju skripte odjemalca, kar je razvidno iz grafa obremenitve.

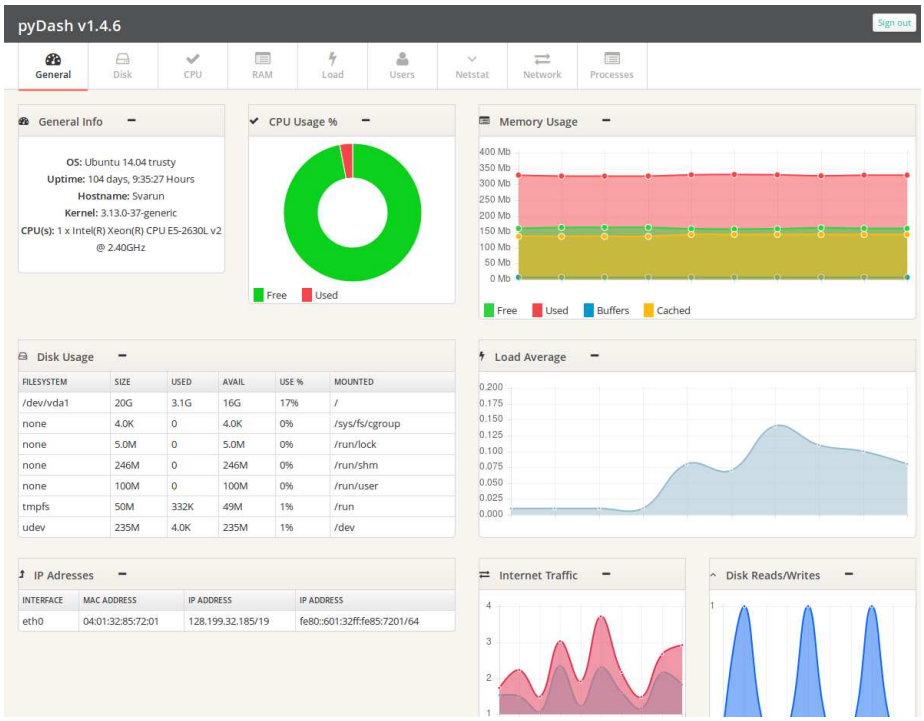
1.6 Testiranje

Za izvedbo testiranja smo prilagodili odjemalca s knjižnico mpi4py [10], za upravljanje gruče pa smo uporabili odprtokodno orodje StarCluster [11]. Orodje je namenjeno ustvarjanju, konfiguriranju in upravljanju gruče virtualnih naprav na oblačni storitvi Amazon EC2.
Testiranje sistema smo razdelili v dva dela - mali test in polni test.

1.6.1 Mali test

V malem testu smo se osredotočili na iskanje razpona velikosti bremena (oglasa), števila odjemalcev in števila poslanih zahtev posameznega odjemalca. Zahteve se zaradi blokirajočega načina izvajanja pošiljajo zaporedno. Izvedli smo več ločenih testov z naslednjimi parametri:

1. velikost odgovora: 25kB, število odjemalcev: 5, število poslanih poizvedb: 1000,
2. velikost odgovora: 25kB, število odjemalcev: 10, število poslanih poizvedb: 1000,



Slika 1.3: PyDash monitoring.

3. velikost odgovora: 25kB, število odjemalcev: 15, število poslanih poizvedb: 1000.

Za omenjene parametre smo se odločili iz več razlogov. Pri določitvi velikosti odgovora smo se osredotočili na realno velikost oglasov, ki se prikazujejo na spletnih straneh in so ponavadi v obliki slik ali animacij. Ugotovili smo, da je njihova velikost ponavadi nekaj kB, zato smo pripravili nekaj odgovorov realne velikosti in nekaj večjih, za boljšo simulacijo. Število poizvedb smo postavili na 1000, da bi odpravili nezaželjene zunanje dejavnike, ki vplivajo na zakasnitveni čas in si omogočili boljšo aproksimacijo. Hkrati nam omenjeno število zahtev omogoča preverjanje konsistentnosti meritev. Če bi število zahtev povečali, ne bi pridobili novih informacij, le testi bi se izvajali dlje.

PARAM	CPU	MEM	BAND	IO
1	28.80 %	27 %	3.4 Mb/s	2
2	36.80 %	27 %	4.1 Mb/s	2
3	51.8 %	27 %	4.8 Mb/s	2

Tabela 1.1: Meritve malega testa.

V tabeli 1.1 so predstavljene povprečne vrednosti malega testa pri različnih parametrih. Pomeni stolpcev so sledeči:

- PARAM: zaporedna številka izbranih parametrov
- CPU: povprečna poraba centralne procesne enote
- MEM: povprečna poraba pomnilnika
- BAND: povprečna poraba pasovne širine
- IO: povprečno število vhodno-izhodnih operacij

Pri opravljanju malega testa smo opazili, da strežnik zahtevan oglas shrani v predpomnilnik, saj je v prvi sekundi pošiljanja poizvedb veliko vhodno-izhodnih operacij, nato pa se to število spusti na 0. Našo domnevo smo potrdili tako, da smo strežniku pošiljali poizvedbe po različnih oglasih in s tem dosegli povečanje števila vhodno-izhodnih operacij. Opazili smo tudi, da bo potrebno podatkovno bazo dopolniti z oglasi v velikosti od 5kB do 50kB, saj bi ob večjih bremenih strežnik hitro odpovedal.
Prišli smo do zaključka, da bomo pri polnem testu uporabili odgovore v velikosti od 10kB do 100kB, uporabili od 10 do 30 odjemalcev in poslali po 1000 zahtev na odjemalca. Predpostavljamo, da se bo pri izbranih parametrih povečevala poraba CPE, zakasnitev odgovorov in varianca zakasnitve pri izbiri največjih parametrov (100kB in 30 odjemalcev) pa lahko pride do napak.

1.6.2 Polni test

Pri izvedbi polnega testa smo se osredotočili na ugotovitve iz malega testa. Za zagotavljanje večje zanesljivosti meritev in zmanjšanje vpliva zunanjih dejavnikov, smo teste večkrat ponovili v dopoldanskih urah, okoli 10:00, in večernih urah, okoli 22:00. Uporabili smo kombinacijo naslednjih parametrov:

1. velikosti odgovorov: 10kB, 25kB, 50kB, 75kB,
2. število odjemalcev: 10, 15, 20,
3. število zahtev na odjemalca: 1000.

Pri številu odjemalcev smo naleteli na omejitev s strani Amazonove gruče, saj je največja dovoljena velikost gruče 20 naprav. Na njihovo podporo smo naslovili email s prošnjo za povečanje omejitve na večje število naprav, vendar smo njihovo odobritev prejeli po izvedenem polnem testu. Po odobritvi naše prošnje za povečanje omejitve števila naprav, s strani Amazona, smo izvedli dodatni test s 100 odjemalci in odgovori velikosti 10kb in 25kb. Rezultati dodatnega testiranja so predstavljeni na koncu analize.

1.7 Rezultati in analiza

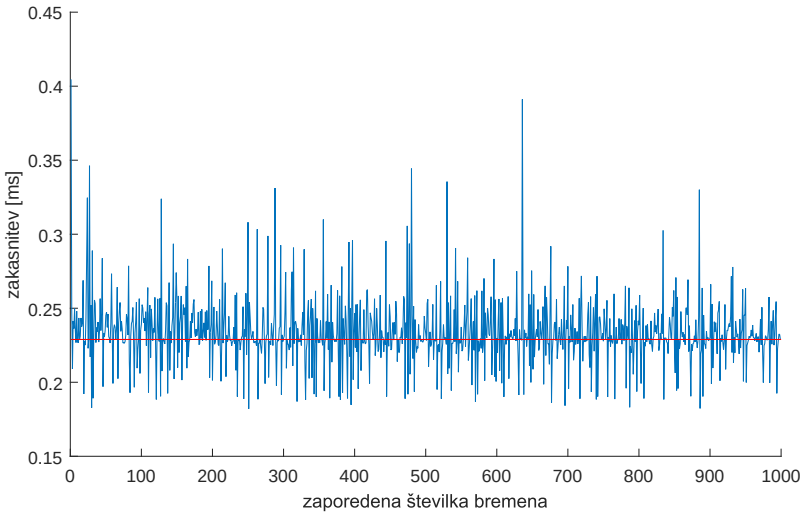
V okviru analize sta nas zanimala zakasnitev poslanih zahtev in poraba strežniških virov.

1.7.1 Zakasnitev zahtev

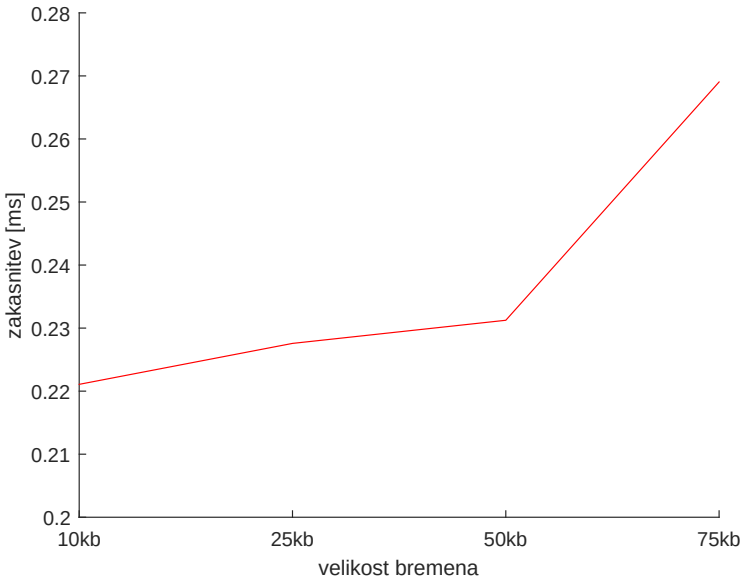
Od vseh rezultatov nas je najbolj zanimal čas zakasnitve poslanih zahtev, torej čas od poslane zahteve do prejetega odgovora. Ugotovili smo, da je zakasnitev prve poslane poizvedbe (cca 0.45 ms) vedno najvišja in približno dvakrat večja od povprečne zakasnitve (cca 0.22 ms). Na sliki 1.4 je prikazan graf zakasnitve zahtev za test z 20 odjemalci, ki so prejemali odgovore v velikosti 50kB. Iz grafa je razvidno, da je prva zakasnitev največja, preostale pa nihajo med 0.19 in 0.4 ms.

Na sliki 1.5 je prikazana povprečna zakasnitev zahtev za različne odgovore. Izkaže se, da se zakasnitev najbolj poveča pri povečanju bremena iz 50kb na 75kb, za 0.04 ms, zaradi omenjene pasovne širine strežnika.

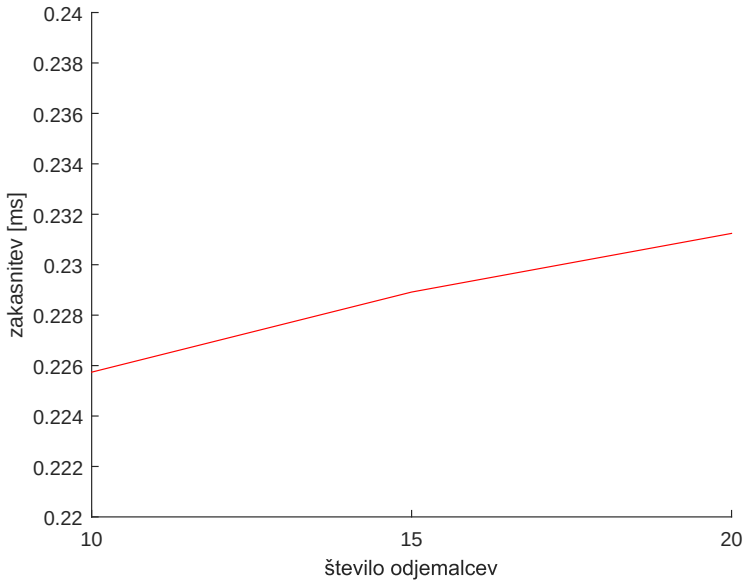
Na sliki 1.6 je prikazana povprečna zakasnitev zahtev za različno število odjemalcev. Razvidno je, da je zakasnitev izredno majhna - manj kot 0.01 ms, kar je rezultat skalabilnosti strežniške aplikacije in baze. V naslednjem razdelku, kjer je opisana analiza porabe strežniških virov je razvidno, da je aplikacija s povečanjem števila zahtev rezervirala in uporabila večje število virov, da bi lahko zahteve dovolj hitro sprocesirala.



Slika 1.4: Graf zakasnitev za 20 odjemalcev pri velikosti odgovora 50kB.



Slika 1.5: Povprečni čas zakasnitve za različne odgovore.



Slika 1.6: Povprečni čas zakasnitve za različno število odjemalcev.

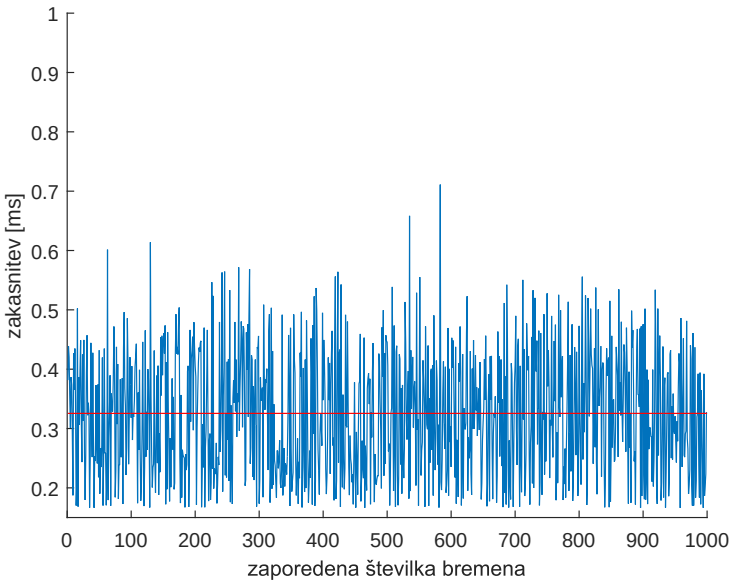
Pri dodatnem testu s 100 odjemalci smo opazili znatno povečanje zakasnitev odgovorov na zahteve. Na sliki 1.7 in 1.8 so prikazane zakasnitve odgovorov iz dodatnega testa. Če primerjamo sliki 1.8 in 1.4, vidimo da zakasnitve veliko bolj variirajo, povprečna zakasnitev pa je skoraj dvakrat večja.

1.7.2 Poraba strežniških virov

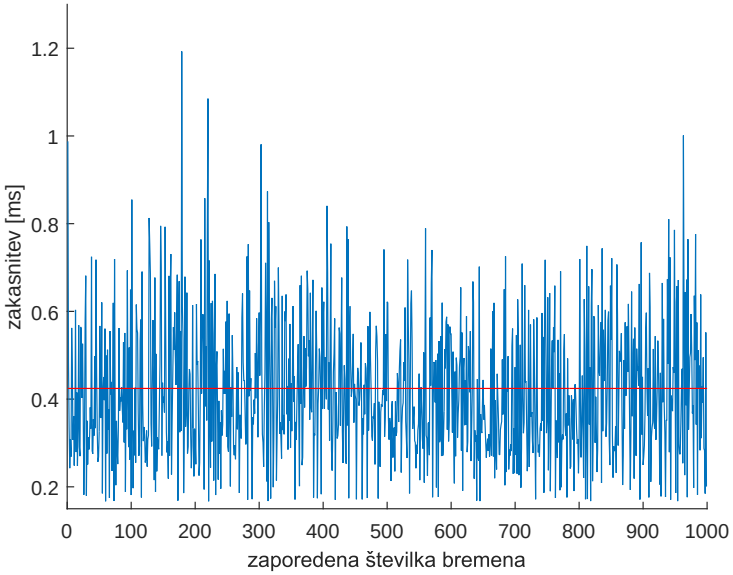
Med izvajanjem testov smo na strežniku merili porabo CPE, pomnilnika, pasovne širine in število bralno-pisalnih dostopov do diska. Za slednje se je izkazalo, da so bralno-pisalni dostopi le v prvih sekundah, saj si aplikacija shrani podatke v predpomnilnik. Poraba pomnilnika (RAM) se med izvajanjem ni spreminjala in je imela konstanto vrednost 139 MB (tudi v pasivnem stanju).

Poraba CPE se je linearno povečevala s povečevanjem števila odjemalcev in velikostjo bremen. Na sliki 1.9 je prikazana poraba CPE v odvisnosti od velikosti bremena. Na podlagi meritev smo izvedli aproksimacijo, kdaj bi aplikacija porabila celoten CPE, kar bi lahko vodilo do napak ali morebitne odpovedi (točke so prikazane z navpično črto).

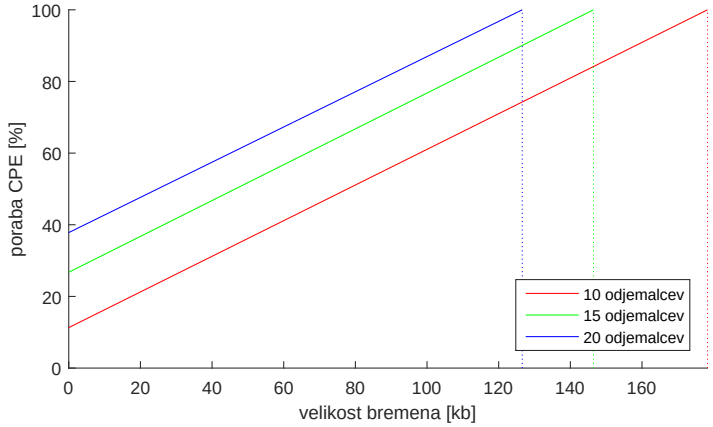
Na sliki 1.10 je prikazana poraba CPE v odvisnosti od števila odjemalcev. Zaradi omejitve s strani Amazona, smo lahko testirali zgolj z 20 odjemalci, zato smo točko morebitnih napak oz. odpovedi aproksimirali na podlagi pridobljenih meritev.



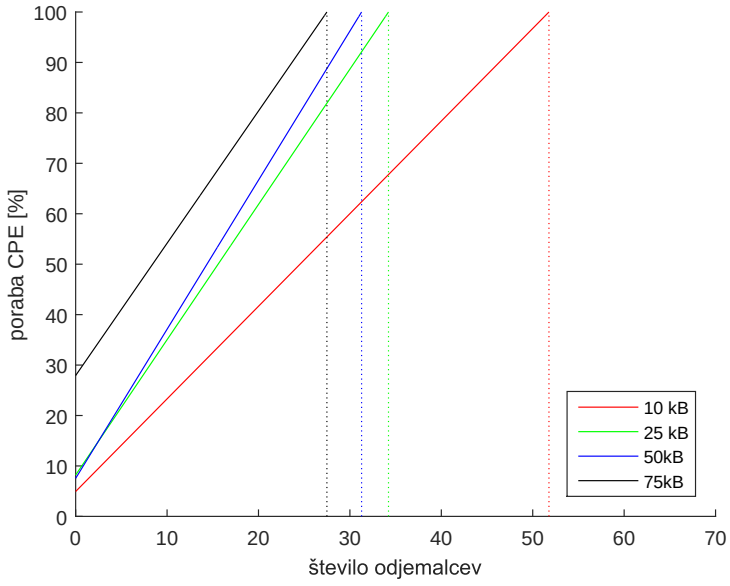
Slika 1.7: Graf zakasnitev za 100 odjemalcev pri velikosti odgovora 10kB.



Slika 1.8: Graf zakasnitev za 100 odjemalcev pri velikosti odgovora 25kB.



Slika 1.9: Poraba CPE v odvisnosti od velikosti odgovora.



Slika 1.10: Poraba CPE v odvisnosti od števila odjemalcev.

V okviru dodatnega testa smo testirali naše aproksimacije iz polnega testa. Po naših napovedih bi morala aplikacija porabiti celotno CPE pri 55 odjemalcih in 10kB velikem odgovoru. Izkazalo se je, da pri 100 odjemalcih in 10kB velikem odgovoru znaša poraba CPE le 85%. Sprva smo menili, da je prišlo do napake, zato smo test večkrat ponovili, vendar vedno dobili enak rezultat. Ugotovili smo, da je naša virtualna CPE skalabilna in se avtomatsko poveča ob povečanju obremenitve, naše aproksimacije zato veljajo samo za fiksno - neskalabilno CPE.

1.8 Zaključek

Sistem, ki smo ga postavili deluje zanesljivo in skalabilno, dokler ima na voljo dovolj sistemskih virov za nemoteno delovanje. Povprečni čas zakasnitve je 0.23 ms in narašča s povečevanjem velikosti bremena, medtem ko se s številom odjemalcev sprva spreminja minimalno, ob povečanju zmogljivosti jedra pa se znatno poveča.

Aplikacija za svoje delovanje potrebuje 140 MB pomnilnika, ki je zaseden tudi v pasivnem stanju. Poraba CPE narašča linearno s povečevanjem števila odjemalcev in velikostjo bremena. Instanca strežnika pa deluje skalabilno in povečuje zmogljivost jedra do določene meje ob povečani obremenitvi. Naša predpostavka linearnega naraščanja zato velja le za določeno obremenitev, preden se poveča zmogljivost jedra. Zaradi zagotavljanja optimalnega delovanja strežnika s strani ponudnika DigitalOcean, ni mogoče uporabljati polne zmogljivosti jedra kadar ta ni potrebna - ni dovolj obremenitve.

V sklopu testiranja smo porabili vse kredite in ugodnosti, ki so nam bile na voljo, in doplačali manjši znesek zaradi prekoračitve porabe pasovne širine.

Poglavje 2

Analiza teoretičnega nadzornega sistema

Žiga Putrle, Žiga Pušnik, Jošt Rovtar

2.1 Uvod

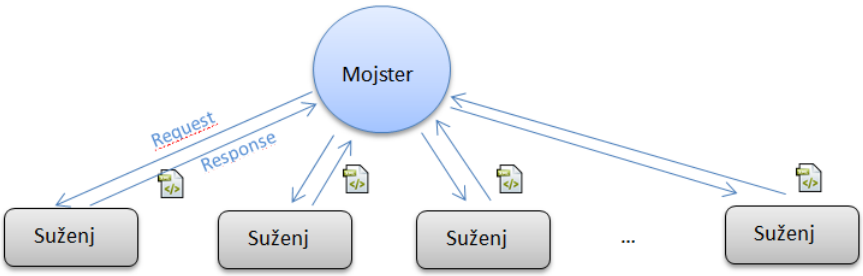
Sledeči članek povzema testiranje in simulacije teoretičnega nadzornega sistema. Analiza je potekala modularno. Nadzorni sistem bo razdeljen na manjše dele, ki so bili simulirani in analizirani s pomočjo pridobljenih podatkov.

2.2 Opis sistema

Sistem na sliki 2.1 deluje na principu "mojster-suženj". Sestavljen je iz nadzornega sistema in manjših, lokacijsko ločenih enot. Manjše enote imajo nalogo lokalnega zbiranja podatkov, ki jih na zahtevo posredujejo nadzornemu sistemu. Nadzorni sistem pridobljene podatke obdela v realnem času in na njih ustrezno regira. Komunikacija med nadzornim sistemom in enotami poteka preko omrežja v smeri mojster-suženj-mojster.

2.2.1 Parametri

Centralni nadzorni sistem je realiziran preko IaaS [12] storitve pri ponudniku Amazon Web Service [13]. Uporabili smo EC2 Ubuntu Server 14.04 LTS general purpose instanco tipa t2 micro z enim navideznim procesorjem (2.5 GHz, Intel Xeon Family) in 1GB pomnilinika. Identične instance so bile vzpostavljene na



Slika 2.1: Model komunikacije.

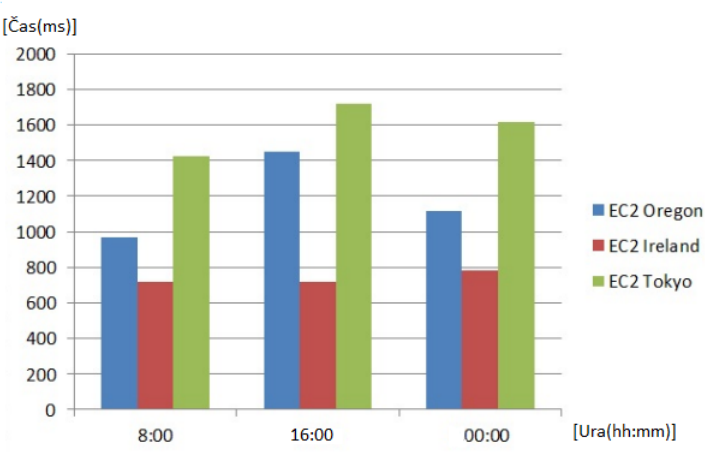
treh lokacijah Oregon, Tokio in Irska.
Uporabljeni enoti (sužnja) sta Rasberry Pi B+ [14] (lokacija: Ljubljana) in namizni računalnik srednjega cenovnega razreda (lokacija: Škofja Loka).
Uporabili smo dva ločena programa. Prvi program (ki teče na mojstru) pošlje zahtevo za podatke določenemu sužnju, ki je identificiran preko IP naslova. Drugi program (ki teče na sužnju) pa sprejme zahteve in odgovori s prenosom podatkov. Prenos poteka preko TCP protokola. Za implementacijo komunikacije smo uporabili programski jezik Java [15]. Programa sta javno dostopna na portalu BitBucket [16].
Za simulacijo podatkov smo uporabili tri XML datoteke z velikostjo 2KB, 11KB in 63KB.
Vse meritve so bile opravljene v UTC + 1 lokalnem času.

2.3 Moduli

Testiranje smo razdelili na več neodvisnih enot, ki pokrivajo najpomembnejša področja: hitrost prenosa, procesorsko moč in pomnenje podatkov. Izbrana področja imajo kritičen pomen pri odzivanju celotnega sistema v realnem času.

2.3.1 Prenos podatkov

Iskanje lokacije z najmanjšo latenco prenosa podatkov.
Simulirali smo prenos podatkov med nadzornim sistemom in ostalimi enotami. Teste smo izvajali na relacijah: Oregon - Ljubljana, Tokio - Ljubljana, Irska - Ljubljana, Oregon - Škofja Loka, Tokio - Škofja Loka in Irska - Škofja Loka. Latenco prenosa podatkov smo testirali na periodo osmih ur (8:00, 16:00, 24:00). S tem smo določili časovno območje, ki ponuja najnižjo latenco.
Na vsaki relaciji smo 100-krat zapored prenesli vsako od treh XML datotek.
Iz rezultatov je razvidno, da je največja latenca prisotna na relaciji Tokio - Ljubljana (Slika 2.2) in Tokijo - Škofja Loka (Slika 2.3). Najmanjšo latenca se



Slika 2.2: Prenos podatkov - Ljubljana.

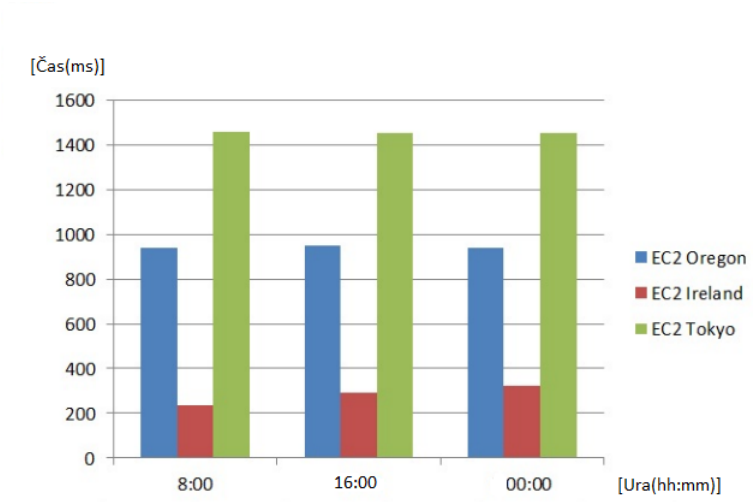
pojavi na relaciji Irska - Ljubljana in Irska - Škofja Loka. Zbiranje podatkov ob različnih urah ne vpliva na relacije vezane na Škofjo Loko. V primeru Ljubljane pa bi bilo optimalno zbirati podatke okoli 8:00 ure (Slika 2.2). Vzrok za razlike v latenci ob različnih urah, pripisujemo različni obremenitvi omrežja.
Zaradi najmanjše latence bi bila Irska najboljša izbira za postavitev nadzornega sistema. Podatki se bi pridobivali okoli 8:00 ure, saj je takrat obremenitev omrežja najmanjša.

2.3.2 Obdelava podatkov

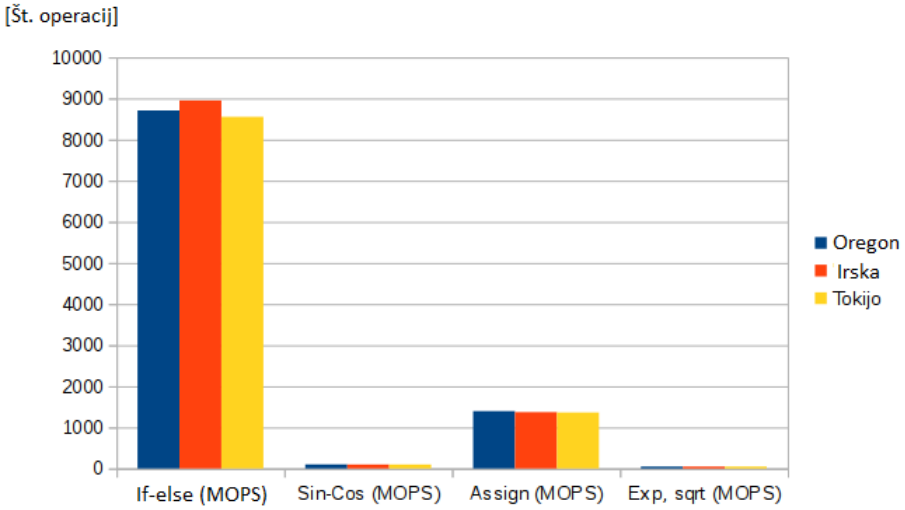
Merjenje hitrosti obdelave podatkov na različnih lokacijah je bilo opravljeno s tremi različnimi testi.
Whetstone benchmark [17] se osredotoča na merjenje specifičnih operacij. Podatki, ki so uporabljeni pri izračunih, so dovolj majhni, da ne presegajo L1 predpomnilnika in s tem ne zmanšajo natančnosti merjenja.
Operacije prisotne pri merjenju so If-Else operacije, Sinus-Cosinus operacije, določitve (en. assigned), kvadratno korenjenje (en. sqrt) in exponentne operacije (en. exp.). Uporabljena enota je MOPS (milijon operacij na sekundo).
Minimalna razlika 2.4 med posameznimi instancami je posledica različne obremenitve sistema v času testiranja.

Naslednji test, ki smo ga izvedli je Geekbench 3 [18]. Vsebuje 13 programov za ocenjevanje hitrosti operacij s celimi števili, 10 programov za ocenjevanje hitrosti operacij s plavajočo vejico, ter 4 programe za ocenjevanje hitrosti glavnega pomnilnika. Graf 2.5 prikazuje rezultate opravljenih meritev. Največja performančna razlika se je pojavila pri izvajanju floating-point in memory operacij. Najboljši razultat je dosegla instanca v Tokiu.

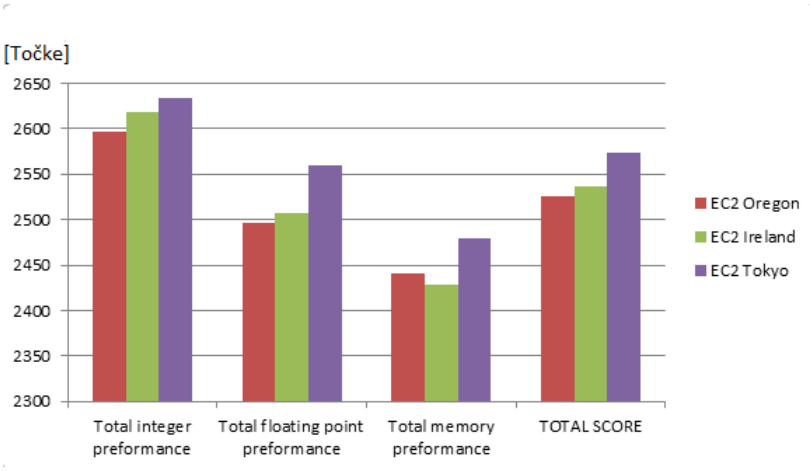
Zgornji graf prikazuje skupne rezultate opravljenih testov. Podrobno poro-



Slika 2.3: Prenos podatkov - Škofja Loka.



Slika 2.4: Whetstone - rezultati.



Slika 2.5: Geekbench - skupni rezultati.

čilo merjenja vsake od instanc si lahko ogledate na naslovih:

- Oregon - http://browser.primatelabs.com/geekbench3/2353399
- Ireland - http://browser.primatelabs.com/geekbench3/2353452
- Tokio - http://browser.primatelabs.com/geekbench3/2353511

Zadnji test, ki smo ga uporabili za testiranje hitrosti obdelave podatkov je UnixBench [19]. Ta omogoča ocenjevanje celotnega sistema na katerem teče operacijski sistem, ki temelji na Unix platformi. Test je bil zagnan na Oregon instanci in na dodatnem sistemu.

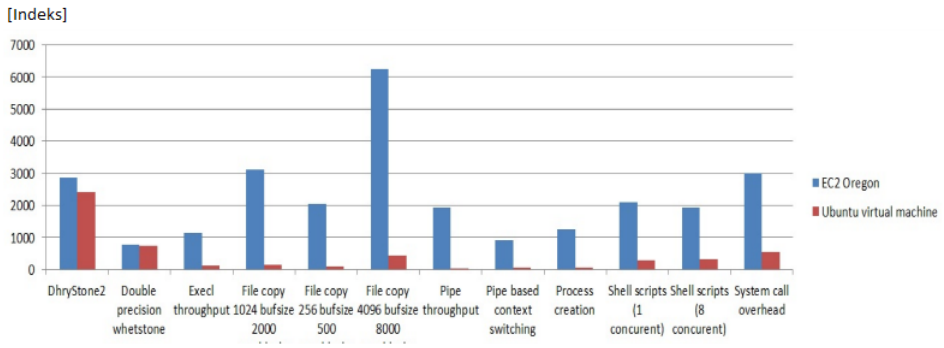
Specifikacije dodatnega sistema so sledeče:

- CPU: Intel Pentium B960, 2.20 GHz,
- RAM: 8GB,
- operacijski sistem: Windows 8.1 in
- virtualni operacijski sistem: Ubuntu 14.04 LTS - 2 jedri - 4GB RAM.

Specifikacije dodatnega sistema so podobne specifikacijam instanc na Amazonu, zato je mogoče predvideti, da bodo rezultati testov pomnilniških operacij boljši na oblačni instanci. Testi povezani s procesorsko zmogljivostjo bi morali doseči približno enake rezultate na obeh sistemih.

UnixBench zažene sledeče teste:

- Dhrystone - Osredotočen je na delo z nizi in ne analizira operacij s plavajočo vejico.

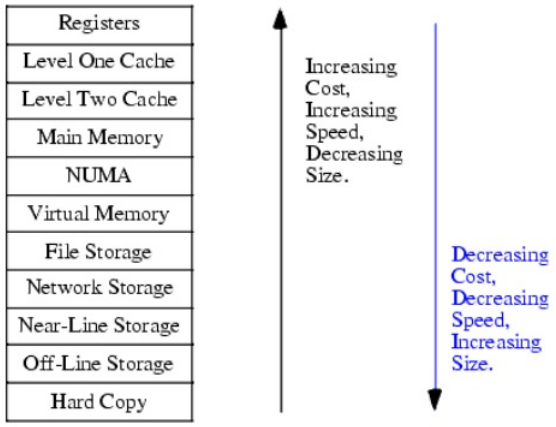


Slika 2.6: UnixBench - rezultati.

- Whetstone - Meri hitrost in učinkovitost operacij s plavajočo vejico.
- Execl thourghput - Meri število 'execl' klicev na sekundo ('execl' pripada 'exec' družini ukazov).
- File copy - Meri hitrost prenosa podatkov iz ene datoteke v drugo.
- Pipe Throughput - Kolikokrat v sekundi lahko proces zapiše in prebere 512 B iz cevovoda.
- Pipe-based Context Switching - Meri število izmenjav podatka (tipa 'integer') med procesi.
- Process creation - Meri kolikokrat lahko proces ustvari otroka in ga ubije, pri tem se otroku alocira tudi pomnilnik.
- Shell scripts - Meri kolikokrat v minuti lahko proces ustvari sočasno kopijo skripte, ki opravlja transformacijo datoteke.
- System call overhead - Ocenjuje ceno dostopa do jedra, kjer se repetitivno izvaja ukaz 'getpid'. Za ocenjevanje je uporabljen izvajalni čas.
- Graphical Test – Preprosti 2D in 3D grafični testi.

Spodnji graf 2.6 prikazuje število doseženih točk. Rezultat potrjuje našo domnevo. Tudi testi sistemskih klicev in operacij so na PaaS instanci dosegli boljše rezultate. Testi z nizi in operacijami s plavajočo vejico kažejo, da tudi preprost, enostaven sistem konkurira PaaS storitvi.

Rezultati kažejo, da obstajajo manjše performančne razlike med izbranimi lokacijami. Predvidevamo, da so razlike nastale zaradi različnih obremenitev strežnikov v času testiranja. Ker gre za javno storitev ne moremo nikoli izvesti testa, ki bi nam za vse lokacije zagotavljal enake okoliščine testiranja. Iz tega sledi da performančne razlike niso dejavnik, ki vplivajo na izbiro lokacije za postavitev nadzornega sistema.



Slika 2.7: Skica oblačne pomnilniške arhitekture [21].

PaaS storitev je priporočljivo uporabiti ko izvajamo 'big data' [20] analizo, pri kater potrebujemo veliko dostopov do vseh nivojev pomnilniškega sistema.

2.3.3 Shranjevanje podatkov

Zaradi vse večjega povpraševanja po shranjevanju podatkov, je potrebno vpeljati nove protokole za dostop do podatkov in implementirati rešitve, ki omogočajo shranjevanje datotek v oblaku. Uporabljajo se sledeči API protokoli: SOAP (Simple Object Access Protocol), REST (Representational State Transfer) in FASP (Fast and Secure Protocol). Lokalno hranjenje podatkov je bločno orientirano. Datoteko razdelimo na bloke specifičnih velikosti, ki jih shranimo na disk. Shranjevanje v oblaku je objektno orjentirano na višjem nivoju.
Zanimiva je pomnilniška hiearhija (Slika 2.7), ki vpeljuje nove elemente: NUMA (Non-uniform memory access), Network storage in Near-Line storage [21].

- NUMA (Non-uniform memory access) [22] je pomnilniška tehnologija, kjer imajo vsa procesorska jedra hiter lokalni pomnilnik, vendar so pomnilniška vozlišča pri tem dostopna dlje oddaljenim jedrom. Potreba po takšnem pristopu se je pojavilja v 60. letih, ko hitrost dostopa do pomnilnika ni več zadostovala procesorju. NUMA rešuje problem večjedrnega dostopa do istega dela pomnilnika, kjer v klasični arhitekturi ostala jedra stradajo. Problem je rešen s pomočjo dodatne strojne in programske opreme. Posebna implemetacija takšnega pristopa je ccNUMA (cache coherent NUMA), kjer imajo vsa jedra konsistento pomnilniško sliko v cache pomnilniku, kar zahteva veliko nepotrebne sinhronizacije. Čeprav je standardna NUMA arhitektura enostavnejša, je pisanje v njej kompleksnejše kot pri ccNUMA arhitekuturo.

- Network-attached storage (NAS) [23] je shranjevanje podatkov na sekundarni disk v omrežju. Različni uporabniki lahko dostopajo do podatkov z različnimi protokoli, kot sta SBM [24] in NFS [25].
- Storage area network (SAN)[26] omogoča shranjevanje v omrežju, vendar je datotečni sistem alociran na lokalnih napravah, za razliko od NAS arhitekture, kjer je celoten datotečni sistem alociran na oddaljenem strežniku.
- Near-Line storage sistem[27] predstavlja kompromis med network storage in offline storage sistemom. Takšen pristop povečuje hitrost mrežnim sistemom.

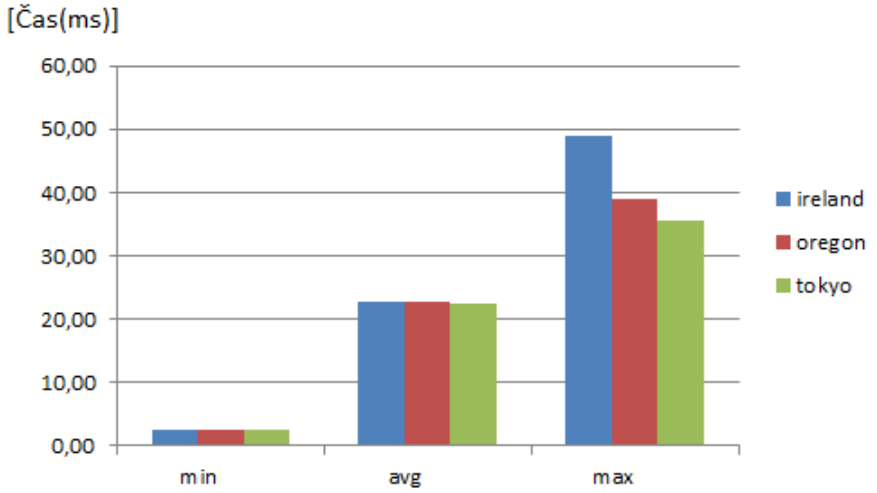
Centralni nadzorni sistem uporablja podatkovne baze za analizo podatkov. Za testiranje smo izbrali paket testov Sysbench [28]. Ta modularen paket, je namenjen ocenjevanju sistema (strojnega in progamskega dela) z podatkovnimi bazami.

Sestavljen je iz petih testov:

- CPU: Računanje praštevil.
- THREADS: Testiranje hitrosti razporejevalnika opravil v primeru velikega števila niti, ki tekmujejo za isti vir.
- MUTEX: Testiranje učinkovitost ključavnic.
- FILEIO: Testiranje različnih I/O operacij. Test pripravi določeno količino podatkov nad katerimi izvede določeno operacijo.
- OLTP: Testiranje hitrosti postavljene podatkovne baze.

Ukazi, ki smo jih uporabili za zaganjanje testov:

- CPU: sysbench –test=cpu –num-threads=8 –cpu-max-prime=20000 run
- THREADS: sysbench –num-threads=64 –test=threads –thread-yields=2000 –thread-locks=5 run
- MUTEX: sysbench –test=mutex –num-threads=2000 –thread-yields=4000 –thread-locks=36 run
- FILEIO:
  sysbench –num-threads=8 –test=fileio –file-total-size=3G –file-test-mode=rndrw prepare
  sysbench –num-threads=8 –test=fileio –file-total-size=3G –file-test-mode=rndrw run
  sysbench –num-threads=8 –test=fileio –file-total-size=3G –file-test-mode=rndrw clanup
- OLTP: nismo zagnali, saj na izbranih instancah ni naložena podatkovna baza

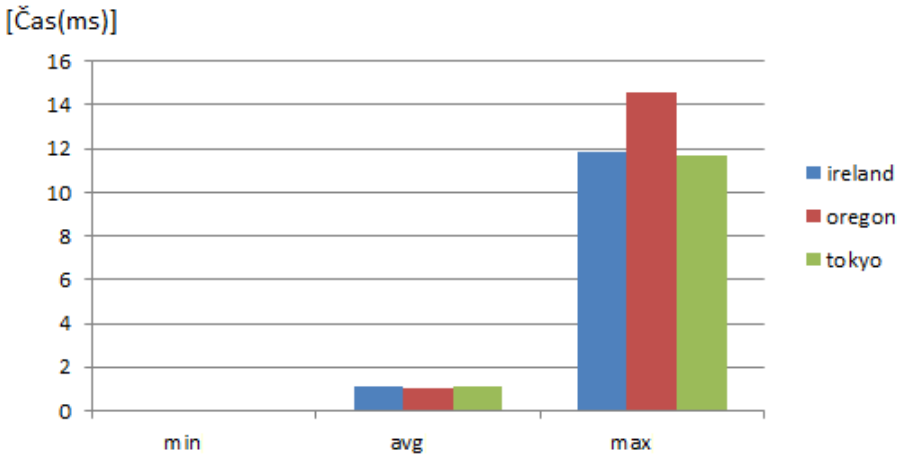


Slika 2.8: Sysbench CPU benchmark.

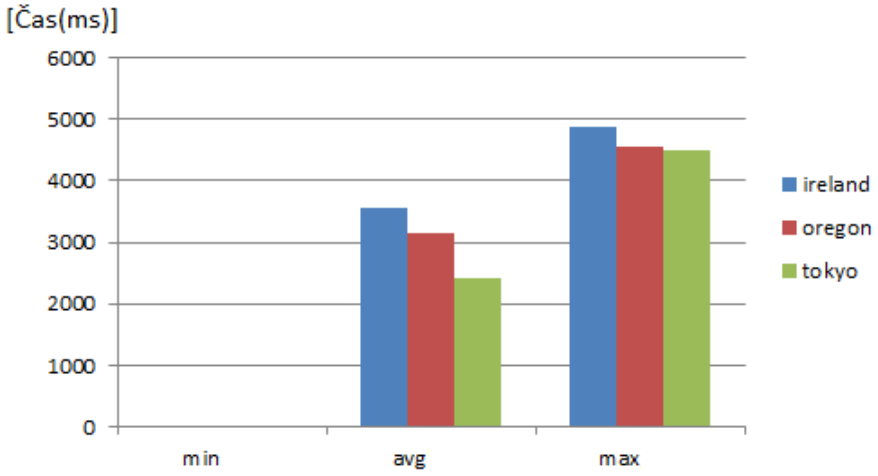
Rezultati kažejo, da obstajajo minimalne razlike med posameznimi sistemi. V grafih so za vsak test prikazani minimalni, povprečni in maksimalni časi za izvedbo posamezne zahteve.
Povprečni časi se najbolj razlikujejo pri MUTEX testu. Tokijo pa ima najboljše delovanje ključavnic.

2.4 Zaključek

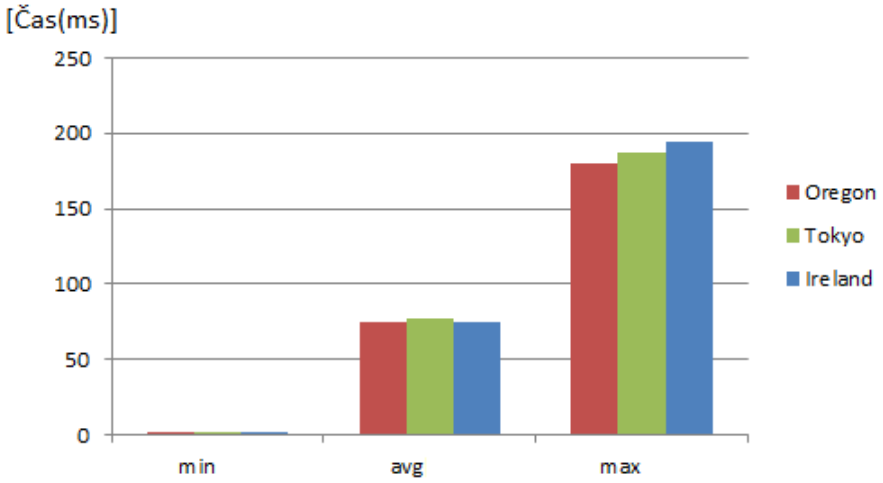
Pri testiranju instanc izbranega IaaS ponudnika smo pokrili tri večja področja: prenos podatkov, obdelavo podatkovi in shranjevanje podatkov.
Prenos podatkov je močno vezan na lokacijo nadzornega sistema. Za optimalen čas prenosa podatakov moramo minimizirati razdaljo med enotami in se izogniti času največje obremenjenosti mreže. Najkrajši čas smo izmerili na relaciji Irska - Ljubljana ob 8:00 uri. Obdelava in shranjevanje podatkov se med instancami minimalno razlikuje. Razlike lahko pripišemo različnim obremenitvam sistema. Pomemben parameter pri optimizaciji nadzornega sistema je tudi stabilnost instance. Zaželjeno je, da ima programska oprema vedno na raspolago zadosti procesorskega časa za obdelavo podatkov. Variacija razpoložljivega procesorskega časa lahko povzroči povečan odzivni čas, kar je pri nadzornih sistemih



Slika 2.9: Sysbench FILEIO benchmark.



Slika 2.10: Sysbench MUTEX benchmark.



Slika 2.11: Sysbnch THREADS benchmark.

nezaželjeno. Da bi se izognili takšnim pojavom moramo zagotoviti stalno razpoložljivost procesorja.
Iz vidika odzivnosti sistema oblačne storitve niso optimalna rešitev.

Poglavje 3

Zmogljivostna primerjava Linux in FreeBSD operacijskih sistemov

Klemen Ferjančič, Matic Pajnič, Miha Pinterič

3.1 Uvod

Tema pričujočega poglavja je testiranje zmogljivosti dveh virtualnih strežnikov z različnimi operacijskimi sistemi Linux (Debian 7) in Unix (FreeBSD 10) na izbranem oblačnem ponudniku. Maskoti obeh operacijskih sistemov lahko vidimo na sliki 3.1.Testirali smo osnovne zmogljivosti sistemov, zmogljivosti lokalnega in medcelinskega omrežja, ter omrežne protokole za prenos podatkov.

Slika 3.1: Maskota Linux-a pingvin Tux in maskota FreeBSD-ja demon (daemon) Beastie.

3.2 Ponudnik

Odločili smo se za ponudnika DigitalOcean [29] in sicer za najcenejšo različico virtualke (5€ na mesec) z naslednjimi značilnostmi:

- 512MB RAM,
- 1 jedro procesorja,
- 20GB SSD Disk,
- 1TB prenosa.

Dejavniki, ki so vplivali na izbiro, so sledeči:

- promocijske kode, ki so nam omogočile brezplačen najem virtualk za obdobje dveh mesecev (10€ kredita),
- ugodna urna postavka, ki omogoča zaganjanje virtualk po koncu brezplačnega obdobja,
- enostaven dostop do vmesnika za upravljanje z virtualkami,
- podpora Linux in FreeBSD,
- strežniki na treh celinah (Severna Amerika, Evropa, Azija) za preprosto testiranje hitrosti omrežja.

3.3 Cilji

Ugotoviti želimo, kateri operacijski sistem se nam bolj splača uporabiti glede na različne storitve, ki jih želimo uporabljati v oblaku. Najprej smo na osnovnih

različicah obeh sistemov pognali standardne teste za merjenje hitrosti procesorja, datotečnega sistema (diska) ter omrežnega sklada in jih primerjali med sabo.

Nato smo smiselne teste poskušali izboljšati s spreminjanjem konfiguracije operacijskih sistemov in testov. Predvsem smo se osredotočili na optimizacije privzetih prevajalnikov, raziskali pa smo tudi možnost optimizacije datotečnih sistemov.

Naslednji cilj je bil merjenje omrežnih hitrosti znotraj enega podatkovnega centra (torej lokalne mreže) in hitrosti med centri, ki so geografsko ločeni (različne celine). Del te naloge je bil namenjen testiranju samega omrežnega sklada operacijskih sistemov, del pa hitrosti ponudnika storitev.

Za konec smo testirali še vpliv na hitrost pri uporabi HTTPS protokola napram HTTP.

Končne ugotovitve zajemajo:

- primerjavo splošnih zmogljivosti obeh sistemov,
- dokumentacijo optimizacij, ki te zmogljivosti izboljšajo,
- meritve hitrosti lokalne mreže podatkovnega centra,
- meritve hitrosti internetnih povezav med dvema fizično ločenima centroma,
- primerjavo omrežnih protokolov za prenos podatkov in vpliv enkripcije.

3.4 Namestitev strežnikov

Najeli smo dve virtualki pri ponudniku v podatkovnem centru Amsterdam 3. Iz operacijskih sistemov smo pridobili bolj točne podatke o specifikacijah sistema in tako potrdili, da sta obe virtualki enaki. Značilnosti virtualke so sledeče:

- procesor: Intel Xeon E5-2630L v2 @ 2.40GHz (1 jedro),
- čipovje: Intel 440FX- 82441FX PMC,
- pomnilnik: 1 x 512 MB RAM,
- disk: 20GB,
- omrežje: Red Hat Virtio device.

Podatki za FreeBSD:

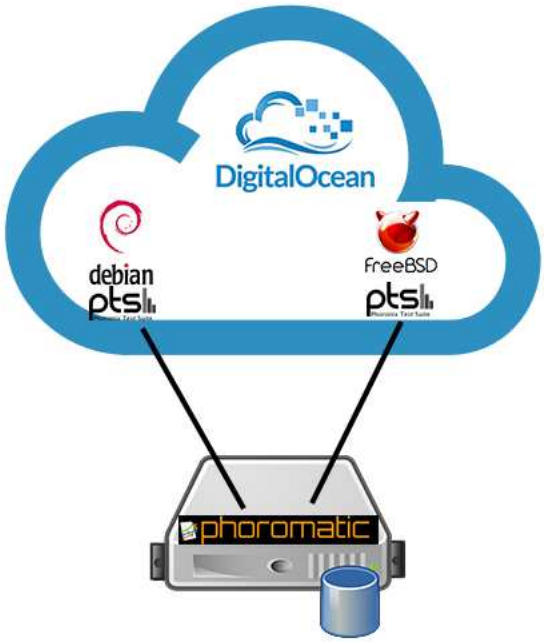
- OS: FreeBSD,
- jedro: 10.1-RELEASE (x86_64),
- prevajalnik: GCC 4.8.4 + Clang 3.4.1 (SVN 208032),
- datotečni sistem: ufs.

Podatki za Debian:

- OS Debian 7.8,
- jedro: 3.2.0-4-amd64 (x86_64),
- prevajalnik: GCC 4.7.2,
- datotečni sistem: ext4.

3.5 Namestitev testnega ogrodja

Za večjo avtomatizacijo in ponovljivost testov smo se odločili za uporabo odprtokodnega ogrodja za testiranje z imenom Phoronix test suite (PTS) [30]. PTS zajema tri komponente:



Slika 3.2: Shema celotnega sistema.

1. PTS klient, ki se namesti na sistem, ki se testira (slave);
2. Phoromatic [31], ki se namesti na ločen sistem in predstavlja spletni vmesnik za master kontrolni sistem, od tu kreiramo in poganjamo teste;
3. Openbenchmarking [32] portal, kjer se zbirajo rezultati s celega sveta; na voljo so tudi vsi opisi in seznam testov.

Več ali manj je celoten sistem odvisen le od php5-cli okolja ter nekaj php5 razširitev, zato je enostaven za namestitev.

Zagon sistema Phoromatic:

```
phoronix-test-suite start-phoromatic-server
```

Začetna namestitev za Debian:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install php5
sudo apt-get install php5-sqlite
wget http://phoronix-test-suite.com/releases/
repo/pts.debian/files/phoronix-test-suite_5.6.0_all.deb
dpkg -i phoronix-test-suite_5.6.0_all.deb
phoronix-test-suite phoromatic.connect xpam.pl:9998/DR1CTH
```

Začetna namestitev za FreeBSD:

```
mkdir -p /usr/src/crypto/openssl/util/
freebsd-update fetch
freebsd-update install
sudo pkg install ftp/wget
rehash

wget "http://www.phoronix-test-suite.com/download.php?file=
phoronix-test-suite-5.6.0"
tar xvfz download.php\?file=phoronix-test-suite-5.6.0
sudo ./install-sh
sudo pkg install php5
sudo pkg install php5-pdo_sqlite
sudo pkg install php5-dom
sudo pkg install php5-zip
sudo pkg install php5-json
sudo pkg install php5-filter
sudo pkg install php5-hash
```

Od tu naprej celotno testiranje poteka prek spletnega vmesnika Phoromatic, poseg v testne sisteme pa je potreben le ob morebitni napaki sistema ali testa. PTS smo uporabili predvsem za začetne teste. Shemo celotnega sistema lahko vidimo na sliki 3.2

3.6 CPU testi

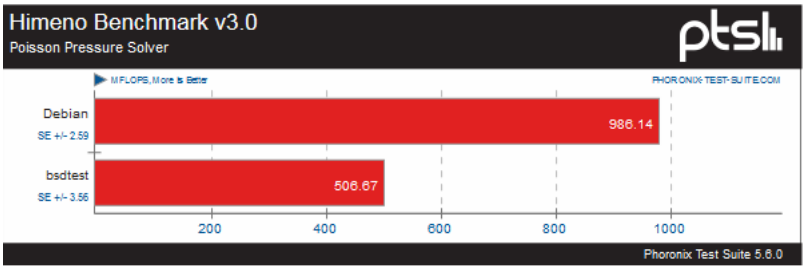
PTS testira tako, da pridobi izvorno kodo testa, uporabi sistemski prevajalnik za grajenje in nato test požene.

Ugotovili smo, da je podpora za FreeBSD pri PTS nekoliko slabša kot za Linux in nekateri testi se ne izvedejo, zato smo te pognali ročno. Vzrok je predvsem v tem, da se glavni razvijalec v glavnem osredotoča na testiranje Linux sistemov, zato se skozi čas na ostalih sistemih pojavljajo napake.

3.6.1 Začetni CPU testi

Najprej smo vse CPU teste prevedli s privzetimi nastavitvami vsakega testa in s privzetim prevajalnikom vsakega operacijskega sistema, zato da dobimo prvi vtis glede tega, kako sta operacijska sistema med seboj primerljiva. Razvijalci testov stremijo k temu, da se ti izvajajo kar najhitreje, zato so ponavadi določene optimizacije že vključene v datoteki za prevajanje (Makefile).

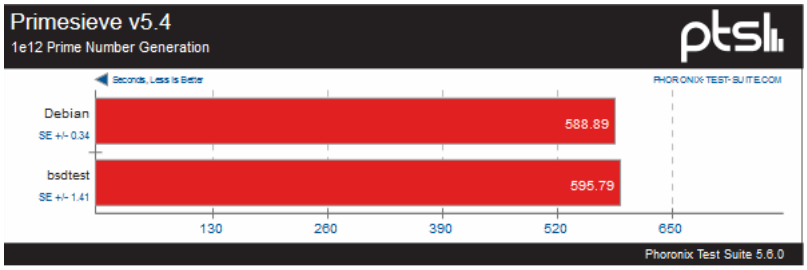
Prvi CPU test je Himeno Benchmark, ki predstavlja problem reševanja Poissonovih enačb in analize fluidov.



Slika 3.3: Reševanje Poissonovih enačb, analiza fluidov.

Kot je razvidno s slike 3.3, je Debian za skoraj 100% hitrejši od FreeBSD pri tem testu. Debian je v testu Himeno Benchmark dosegel 986.14 MFLOPS, medtem ko FreeBSD le 506.67 MFLOPS. MFLOPS je kratica za Millions of FLOating Point Operations per Second (milijon operacij v plavajoči vejici na sekundo).

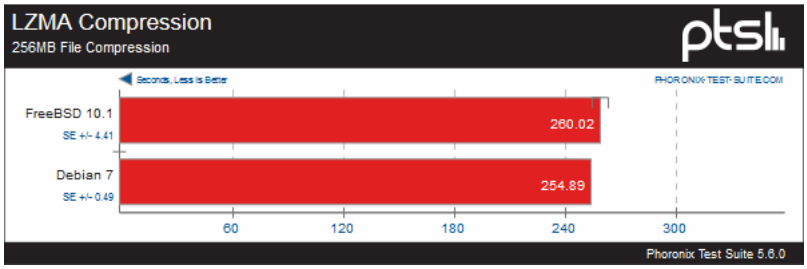
Naslednji CPU test je Primesieve, ki generira 37607912018 praštevil in meri čas, ki je bil za to porabljen.



Slika 3.4: Generator praštevil.

S slike 3.4 vidimo, da sta oba sistema zelo blizu. FreeBSD je za slabih 7 sekund oziroma za dober procent hitrejši.

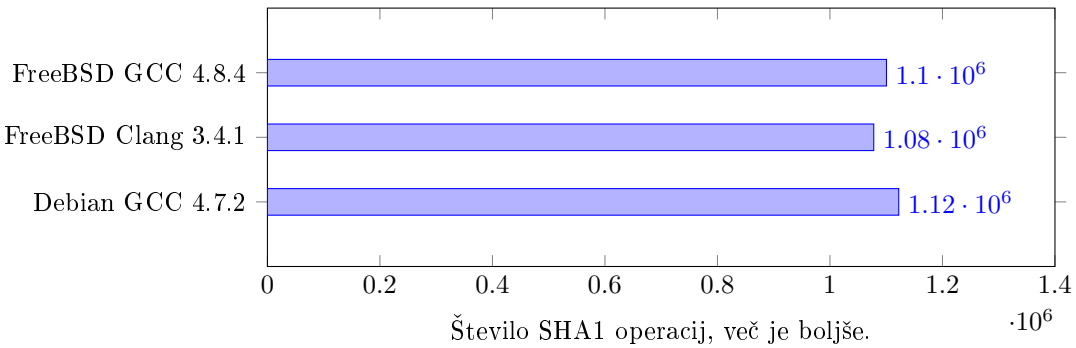
Naslednji CPU test je LZMA. LZMA je precej znan kompresijski algoritem, ki deluje po metodi izgradnje slovarja. Slovar vsebuje informacijo o tem, katera zaporedja bitov se preslikajo v določeno vrednost, cilj pa je, da se podatki z največ ponovitvami v izvorni datoteki preslikajo v čim manj bitov. Test izvede kompresijo na vnaprej določeni datoteki in meri čas.



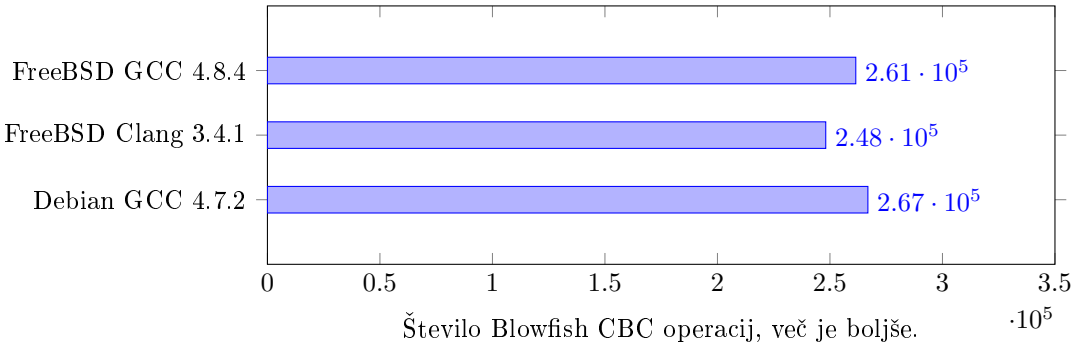
Slika 3.5: LZMA kompresija.

Kot lahko vidimo na sliki 3.5, sta tudi tu oba sistema precej blizu. Debian zmaga za slabih 6 sekund oziroma za 2%.

OpenSSL [33] je eno izmed najbolj uporabljanih odprtokodnih varnostnih orodij za kriptografijo in ma vgrajen tudi hitrostni test. Izbrali smo teste SHA1 (hash funkcija), RSA 2048bit (asimetrična kriptografija, podpisovanje) in Blowfish (simetrična enkripcija). Izvorno kodo verzije 1.0.2a smo prenesli na oba strežnika in jo zgradili s tremi različnimi prevajalniki. S tem smo želeli preveriti obnašanje GCC prevajalnika na sistemu FreeBSD in hkrati testirati z novejšo različico.

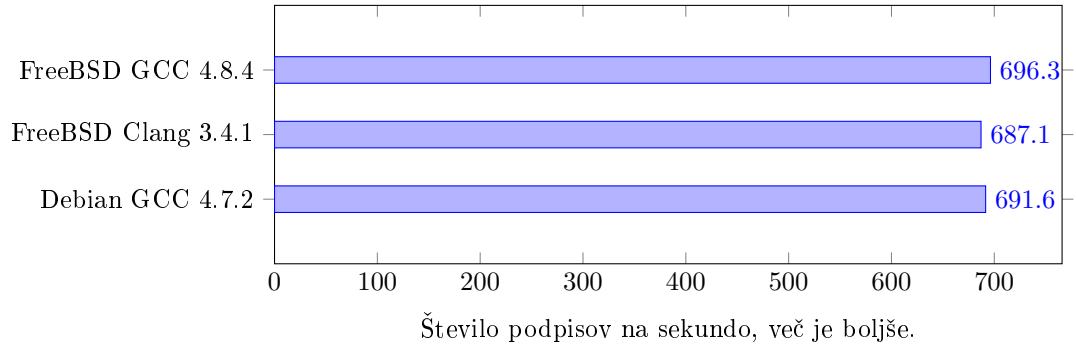


Slika 3.6: OpenSSL SHA1 test, velikost bloka 1024B.



Slika 3.7: OpenSSL Blowfish test, velikost bloka 1024B.

Pri SHA1 in Blowfish testih (vidno s slike 3.6 ter 3.7) je bil najpočasnejši FreeBSD s privzetim prevajalnikom Clang. Uporaba prevajalnika GCC je skoraj izničila zaostanek. Razlike sicer niso bile velike.



Slika 3.8: OpenSSL RSA test, podpisovanje z RSA 2048bit ključem.

Pri RSA testu so rezultati na sliki 3.8 precej izenačeni, zmagovalec je Fre-

eBSD z GCC 4.8.4.

3.6.2 Optimizirani CPU testi

Po začetnem testiranju smo želeli ugotoviti, kako na hitrost testov vpliva vključitev oziroma izključitev določenih optimizacij prevajalnikov.

Uporabili smo privzete prevajalnike (GCC 4.7.2, Clang 3.4.1) in ugotavljali spremembe glede na spreminjanje zastavic [34]:

1. -march=native: optimizira glede na arhitekturo procesorja in ukaze, ki so na voljo. Vrednost native pomeni, da prevajalnik sam ugotovi arhitekturo procesorja,

2. -O0: izklopi vse optimizacije,

3. -O1 -O2 in -O3: vsak višji nivo uporablja več tehnik za pohitritev kode. Seveda se s tem poveča tudi čas prevajanja in v večini primerov tudi velikost binarne datoteke. Točen seznam optimizacij, ki jih vključi posamezen nivo si lahko preberete v GCC dokumentaciji [34]. Clang na žalost nima te dokumentacije, lahko pa poženemo poseben ukaz, ki nam to izpiše [35].

Oba operacijska sistema imata v svojih repozitorijih tudi novejše različice prevajalnikov, vendar smo v tem primeru želeli testirati verzije, ki jih dobimo ob sami namestitvi, t.i. „out of the box“.

Za te zastavice smo se odločili iz razlage v Gentoo Wiki [36], ki je Linux distribucija znana po tem, da se vsak nov nameščen paket zgradi iz izvorne kode. Prevajalnik Clang ima kompatibilne optimizacijske nivoje z GCC, niso pa identični glede na seznam optimizacij, ki jih določen nivo vklopi [35]. Kratek opis nivojev je tudi v priročniku za FreeBSD [37].

Postopek optimiziranja in izvajanja testov je bil sledeči:

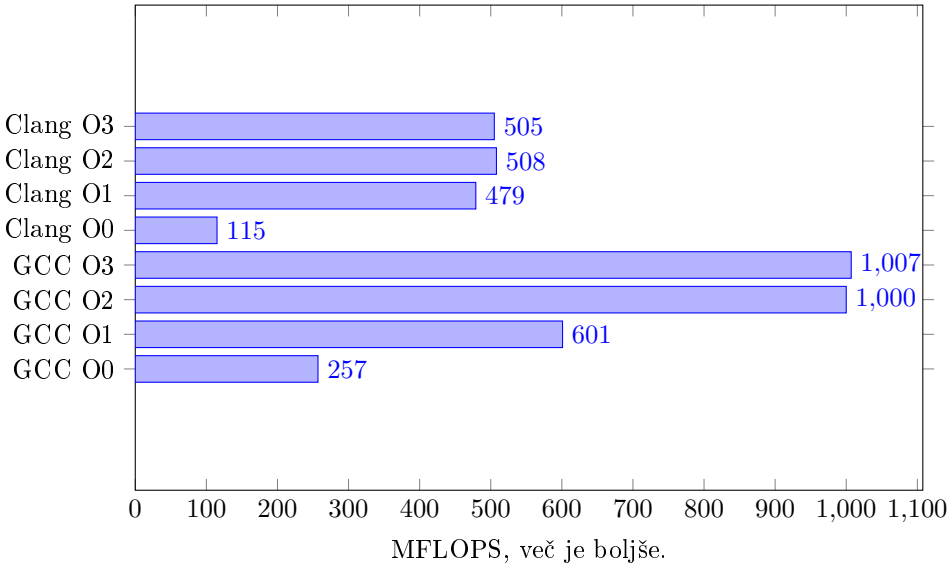
1. pridobitev izvorne kode z orodjem wget (Debian) oziroma fetch (FreeBSD),

2. zagon konfiguracije (./configure), ki pregleda CPU ukaze, ki so na voljo, ter sestavi Makefile,

3. grajenje kode in postavljanje potrebnih zastavic za določen nivo,

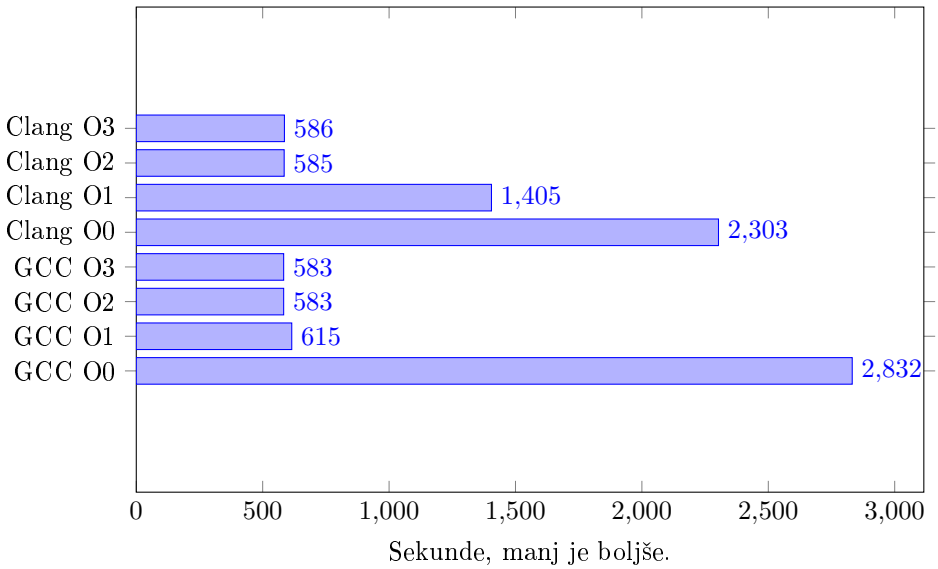
4. zagon testa,

5. ponovno grajenje za naslednji nivo optimizacije ter ponovno testiranje.



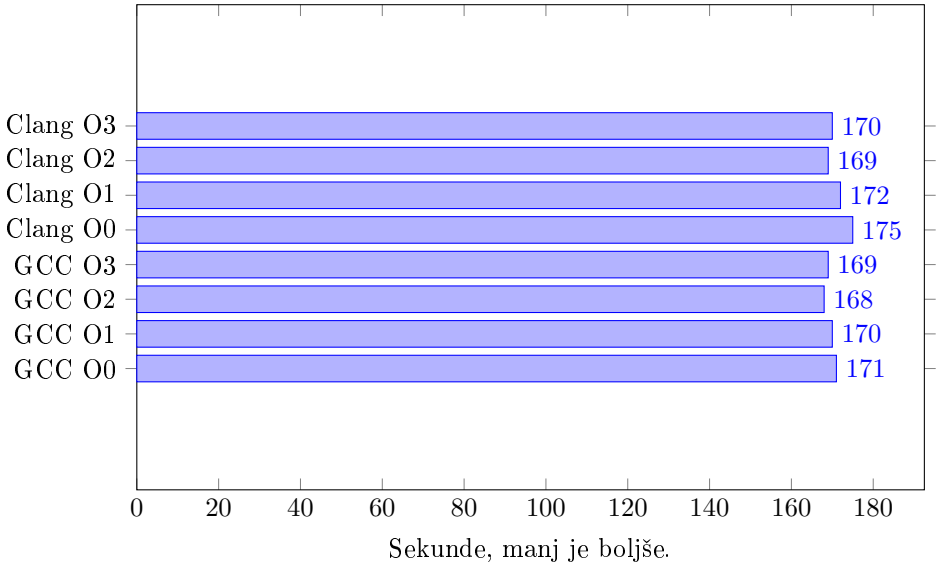
Slika 3.9: Himeno benchmark z optimizacijami.

Že pri prvem optimiziranem testu smo opazili velike razlike, kot je razvidno s slike 3.9. Vidimo, da je GCC v vseh primerih približno dvakrat hitrejši, razen pri O1. Prav tako GCC precej izboljša svoj rezultat pri prehodu iz O1 na O2, medtem ko se Clang skoraj ustavi že pri O1. Pri prehodu iz O2 na O3 pri obeh prevajalnikih ni bilo večjih pohitritev.



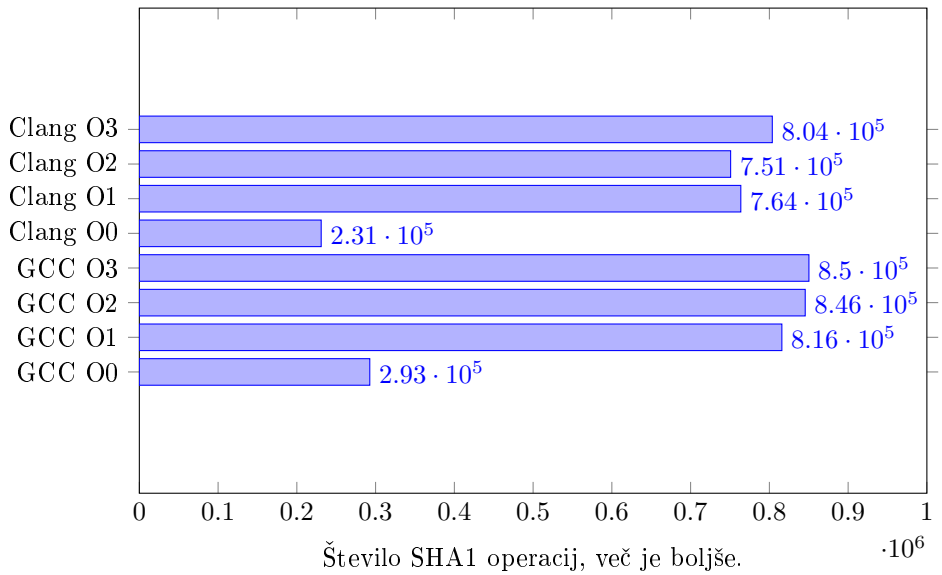
Slika 3.10: Primesieve z optimizacijami.

Slika 3.10 je malo manj zanimiva od slike 3.9. V testu Primesieve je bila največja razlika pri stopnji O1, kjer je GCC napram O0 izvajanje pohitril za 80% medtem ko je Clang izvajanje pohitril le za 40%. Pri stopnjah O2 in O3 sta se prevajalnika izenačila. Velja še omeniti, da je bil Clang pri O0 za malenkost hitrejši od GCC.



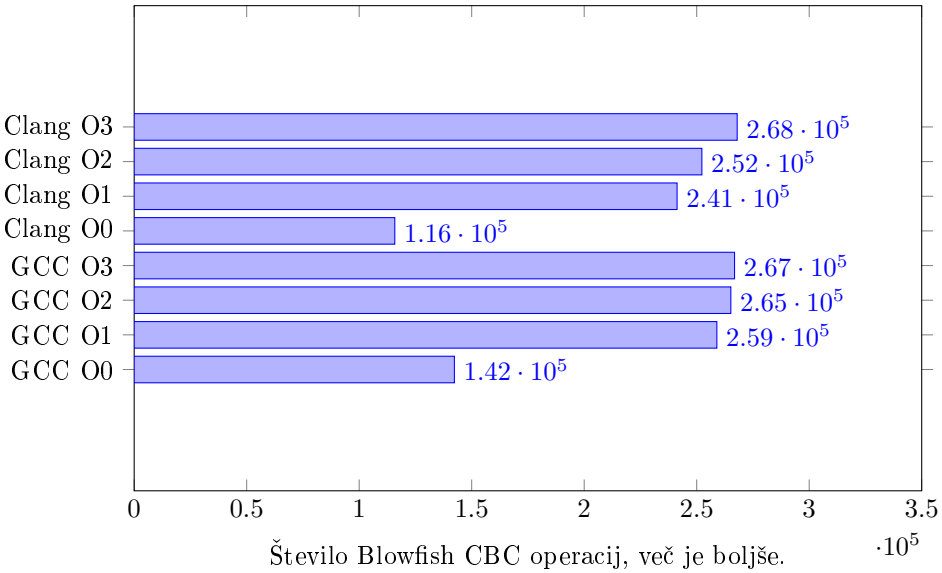
Slika 3.11: LZMA kompresija 256MB datoteke.

Kot je razvidno s slike 3.11 optimizacija LZMA testa ni prinesla večjih pohitritev.



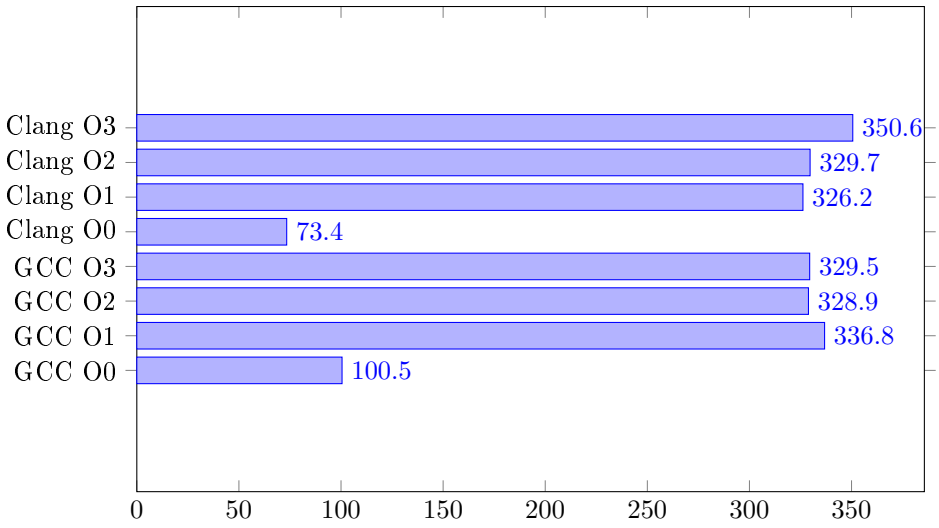
Slika 3.12: OpenSSL SHA1 hitrostni test, velikost bloka 1024B.

Optimizacija testa OpenSSL SHA1 nad stopnjo O1 ni prinesla večjih pohitritev, pri prevajalniku Clang pa se je na stopnji O2 zgodila celo regresija. V splošnem je bil prevajalnik GCC na vseh nivojih hitrejši za nekaj procentov. Rezultate vidimo na sliki 3.12.



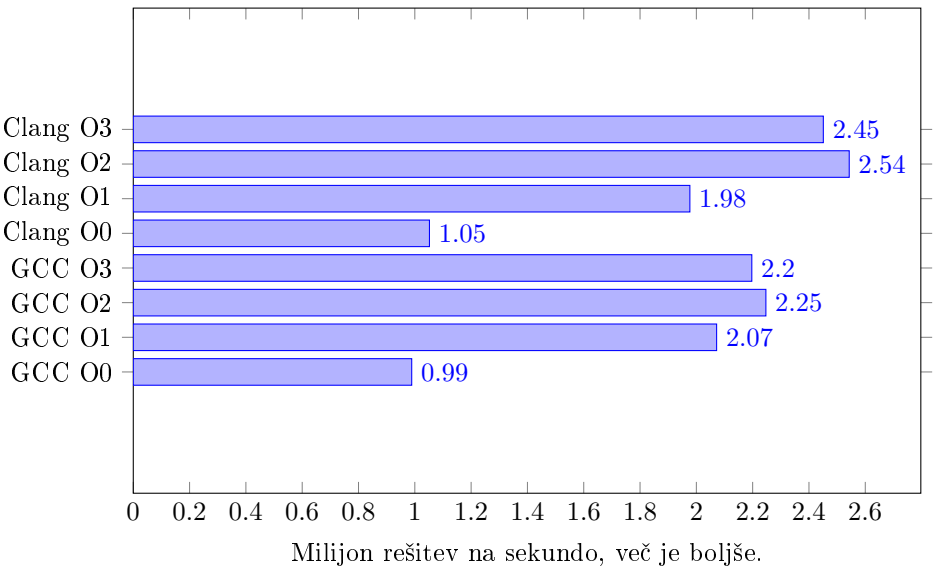
Slika 3.13: OpenSSL blowfish hitrostni test, velikost bloka 1024B.

Rezultati OpenSSL Blowfish testa na sliki 3.13 so podobni rezultatom testa SHA1. GCC je na vseh nivojih nekoliko hitrejši, regresija pri Clang nivoju O2 pa se ni ponovila.



Slika 3.14: OpenSSL RSA 2048bit hitrostni test, velikost bloka 1024B.

Pri testu OpenSSL RSA, vidnem na siki 3.14, je na nivojih O0 in O1 zmagal GCC, pri O2 in O3 pa Clang. GCC je nad nivojem O1 doživel celo manjšo regresijo.



Slika 3.15: N-Queens, polje 16x16, 16 kraljic.

Zadnji CPU test je N-Queens [38]. Gre za problem N kraljic na šahovskem polju NxN, ki jih moramo postaviti tako, da se nobena med seboj ne napada. Testu podamo velikost problema in ta nato izračuna vse možne rešitve ter meri čas. Kot lahko vidimo s slike 3.15, je bil Clang na nivojih O0, O2 in O3 hitrejši od GCC, pri O1 pa počasnejši. Oba prevajalnika sta iz prehoda O2 na O3 doživela regresijo.

3.7 Testi datotečnega sistema

Testirali smo EXT4 (Extended 4, Debian) in UFS (Unix FileSystem, FreeBSD) datotečna sistema. Oba imata na voljo določen nabor spremenljivk, ki jih lahko prilagajamo in s tem vplivamo na hitrost določenih tipov branj, pisanj, kreiranj ter brisanj datotek. Raziskali smo vpliv teh spremenljivk in možnost spreminjanja le teh, vendar smo na koncu teste pognali le na privzetih nastavitvah. Eden izmed razlogov je ta, da večina parametrov skoraj ne vpliva na hitrost [39]. Če vplivajo, je to ponavadi poslabšanje hitrosti, npr. z nastavitvijo parametra data=journal, ki pred zapisom podatkov na disk prej zapis potrdi v datotečnem dnevniku. Ena izmed nastavitev, ki v določenih primerih pohitri pisanje je nobarrier, ki pa se ne priporoča za volatilne sisteme (sisteme brez baterije), saj odstrani prepreke, ki zagotavljajo zapis podatkov na disk v vrstnem redu, kot so bili potrjeni v dnevniku. Drugi problem je ta, da na virtualkah DigitalOcean ne moremo ustvarjati novih particij, kjer bi lahko nov datotečni sistem ustvarili in mu spreminjali nastavitve, ne da bi s tem vplivali na glavni sistem. Za testiranje sprememb je datotečni sistem potrebno vsakič odmontirati, priporočeno pa je tudi vsakokratno formatiranje med testi. Takšno testiranje na root

particiji (glavna, delovna particija) je zelo težko, formatiranje pa pravzaprav nemogoče. Zaradi teh ugotovitev smo se odločili, da te optimizacije opustimo in raje poženemo nekaj več testov na privzetih nastavitvah datotečnih sistemov.

Spremenljivke za EXT4 [40], [41], ki jih lahko spreminjamo s programom tune2fs:

- barrier oziroma nobarrier: vključene prepreke zagotavljajo, da se podatki zapišejo na disk v enakem vrstnem redu, kot so bili potrjeni v dnevnik,
- data=journal: podatki in metapodatki se najprej zapišejo in potrdijo v dnevnik, šele nato na datotečni sistem,
- data=ordered: v dnevniku se hranijo le metapodatki,
- data=writeback: dnevnika se ne uporablja,
- delalloc in nodelalloc: bloki so ali alocirani vnaprej ali pa se čaka do trenutka, ko se podatki zapišejo,
- inodereadaheadblks: podano je število blokov, ki se berejo vnaprej, kar lahko pohitri sekvenčna branja.

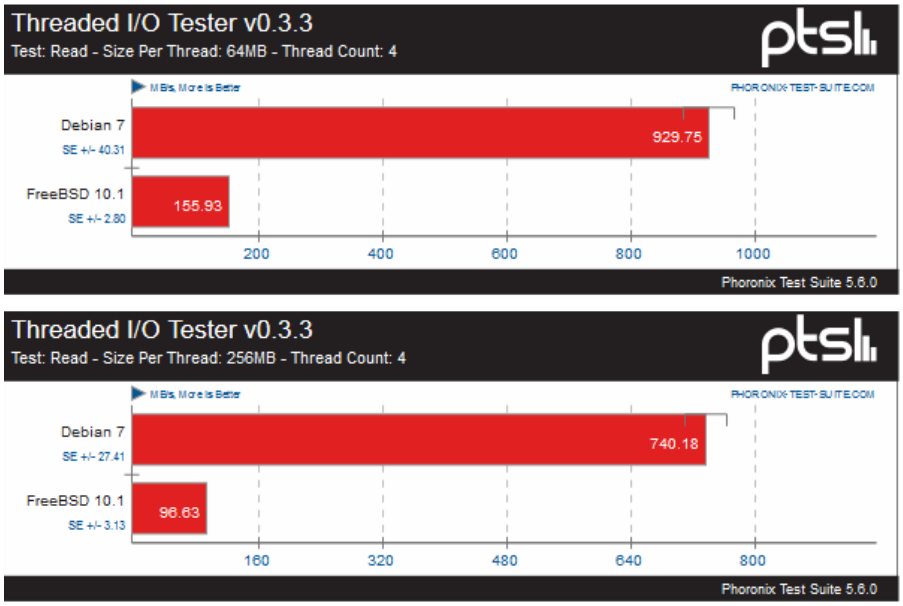
Spremenljivke za UFS [42], [43] spreminjamo z ukazom tunefs in zastavicami:

- -n: mehke posodobitve (soft updates), metapodatki se hranijo v pomnilniku,
- sistemski ukaz vfsreadmax: podano je število podatkov, ki se preberejo vnaprej, podobno kot inodereadaheadblks pri EXT4,
- -j: pomeni isto kot zastavica -n, vendar z dnevnikom.
- -o time: optimizira čas alociranja blokov namesto fragmentacije.

Za testiranje datotečnih sistemov smo uporabili več različnih orodij, predvsem pa so nas zanimali štirje ključni podatki: sekvenčno branje, sekvenčno pisanje, naključno branje ter naključno pisanje.

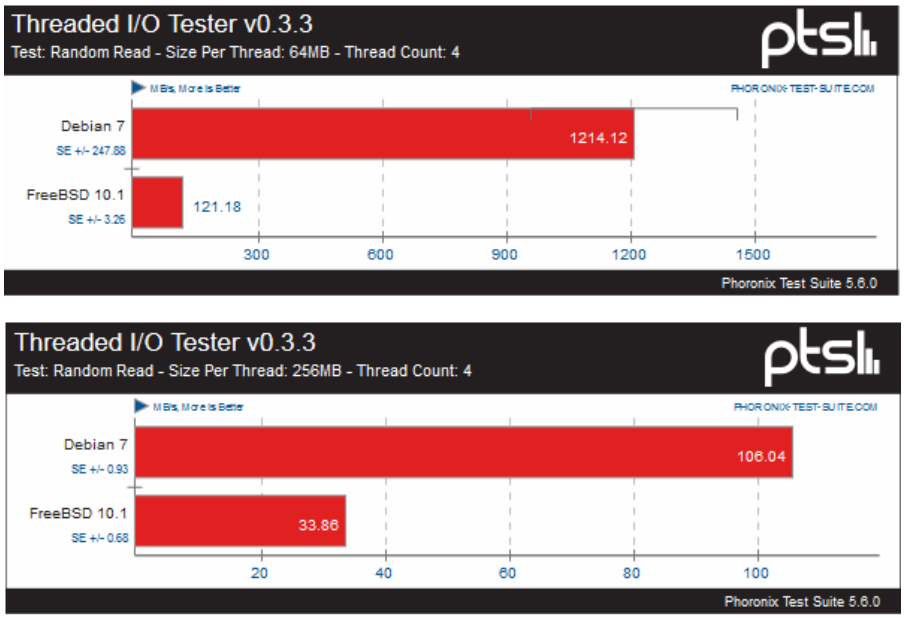
ThreadedIO je večniten test datotečnega sistema. Testira branje in pisanje različnih velikosti datotek [44]. Pri večnitnem IO testu branja (4 niti), kjer so se operacije izvajale nad različnimi velikostmi datotek (64MB in 256MB), je bil precej hitrejši Debian, kar lahko razberemo s slike 3.16. V primerjavi s FreeBSD je pri bralnih dostopih dosegel približno 6-krat večjo hitrost pri velikosti datotek 64MB oziroma približno 8-krat večjo hitrost pri velikosti datotek 256MB.

Pri pisalnih dostopih se je Debian prav tako izkazal za boljšega, so pa bile razlike med sistemoma nekoliko manjše, kar je vidno na sliki 3.17. Hitrejši je bil za 33% pri velikosti datoteke 64MB in za 47% pri velikosti 256MB.

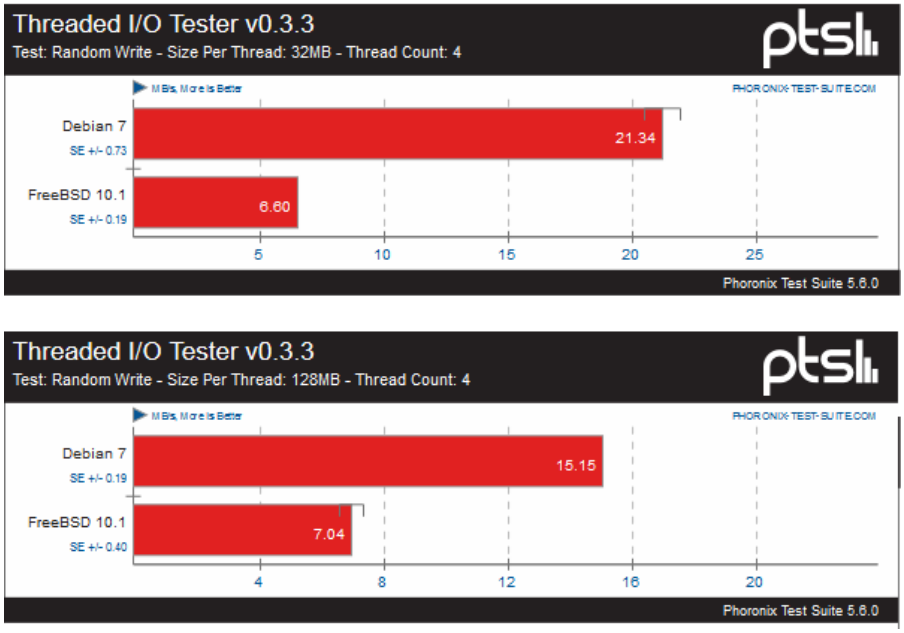


Slika 3.16: Večnitni IO test, branje, 64MB in 256MB.

Slika 3.17: Večnitni IO test, pisanje, 64MB in 256MB.



Slika 3.18: Večnitni IO test, branje, naključen dostop, 64MB in 256MB.



Slika 3.19: Večnitni IO test, pisanje, naključen dostop, 64MB in 256MB.

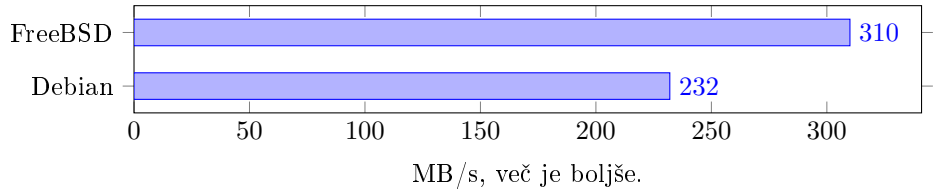
Naslednji test z uporabo ThreadedIO je enak prvemu, le da so bili tokrat dostopi naključni. Razlike med platformama so bile še večje (vidno s slike 3.18) kot pri prejšnjem testu. Debian je dosegel kar 10-krat večjo hitrost kot FreeBSD pri bralnih dostopih velikosti 64MB, pri 256MB pa več kot trikratno.

Na sliki 3.19 vidimo še pisalne naključne dostope. Debian je bil ponovno zmagovalec, več kot trikrat hitrejši pri velikosti 64MB in več kot dvakrat hitrejši pri 256MB.

DD [45] je standardno orodje v *nix okoljih (okolja z imeni s končnico nix, npr. Unix, mednje spada tudi Linux.) za prenašanje podatkov iz neke vhodne na neko izhodno napravo. Test izvede kopiranje ničel v neko izhodno datoteko. Ker ima FreeBSD svojo različico orodja, ki mu manjka opcija fdatasync, ki poskrbi za to, da se vsi podatki res zapišejo na disk pred izhodom procesa, smo na FreeBSD namestili GNU različico (paket coreutils 8.23). Ko smo test izvedli brez te zastavice je bil neizmerno hiter, ker se zapis potrdi že v pomnilniku, zato je bil za primerjavo neuporaben. Test zapiše 512MB veliko datoteko z ničlami.

Ukaz:

```
dd if=/dev/zero of=sb-io-test bs=1M count=512 conv=fdatasync
```

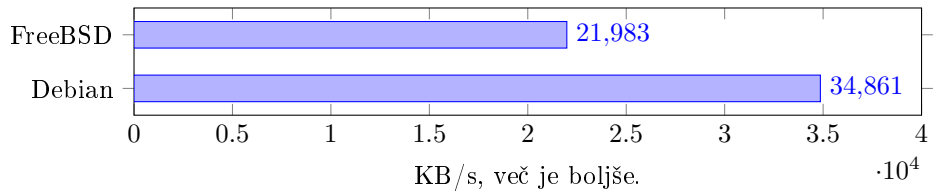


Slika 3.20: Hitrosti pisanja 512MB podatkov z orodjem dd.

Kot je razvidno s slike 3.20 je FreeBSD pri zapisu za 33% hitrejši.

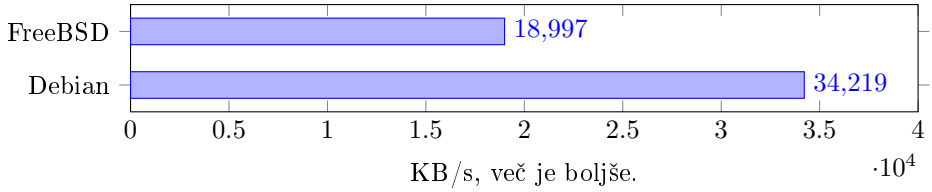
Naslednji test je FIO (Flexible IO Tester) [46]. Pognali smo ssd-test, ki ga najdemo v primerih uporabe (/usr/share/doc/fio/examples/ssd-test na Debian-u). Privzete nastavitve smo spremenili, in sicer ioengine=sync (privzeta nastavitev je libaio, ki deluje le na Linux sistemih) in directory=/ (lokacija našega root datotečnega sistema).

Prvi test je sekvenčno branje, vidimo ga na sliki 3.21. FreeBSD je bil za 37% počasnejši.



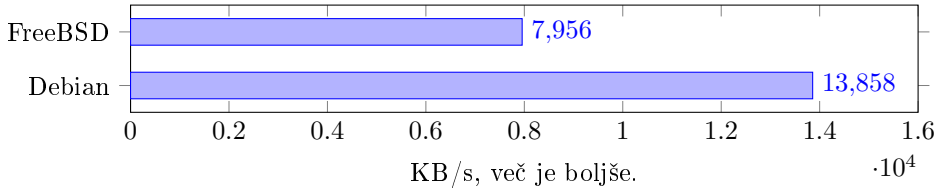
Slika 3.21: FIO sekvenčno branje.

Drugi test je naključno branje, vidimo ga na sliki 3.22. Pri tem testu je bil Debian za kar 45% hitrejši.



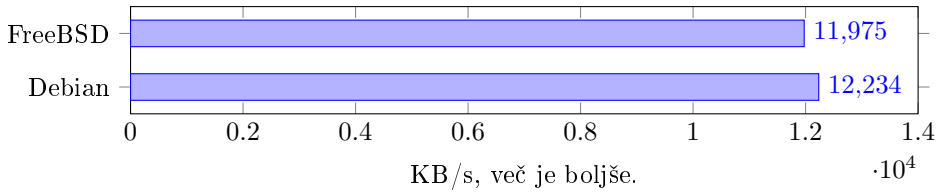
Slika 3.22: FIO naključno branje.

Tretji test je sekvenčno pisanje, vidimo ga na sliki 3.23. Tudi pri tem testu je bil Debian za kar 43% hitrejši.



Slika 3.23: FIO sekvenčno pisanje.

Četrti in zadnji FIO test je naključno pisanje, vidimo ga na sliki 3.24. Tudi pri tem testu je bil Debian hitrejši, vendar le za 3%.



Slika 3.24: FIO naključno pisanje.

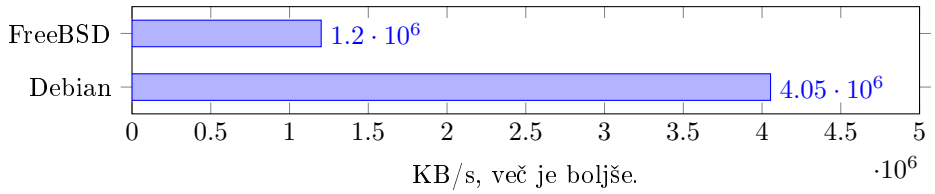
Kot zadnje testno orodje datotečnih sistemov smo uporabili IOZONE [47]. Podobno kot FIO in ThreadedIO zna meriti sekvenčno ter naključno branje in pisanje. Pognali smo ga z ukazom:

```
iozone -s 200M -i0 -i1 -i2 -e
```

, ki test izvede na 200MB veliki datoteki. Z -i zastavico določimo, kakšne vrste teste želimo izvesti (0-branje, 1-pisanje, 2-naključno branje in pisanje). Zasta-

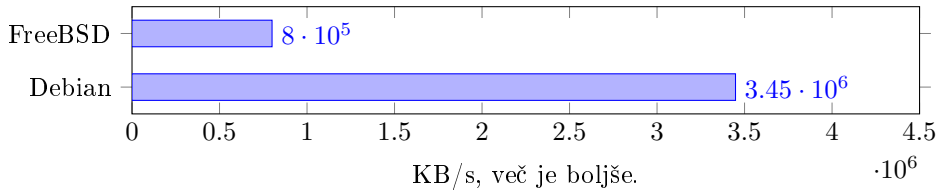
vica -e povzroči, da se res vsi podatki zapišejo na disk pred izhodom programa (podobno kot fdatasync pri dd).

Prvi test je sekvenčno branje, vidimo ga na sliki 3.25. Debian je bil pri tem testu skoraj 3,4-krat hitrejši.



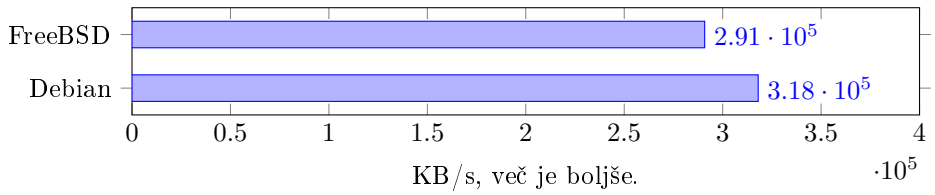
Slika 3.25: IOZONE sekvenčno branje.

Drugi test je naključno branje, vidimo ga na sliki 3.26. Pri tem testu je bil Debian kar 4,3-krat hitrejši.



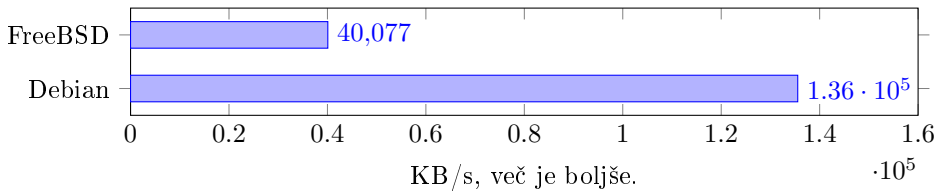
Slika 3.26: IOZONE naključno branje.

Tretji test je sekvenčno pisanje, vidimo ga na sliki 3.27. Tudi pri tem testu je zmagovalec Debian, razlika pa je manjša, le za 9%.



Slika 3.27: IOZONE sekvenčno pisanje.

Četrti in zadnji IOZONE test je naključno pisanje, vidimo ga na sliki 3.28. Debian je bil ponovno precej hitrejši in to za 3,4-krat.



Slika 3.28: IOZONE naključno pisanje.

3.8 Testi sistema kot celote

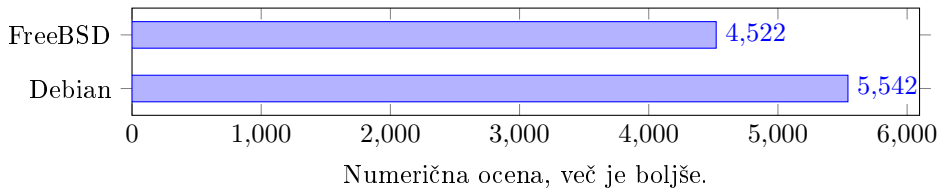
Testi sistema kot celote hkrati obremenijo več delov operacijskega sistema in v določeni meri simulirajo realne obremenitve.

Prvi tak test je BlogBench, ki simulira blog. Simulacija vključuje naslednje funkcije oziroma aktivnosti: dodajanje novih objav, posodabljanje objav, dodajanje slik in komentarjev [48].

Slika 3.29: Blogbench.

Rezultat testa BlogBench je presenetljiv (viden s slike 3.29), glede na to, da je bil FreeBSD pri testih datotečnega sistema precej počasnejši, tu pa je hitrejši, čeprav BlogBench najbolj testira prav disk oziroma podatkovno bazo.

Naslednji test je PHPBench [49], ki testira hitrost izvajanja nekaterih PHP funkcij (rand, string_append, strlen) in iz tega izoblikuje neko končno oceno. Kot vidimo s slike 3.30 je bil Debian boljši za približno 1000 točk. V času je to pomenilo 180 sekund proti 221 sekundam, torej 41 sekund hitrejše izvajanje.

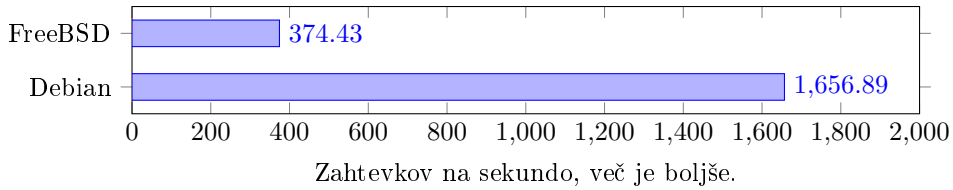


Slika 3.30: PHPBench.

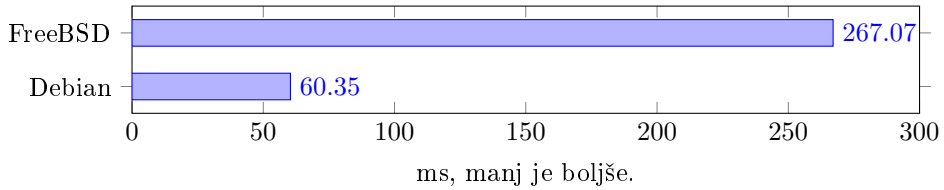
Zadnji izmed celovitih testov je Apache Benchmark (ab) [50]. Na oba strežnika smo namestili Apache 2 strežnik in testirali zmogljivost v localhost načinu (strežnik prejema zahtevke od samega sebe). Ukaz je sledeč:

```
ab -r -n 5000 -c 100 http://IP/
```

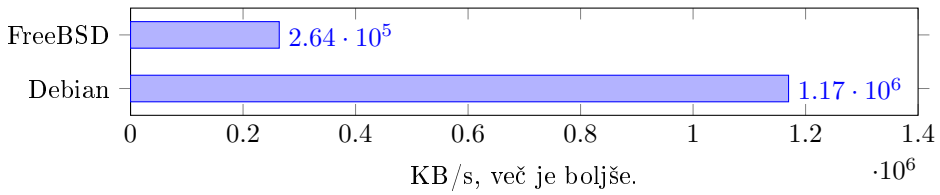
, kjer zastavica -r pomeni, da ob morebitni napaki nadaljujemo (neizvedeni zahtevki se izpišejo na koncu testa, vendar jih pri našem testiranju ni bilo), -n nam pove skupno število zahtevkov, -c pa koliko zahtevkov je poslanih naenkrat (concurrent). Kot rezultat dobimo število zahtevkov obdelanih na sekundo, povprečen čas enega zahtevka ter samo hitrost pretakanja podatkov. Oba strežnika sta stregla identično statično HTML stran velikosti 702KB. Na sistemu Debian je bila uporabljena verzija Apache 2.2, na FreeBSD pa 2.4, oba s privzetimi nastavitvami. Lahko bi sicer prevedli enako verzijo Apache-ja iz kode, vendar smo želeli uporabiti pakete, ki jih ponujata sistema v svojih uradnih repozitorijih. To je tudi najpogostejši način, ki se ga bo povprečen uporabik poslužil za namestitev.



Slika 3.31: AB število zahtevkov na sekundo.



Slika 3.32: AP povprečen čas ene zahteve.



Slika 3.33: AB povprečna hitrost prenosa.

Kot lahko vidimo iz slik 3.31, 3.32 in 3.33, je Debian pri privzetih nastavitvah spletnega strežnika uspel obdelati precej več zahtev v sekundi, to storil v krajšem času in seveda imel večjo hitrost prenosa.

3.9 Testiranje omrežnih hitrosti

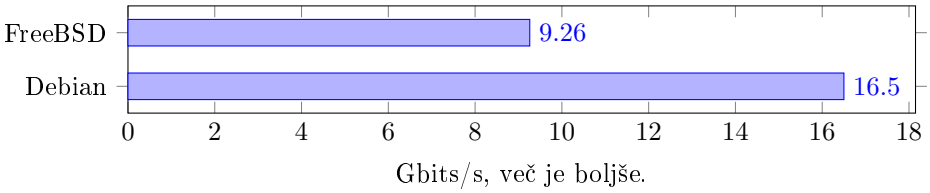
3.9.1 Hitrost loopback

Za merjenje hitrosti omrežja smo uporabili orodje Iperf [51]. Poženemo ga enkrat kot strežnik in drugič kot klient. Za merjenje hitrosti loopback-a (pretok skozi localhost na istem strežniku) smo iperf pognali kot strežnik, nato pa v novi SSH seji še kot klient.

Ukazi:

```
iperf -s
iperf -c IP -d
```

Zastavica -s požene strežnik. Zastavica -c požene klienta, ki se poveže na IP strežnika, prenos testira obojestransko (bidirectional, zastavica -d).



Slika 3.34: Hitrosti loopback adapterja.

Pri testiranju loopback omrežne hitrosti (vidno s slike 3.34) je bil Debian za kar 36% hitrejši. Glede na to, da imata oba sistema enako mrežno strojno opremo, je test več ali manj odvisen samo od TCP/IP omrežnega sklada in gonilnikov.

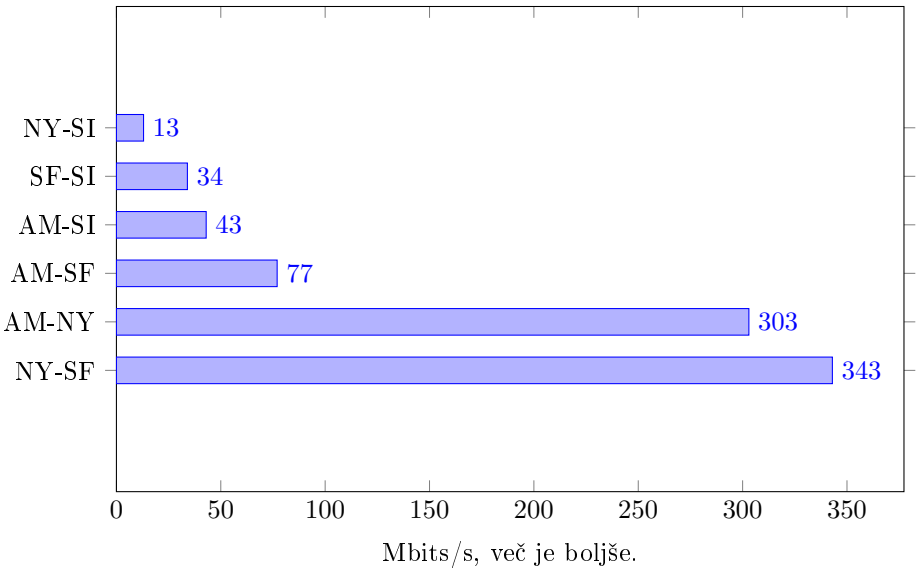
3.9.2 Omrežne hitrosti znotraj podatkovnega centra

Oba strežnika sta najeta v istem podatkovnem centru, zato sta praktično povezana preko LAN omrežja, po vsej verjetnosti preko stikal. Izmerili smo hitrosti

v primeru da je Debian strežnik in FreeBSD klient ter obratno. V obeh primerih so bile hitrosti zelo podobne, in sicer okrog **930 Mbits/s**, kar je precej blizu gigabitni hitrosti in pomeni, da več ali manj izkoriščamo vso strojno hitrost gigabitnega omrežnega vmesnika.

3.9.3 Omrežne hitrosti med geografsko ločenimi podatkovnimi centri

Najeli smo še tri Debian virtualke in sicer na lokacijah New York, San Francisco ter Singapur. Hitrost internetnih povezav smo izmerili z orodjem Iperf na enak način kot LAN hitrosti in sicer na način vsak z vsakim. Za Evropskega predstavnika smo uporabili že obstoječ strežnik v Amsterdamu. Pri tem testu je zagotovo v ozadju precej spremenljivk na katere težko vplivamo: čas meritev (ponoči, podnevi), zasedenost omrežnih povezav, izpad povezav, razlika med ponudniki hrbtenice, ki jih DigitalOcean uporablja v posameznih centrih in še mnogo drugih. Tudi med posameznimi testi je prihajalo do precejšnjih razlik, kljub temu, da smo jih izvedli večkrat in v zelo kratkih presledkih. Kljub vsem tem vplivom pa lahko rezultate jemljemo kot nek grob pogled na to, kaj lahko pričakujemo od določenih povezav pri tem ponudniku.

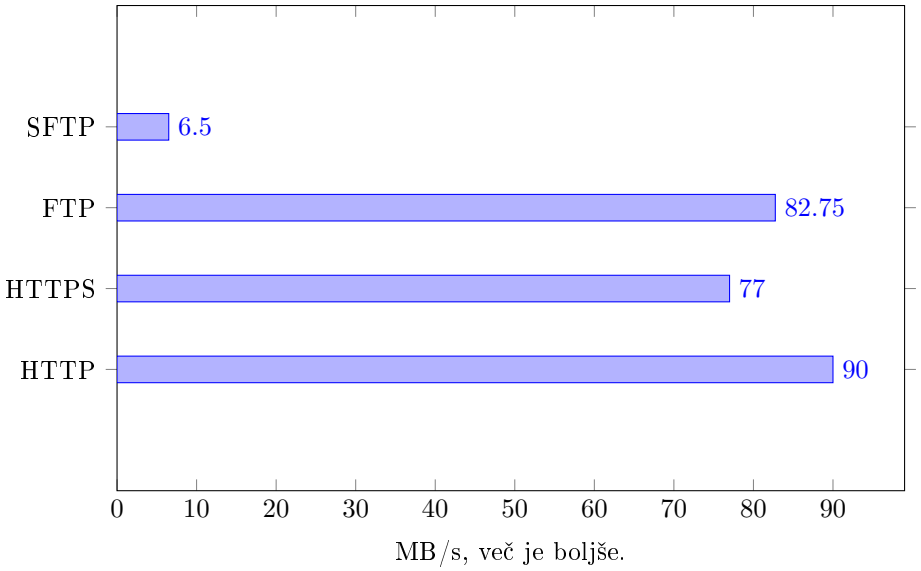


Slika 3.35: Omrežne hitrosti med kontinenti.

Kot je razvidno s slike 3.35 je bila najhitrejša povezava med obalama Severne Amerike (NY-SF), sledi medatlantska povezava med Amsterdamom in New Yorkom (AM-NY), nato Amsterdam-San Francisco (AM-SF), Amsterdam-Singapur (AM-SI), najslabše hitrosti pa so bile dosežene med Severno Ameriko in Azijo (SF-SI ter NY-SI).

3.9.4 Hitrost prenosa podatkov različnih omrežnih protokolov ter vpliv na hitrost pri uporabi enkripcije

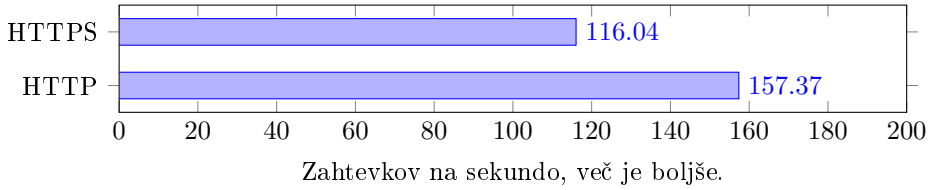
Namen teh testov je ugotoviti, kateri protokol se nam najbolj splača uporabljati, če želimo prenašati večje datoteke preko omrežja, na primer za rezervno kopijo (backup). Debian smo uporabili kot strežnik, FreeBSD pa kot klient, ki na svoj sistem prenaša datoteke. Kot datoteko za prenašanje smo uporabili arhiv izvorne kode spletnega strežnika Apache. Uporabljeni protokoli so: SFTP (SSH File Transfer Protocol), FTP (File Transfer Protocol), HTTP (HyperText Transfer Protocol) ter HTTPS (kriptiran HTTP).



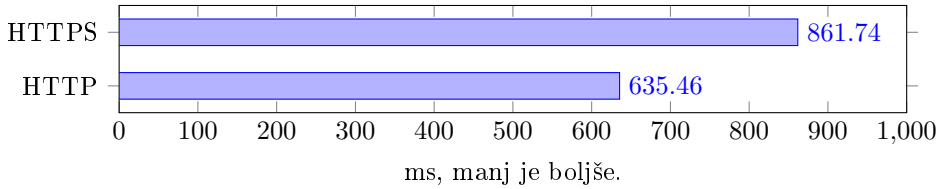
Slika 3.36: Hitrost prenosa datoteke z različnimi omrežnimi protokoli.

Kot lahko vidimo na sliki 3.36 je najhitrejši HTTP, nato FTP, sledi HTTPS, čisto na koncu pa še SFTP. Rezultati niso presenetljivi, saj SFTP deluje prek SSH tunela, ki za prenose datotek ni bil v osnovi nikoli namenjen. HTTPS je seveda nekoliko počasnejši od HTTP, saj se podatki kriptirajo. Še najbolj zanimiv rezultat je ta, da je HTTP hitrejši od FTP, ki že iz samega imena pove, da je namenjen prenosu datotek.

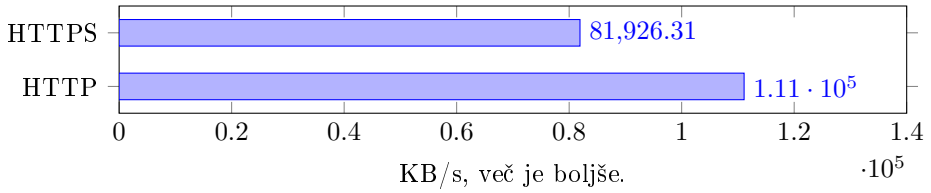
Primerjava med HTTP in HTTPS hitrostjo nam da že slutiti rezultate našega zadnjega testa, kjer smo merili vpliv enkripcije na hitrost spletnega strežnika. Test logično združuje nekatere prejšnje: OpenSSL RSA, Apache Benchmark ter že omenjeno primerjavo protokolov. Za testiranje vpliva enkripcije na hitrost smo ponovili Apache Benchmark test iz poglavja 3.8, le da smo tokrat kot naslov uporabili https in ne http. Seveda smo še prej Apache strežnik skonfigurirali za podporo SSL in generirali samopodpisan RSA certifikat.



Slika 3.37: AB število zahtevkov na sekundo.



Slika 3.38: AB povprečen čas ene zahteve.



Slika 3.39: AB povprečna hitrost prenosa.

Kot lahko vidimo na slikah 3.37, 3.38 in 3.39, se pri uporabi protokola https število obdelanih zahtev, povprečen čas ene zahteve ter povprečna hitrost prenosa poslabšajo za 27%. V realnem svetu to pomeni, da če želimo na svoji spletni strani uporabljati https, moramo zakupiti za 27% hitrejšo strojno opremo ali je vsaj toliko imeti še v dobrem, če želimo, da nam strežnik zmore obdelati enako število uporabnikov kot pri http.

3.10 Zgodovina uporabljenih operacijskih sistemov, prevajalnikov in datotečnih sistemov

Pri CPU testih je večkrat zmagal Debian, zahvala pa gre predvsem prevajalniku GCC. GCC je že zelo star prevajalnik z začetki, ki segajo v konec osemdesetih let prejšnjega stoletja. Razvoj vodi organizacija FSF-Freedom Software Foundation in je de-facto najbolj uporabljan prevajalnik v Linux operacijskih sistemih [52].

Clang je po drugi strani še precej nov prevajalnik, začetki segajo v leto 2005. Clang je privzeti prevajalnik na operacijskem sistemu FreeBSD postal šele z izdajo, ki smo jo testirali, torej 10.0, prejšnje izdaje pa so uporabljale GCC. Največje zasluge za razvoj Clang prevajalnika ima podjetje Apple [53].

Pri datotečnih testih je bil Debian razen izjem vedno hitrejši. Tu je zgodba obrnjena, saj je UFS precej starejši od EXT4 in sega v osemdeseta leta prejšnjega stoletja [54], medtem ko je bil EXT4 predstavljen šele leta 2008 kot naslednik EXT3 [55]. Sami datotečni sistemi EXT sicer segajo v začetek devetdesetih let. UFS je znan predvsem kot zelo stabilen datotečni sistem, ki nerad dodaja nove funkcionalnosti. Na obeh straneh je trenutno razvoj usmerjen v razvoj datotečnih sistemov s poudarkom na integriteti podatkov, torej zaznavanju in popravljanju t.i. obrnjenih bitov na disku (bit rot). Glavna predstavnika sta ZFS in BTRFS.

V splošnem je Linux trenutno eno izmed najbolj uporabljanih jeder v operacijskih sistemih na svetu. Uporablja se v preštevilnih namiznih in strežniških distribucijah, vgrajenih sistemih ter v mobilnem sistemu Android. Na njem dela precej več razvijalcev in ima finančno podporo v največjih računalniških podjetjih na svetu (Google, Samsung, Intel, Red Hat, IBM...). FreeBSD ima precej manj razvijalcev, najbolj znani podporniki pa so NetApp, Netflix, Juniper, Google [56]... Najbolj znan je po tem, da ga je za osnovo Mac OS uporabil Apple, največ pa se uporablja v strežniških okoljih ter usmerjevalnikih.

3.11 Zaključek

Kljub številnim testom je na koncu vedno težko potegniti nek splošen zaključek o tem, kaj naj bi bila boljša izbira. Vsak izmed operacijskih sistemov je boljši na določenih področjih in naši testi so le vrh ledene gore vseh možnih obremenitev.

Kar se tiče CPU testov je bil Debian precej hitrejši v Himeno benchmark, medtem ko sta bila oba sistema zelo blizu pri testih LZMA, OpenSSL ter Primesieve. Tudi pri optimizaciji je bila največja prednost za Debian v Himeno benchmark, prednost pa si je nabral tudi pri optimizaciji Primesieve. Na LZMA in OpenSSL teste so optimizacije vplivale minimalno. V kolikor bi želeli iztisniti kar največjo hitrost iz naših programov, bi priporočili uporabo čim novejše različice prevajalnika GCC ter optimizacijski nivo O2 ali O3. Tudi Clang opravi svoje delo solidno in je v več testih zelo primerljiv z GCC-jem.

Glede datotečnih sistemov je EXT4 po našem mnenju boljša izbira, saj je pri več testih za večkratnik prehitel UFS. Pri sekvenčnem in naključnem branju je bil Debian pri vseh testih absolutno hitrejši, tudi za večkratnik. Velike razlike v prid Debianu so bile tudi pri naključnem pisanju, medtem ko so bile pri sekvenčnem pisanju manjše.

Pri celostnih testih je z velikim vodstvom v Apache benchmark zmagal Debian, prav tako z občutno razliko v PHPBench, medtem ko je FreeBSD z veliko prednostjo zmagal v testu Blogbench.

Pri omrežnih hitrostih so bili rezultati precej predvidljivi. V kolikor moramo prenašati veliko količino podatkov med virtualkami, je najbolje, če so te čim

bližje. Če želimo geografsko ločene lokacije se najbolje odrežejo obale ZDA ali ZDA-Evropa. V času pisanja je ponudnik dodal še nov podatkovni center v Frankfurtu. Razlike med samimi protokoli niso zelo velike, vsekakor pa se za večje prenose izogibamo uporabi SFTP, razen če potrebujemo integriteto in avtentičnost prenesenih podatkov.

V času pisanja je izšla tudi nova različica operacijskega sistema Debian, verzija 8. Ta prinaša novejše Linux jedro in posodobljene pakete programske opreme. Če najemamo novo virtualko v tem trenutku skoraj ni razloga, da ne bi izbrali novejše verzije.

Poglavje 4

Zmogljivst Beowulf gruče Raspberry Pi računalnikov

Gregor Vitek, Žan Palčič, Maj Smerkol

4.1 Uvod

Na spletu lahko najdemo že opravljene znane benchmarke [57], ki ocenjujejo določene lastnosti oblačnih sistemov (S2). Tu lahko najdemo podatke o procesni moči sistema, ki ga dobimo v uporabo, količini pomnilnika na takšnem sistemu in zmogljivosti opravljanja nekaterih znanih testov, kot so na primer urejanje velike količine podatkov. Te meritve, ki jih lahko preberemo med že opravljenimi testi, pa ne kažejo na zmogljivost neke določene storitve, kot jo doživlja uporabnik, ampak samo povejo, kako zmogljiva je strojna oprema, ki jo dobimo na voljo. Končnega uporabnika storitve ponavadi zanima predvsem hitrost odzivanja, ki je odvisna od več parametrov. Med slednje spadajo hitrost in latenca povezave od naprave uporabnika (angl. *end point*) do fizične lokacije oblačne storitve ali strežnika, velikost poslanega zahtevka, hitrost odbelave zahtevka, hitrost in latenca poslanega odgovora iz oblaka ali strežnika proti uporabniku in drugo. Pri tem lahko lahko nekatere storitve implementiramo na tak način, da uporabnik verjame, da je odzivni čas veliko manjši, kot dejansko je (na primer shranjevanje datotek na disk v oblaku). Na končno uporabnikovo izkušnjo hitrosti vpliva tudi zmogljivost naprave, ki predstavlja njegovo dostopno točko.

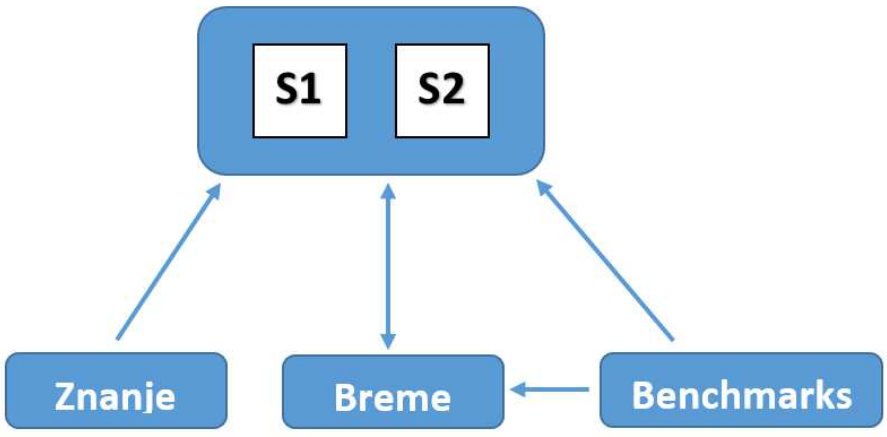
Kot omenjeno zgoraj se na spletu najdejo seznami spletnih sistemov, ki jih lahko uporabniki med seboj primerjajo. Primerjajo lahko rezultate za različne parametre, kot so hitrost procesorja pri računanju s plavajočo vejico ali celimi števili, hitrost prenosa pri branju podatkov oz. pisanju na pomnilnih enotah, hitrost prenosa podatkov lokalno znotraj oblačne storitve in drugo. Pri testiranju

teh parametrov se lahko uporabi različna orodja, kot so SPEC CPU 2006, Test Harness, TeraSort, Geekbench in še mnogo drugih. Pri sami izbiri programov se moramo osredotočiti tudi na bremena, ki jih lahko s posameznim programom definiramo in tako testiramo željene parametre. Ključen kriterij poleg kakovosti storitev in definiranja bremen je tudi cena. Zaželjeni so prosto dostopni oz. zastonjski progami.

Pri testiranju spletnih strežnikov (S1) uporabnika navadno zanima število zahtev, ki jih strežnik obdela, latenca oz. čas odziva strežnika za novo povezavo ali zahtevek, in količina prenesenih podatkov v sekundi, glede na različne parametre (velikost, shranjevanje v predpomnilnik, različna pasovna širina). Za izvajanje stresnih testov se na spletu nahaja veliko orodij, ki lahko pridejo v pomoč (Apachech, Apache JMeter, Curl-loader, OpenSTA).

4.1.1 Znanje

Sistem, ki ga bomo testirali, smo izbrali na podlagi svojega znanja in zanimanja. Člani te skupine imamo predznanje iz arhitekture in organizacije računalniških sistemov, kot so procesne enote, pomnilniška hierarhija in vhodno/izhodne naprave. Poznamo tudi paralelno programiranje na različnih platformah v jeziku C, kar bi nam lahko pomagalo pri optimizaciji določenih storitev. Imamo predznanje iz strežniških arhitektur in osnov spletne komunikacije. Imamo tudi omejene izkušnje uporabe PaaS (angl. *platform as a service*) za postavitev spletnih strani in postavitve podakovnih baz na teh strežnikih. Nimamo pa izkušenj s testiranjem in merjenjem zmogljivosti katerega koli od naštetih modelov.



Slika 4.1: Odvisnost izbire S1 in S2 glede na podane atribute.

4.1.2 Izbira ciljnih sistemov

Za ciljni sistem bi lahko izbrali kakšnega od večjih ponudnikov oblačnih storitev kot so Google Cloud, Amazon web service, Microsoft Azure, lahko tudi kakšnega manjšega, na primer Rackspace. Večina teh ponudnikov ima omejena zastonjska testna obdobja, med katerimi bi lahko izvedli meritve. Ker imamo dostop do računalnika Raspberry Pi [1] in predvsem zastonjskih verzij oblačnih storitev, nas zanima primerjava med temi platformami. Raspberry Pi je računalnik, ki stane okrog 30 evrov in ne porabi praktično nič elektrike, v zameno pa ponuja primerno majhno zmogljivost. Ne plačljive oblačne storitve so močno časovno omejene ali pa ponujajo prav tako zelo majhne zmogljivosti. Zaradi naših predznanj lahko storitev za strežnik (glej sliko 4.1), katerega ustroj in delovanje poznamo, bolje optimiziramo, kot oblačno storitev, kar je prav tako vredno preveriti.

Ker pa je implementacija optimizirane storitve in postavitev lastne gruče relativno zahtevno opravilo, mi pa imamo na voljo omejen čas, smo se odločili za testiranje zmogljivosti gruče računalnikov Raspberry Pi. Le-ti so se v zadnjem času zaradi nizke cene na mnogih področjih računalništva začeli uporabljati. Nas zanima, ali so računalniki takšnega tipa primerni tudi za uporabo na področjih, ki zahtevajo računsko moč. Na podlagi izmerjenih rezultatov lahko primerjamo zmogljivost z drugačnimi sistemi, na katerih so meritve že opravljene. Prav tako lahko nadaljujemo raziskovanje na tem področju s poskusom optimizacije enake storitve na drugačnem sistemu v prihodnosti.

4.1.3 Breme in cilj primerjave

Za storitev, ki jo bo naša gruča ponujala, smo izbrali manipulacijo nad frekvenčnim spektrom avdio datotek. To pomeni, da smo implementirali porazdeljen algoritem FFT, ki pretvori datoteko iz vzorčnega prostora v frekvečni spekter, nato nad tem opravimo preprosto operacijo, na primer zamik v višje frekvence, nato pa z inverznim FFT spet naredimo datoteko v vzorčnem prostoru.

Bremena so tako različno velike zvočne datoteke. Ker naš spletni strežnik, ki to storitev ponuja, sprejme eno ali več datotek, testiramo tudi izvajanje večih instanc algoritma hkrati.

Cilj primerjave je ugotoviti, ali je računalnik Raspberry Pi primeren za postavitev gruče, ki bo izvajala računsko zahtevne storitve. Da bi to ugotovili, moramo primerjati čas izvajanja na procesorju, čas latence oddaljenega sistema prek medmrežja in čas pošiljanja podatkov po gruči.

4.2 Sistem

Naš sistem je gruča računalnikov, narejena po vzoru Beowulf gruč (glej sliko 4.2). Namenjena je zaganjanju porazdeljenih programov. V mrežo smo povezali štiri računalnike Raspberry Pi, model 1B, tri modele 1B+. Povezani so v gručo s

[1] http://www.raspberrypi.org/

Slika 4.2: Fotografija gruče. Na levi je 100 megabitno stikalo, na desni pa računalniki Raspberry Pi na pladnju.

pomočjo knjižnice MPI[2]. Do njega imamo SSH dostop tudi z zunanjega omrežja. Storitev je dostopna prek strežnika, ki smo ga implementirali sami. Programska koda strežnika in implementacije algoritma FFT je dostopna na githubu na naslovu `https://github.com/Halofit/ClusterFFT`.

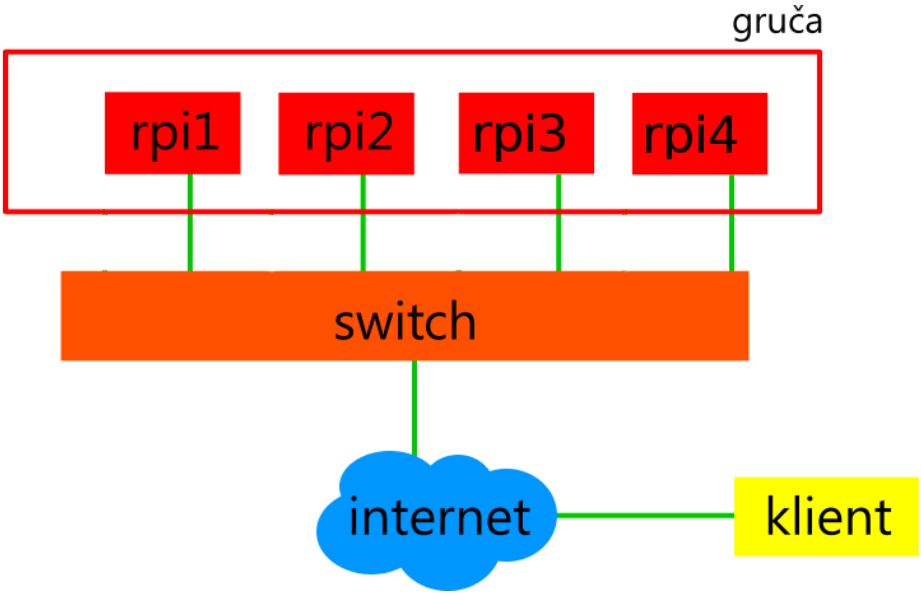
Do tega sistema dostopamo s klienti, ki so izven lokalnega omrežja (glej sliko 4.3). Gručo uporabljamo kot računski strežnik, ki po zahtevi klienta izvaja porazdeljeno hitro Fourierjevo transformacijo. Klient prek omrežja pošlje podatke v obliki glasbene datoteke, gruča pa podatke sprejme (to stori strežnik, ki teče na master računalniku gruče) in izvajanje algoritma porazdeli prek vseh računalnikov v gruči, vključno s sabo.

Gruča se nahaja na Velikem Ločniku, na omrežju Telekoma. Dostopna je prek širokopasovne povezave po optičnem kablu (angl. *FTTH - fiber to the home*) s hitrostjo 20Mb/s v obe smeri. Vse povezave po hiši so po bakreni parici kategorije 6, omrežna oprema pa je prepustna najmanj 100Mb/s.

V začetku smo testirali dostop do gruče od zunaj in delovanje sistema MPI na računalniku rpi1. To smo storili tako, da smo na njem poganjali z MPI implementiran algoritem porazdeljeni Quicksort (porazdeljeno hitro urejanje[3]).

[2]Message Passing Interface, knjižnica za medprocesno komunikacijo prek omrežja ali na lokalnem računalnikom

[3]http://en.wikipedia.org/wiki/Quicksort#Parallelization



Slika 4.3: Shema sistema. Rdeči kvadrati so računalniki Raspberry Pi.

Pri povezovanju Raspberry Pi-jev v gručo smo uporabili Message Passing Interface(MPI), ki je standard za izmenjevanje sporočil med računalniki oz. procesih v več-računalniških sistemih, gručah in delovnih postajah. Omogoča komunikacijo točka-točka, skupinske komunikacije, spremljanje delovanja in tudi spreminjanje topologije za posamezen program. MPI je standard, ki omogoča velik nabor funkcij, vendar za uporabo standarda ni potrebno poznati vseh. Pri sami komunikaciji računalniki med seboj uporabljajo implementacijo MPICH2 [58], ki je zelo razširjena programska knjižnica, ki je enostavna za vzpostavitev in uporabo. Na vsakem računalniku je implementacija MPICH2 in datoteka (machinefile) [59], v kateri so zapisani IP naslovi sodelujočih (sosednjih) lokalnih računalnikov, na katere se delo porazdeli.

Implementirali smo porazdeljeni algoritem FFT, ki se izvaja na gruči.

4.2.1 Implementacija storitve

V gručo smo dodali več računalnikov, da je prava gruča, ne le en računalnik, na katerem teče MPI. Raspberry Pi smo pripravili na delovanje v gruči tako, da smo klonirali sliko datotečnega sistema. Potrebni so bili popravki v konfiguraciji omrežja in sprememba hostname-a vsakega računalnika. Po tem je gruča delovala brez težav.

Implementirali smo storitev, ki zvočno datoteko pretvori v binarno s pomočjo knjižnice `libsndfile` [60]. Testiranje smo izvedli z zvočno datoteko, veliko približno 8 miljonov vzorcev (okrog 3 minute zvoka, nekompresirana velikost je okrog 15MB). Program na enem jedru teče zelo počasi. Ni se izpolnil strah, da

bi algoritem tekel prehitro, da bi se ga izplačalo poganjati na gruči. Imeli smo nekaj težav s kodiranjem datoteke nazaj v zvočno datoteko.

Pretvorbo izvajamo z rekurzivnim algoritmom FFT. Implementacija je zaradi uporabe razreda ValArray iz knjižnice STL počasna, če je prevanjanje izvedeno z Microsoft Visual Studio prevajalnikom. Program, ki se izvaja na gruči, na kateri teče Linux (distribucija Arch Linux), je preveden s prevajalnikov gcc, ki se v tem primeru obnaša veliko bolje - koda se izvaja hitreje.

Implementirali smo strežnik, ki sprejema zahtevke z eno ali več datotekami in nato odgovori. Implementirali smo tudi klienta, ki pošilja zahtevek z eno ali več datotekami in sprejme odgovor strežnika.

Program, ki izvaja manipulacije frekvenčnega spektra, upošteva lastnosti hitre Fourierjeve transformacije in pravilno obdeluje zvok. Zavržemo del frekvenčnega spektra, ki je zrcalna slika relevantnega dela. Transformacije, ki jih izvajamo na preostanku, so zato smiselne in se obnašajo kot pričakovano.

4.3 Meritve

Naš problem predstavlja veliko računsko breme gruči, ampak relativno majhno breme za prenos podatkov prek omrežja. Na enem vozlišču traja obdelovanje 5 sekund zvoka okrog 40 sekund. Algoritem je hitrostnega razreda $O(n * log(n))$.

Zaradi tipa problema in bremena nas zanimajo časi računanja (procesiranja na gruči, wall-clock time, v odvisnosti od števila uporabljenih vozlišč in velikosti bremena), prenosa podatkov na in z gruče (latenca omrežja - čas prenosa je najbrž zanemarljiv, saj so datoteke velike med 1MB in 15MB), latence prenosa podatkov med vozlišči v gruči (da vidimo, ali je paralelizacija na gruči primerna ali bi bilo bolje uporabljati paralelizacijo na več jedrih ali podobni arhitekturi) in največje število datotek, ki jih lahko obdeluje hkrati, preden se gruča ali njeni deli sesujejo.

Izvajamo več meritev:

- celoten čas izvajanja storitve od trenutka, ko klient začne pošiljati podatke, do trenutka, ko se do konca prenese odgovor strežnika.
- čas izvajanja računskega dela storitve od klica programa FFT do konca njegovega izvajanja.
- čas prejemanja in pošiljanja podatkov prek omrežja.
- čas pošiljanja podatkov med računalniki v gruči po lokalnem omrežju (klici funkcij MPI scatter in MPI gather).
- skupna hitrost obdelava podatkov v bajtih na sekundo.

Meritev časa izvajanja računanja in meritev časa prenosa podatkov med procesi smo ponovili tudi na računalniku s procesorjem Intel i5-3317U, s frekvenco ure 1,7GHz in 4GB DDR3 delovnega spomina in 3MB L3 cache-a. Pri tem smo

program izvajali lokalno, brez strežnika. Rezultati nam dajo primerjavo med večjedrnim procesorjem s hyperthreadingom in gručo enojedrnih. Bralec lahko primerja tudi zmogljivost glede na ceno, pri čemer mora vedeti, da računalnik Raspberry Pi stane okrog 30 evrov, stikalo, ki smo ga uporabili, stane 40 evrov do 100 evrov, računalnik z Intelovim procesorjem, ki smo ga uporabili, pa okrog 800 evrov.

Meritve smo izvedli na 1, 2 in 4 računalnikih v gruči. Izvedli smo teste z eneno vhodno zvočno datoteko, dolgo 1, 5, 20, 60 in 180 sekund ter z 1, 5 in 10 vhodnimi datotekami velikosti 5 sekund. Posebej smo testirali še več datotek po 180 sekund, s čimer smo uspeli sesuti gručo.

Zaradi zelo dolgega časa izvajanja storitve za datoteke, dolge 180s, smo to meritev izvajali le na vseh 4 računalnikih v gruči in na procesorju Intel i5.

Vse teste smo ponovili trikrat.

4.3.1 Izmerjeni časi: 1 datoteka, različne velikosti datoteke

Meritev celotnega časa izvajanja storitve od trenutka, ko klient začne pošiljati podatke, do trenutka, ko se do konca prenese odgovor strežnika.

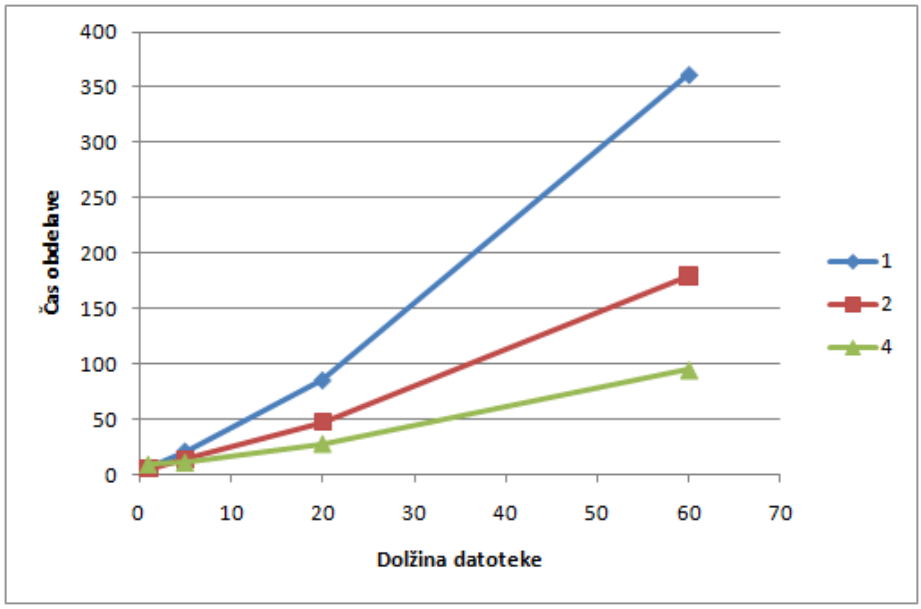
RPIs	n[s] = 1	n[s] = 5	n[s] = 20	n[s] = 60	n[s] = 180
1	6,36	21,85	86,11	359,54	-
	6,23	20,70	85,98	361,90	-
	6,48	21,73	84,97	363,97	-
2	5,27	14,50	44,69	179,31	-
	5,23	14,08	48,99	179,84	-
	5,20	13,08	48,35	178,99	-
4	8,26	10,67	27,85	94,49	193,99
	9,31	10,66	28,50	93,18	203,01
	9,34	11,06	27,15	97,29	201,75

Tabela 4.1: Tabela izmerjenih vrednosti časa izvajanja celotnega zahtevka.

Na grafu 4.4 vidimo, da se, ko podvojimo število vozlišč v gruči, čas izvajanja skoraj razpolovi. Iz grafa 4.5 lahko vidimo, da je razlika med linearno pohitritvijo in doseženo pohitritvijo predvsem posledica daljšega prenosa podatkov prek omrežja, ki seveda ni odvisen od velikosti gruče.

4.3.2 Izmerjeni časi prenosa podatkov in režije (angl. *overhead*) v sekundah: 1 datoteka, različne velikosti datoteke

Rezultati meritev so prikazani v tabeli 4.2.



Slika 4.4: Graf časa celotnega izvajanja zahtevka.

RPIs	n[s] = 1	n[s] = 5	n[s] = 20	n[s] = 60	n[s] = 180
1	0,17	0,29	1,57	4,40	-
	0,12	0,36	1,24	3,83	-
	0,12	0,35	1,09	3,82	-
2	0,13	0,34	1,28	3,79	-
	0,12	0,34	1,47	4,16	-
	0,15	0,68	1,12	3,79	-
4	0,66	1,59	5,25	4,35	17,96
	0,46	4,65	7,90	8,32	20,09
	0,78	2,06	8,32	7,39	19,61

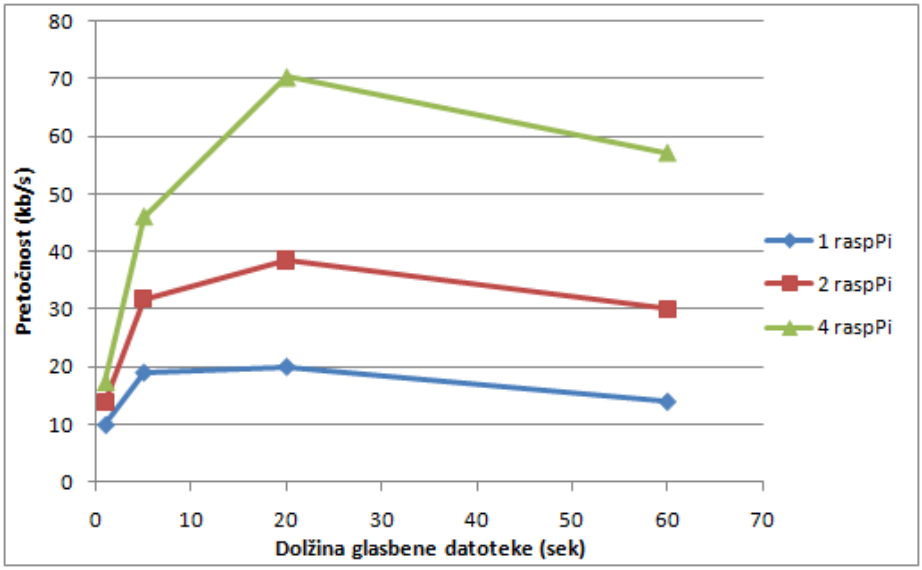
Tabela 4.2: Tabela izmerjenih vrednosti časa prenosa podatkov na prek omrežja na gručo.

4.3.3 Izmerjena hitrost obdelave podatkov v kB/s: 1 datoteka, različne velikosti datoteke

Na grafu 4.5 vidimo primerjave hitrosti obdelave podatkov za celotno storitev (velikost datoteke v kilobajtih deljena s časom, kot ga vidi klient). Zanimivo je, da je očitno okrog datotek velikosti 20 sekund optimalna točka, pri kateri se časi obdelave podatkov, prenosa podatkov in latenca omrežja pridejo v za uporabnika najboljše razmerje. Lahko razberemo tudi, da je hitrost obdelave odvisna skoraj linearno od števila računalnikov v sistemu. To pomeni, da še nismo dosegli maksimalne pohitritve po Amdahlovem zakonu in bi s podvojitvijo

RPIs	n[s] = 1	n[s] = 5	n[s] = 20	n[s] = 60
1	9636	19781	20137	14434
	10871	18099	20684	14948
	10872	18872	20245	14863
2	14429	33045	38939	30263
	12404	30738	37932	29522
	14356	32215	39086	29255
4	17298	47105	70233	57326
	17392	45511	70452	56347
	17230	47181	70283	58203

Tabela 4.3: Tabela izmerjenih vrednosti hitrosti obdelave podatkov. n[s] je dolžina zvočne datoteke v sekundah, RPIs pomeni število vozlišč v gruči.



Slika 4.5: Graf hitrosti obdelave podatkov.

števila računalnikov lahko še vsaj enkrat podvojili zmogljivost gruče.

4.3.4 Izmerjen čas računananja FFT v sekundah: 1 datoteka, različne velikosti datoteke

V tabeli lahko vidimo, da je pohitritev pri dodajanju vozlišč v gručo superlinearna. To se pri porazdeljevanju programa skoraj nikoli ne zgodi. V našem primeru je do tega prišlo, ker nalogo razdelimo na manjše podprobleme na drugačen način, kot je to standardno pri FFT. Ponavadi se pri FFT zvočno datoteko razdeli na lihe in sode vzorce, mi pa smo celotno datoteko razdelili na več manjših. Ob tem sicer izgubimo na kvaliteti zvoka, vendar to zaradi velikosti datotek

RPIs	n[s] = 1	n[s] = 5	n[s] = 20	n[s] = 60
1	4,71	18,16	82,15	357,95
	5,15	18,70	79,56	345,43
	4,26	19,90	82,18	348,29
2	2,16	9,23	39,93	166,33
	2,15	9,64	41,16	171,19
	2,12	9,24	39,71	173,15
4	1,13	4,65	19,45	84,26
	1,11	4,54	19,98	85,31
	1,07	4,67	19,47	82,40

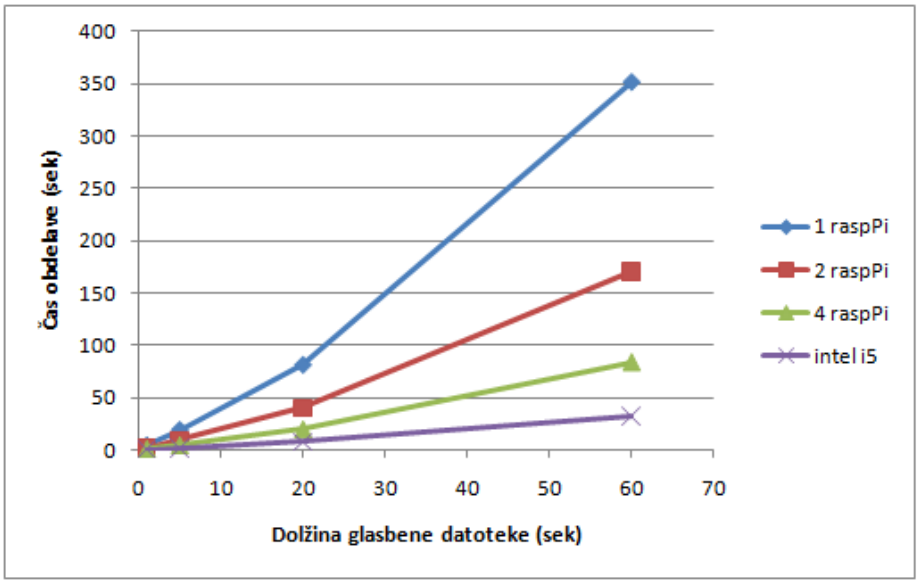
Tabela 4.4: Tabela izmerjenih vrednosti časa izvajanja algoritma FFT. n[s] pomeni dolžina zvočne datoteke v sekundah, RPIs pa število vozlišč v gruči.

ni problematično, saj se v praksi ponavadi stori isto, vendar na še mnogo več mnogo manjših kosov, kot smo to storili mi. Razlika v kvaliteti zvoka ni opazna za človeka. Rezultat tega je, da zahtevnost algoritma pade in pride do rahlo superliearne pohitritve. Glej poglavje zaključek 4.4.

4.3.5 Čas izvajanja algoritma FFT na računalniku s procesorjem i5, za primerjavo računske zmogljivosti

št. jeder	n[s] = 1	n[s] = 5	n[s] = 20	n[s] = 60	n[s] = 180
1	1,15	4,84	21,56	86,54	133,01
	1,07	4,76	19,74	86,91	187,35
	1,07	4,57	17,65	86,50	179,31
2	0,79	3,33	12,89	50,69	112,80
	0,63	2,63	11,51	49,31	73,87
	0,66	2,74	11,26	48,28	107,95
4	0,50	1,87	7,85	33,92	71,17
	0,41	1,76	7,38	31,56	68,11
	0,46	1,84	7,43	31,46	66,88

Na grafu 4.6 vidimo primerjavo časov izvajanja algoritma FFT. Na abscisni osi so različne dolžine datotek, na ordinatni pa čas izvajanja. Skala je povsod linearna. Različni grafi predstavljajo meritve različnih konfiguracij sistema. Najhitrejši, označen kot Intel i5, je en dvojedrni procesor s hyperthreadingom. Glede na celotno ceno sistema se gruča izkaže za boljšo rešitev. Zanimiva je tudi primerjava med gručo z različnim številom vozlišč, saj je pohitritev praktično linearno odvisna od števila vozlišč v gruči.



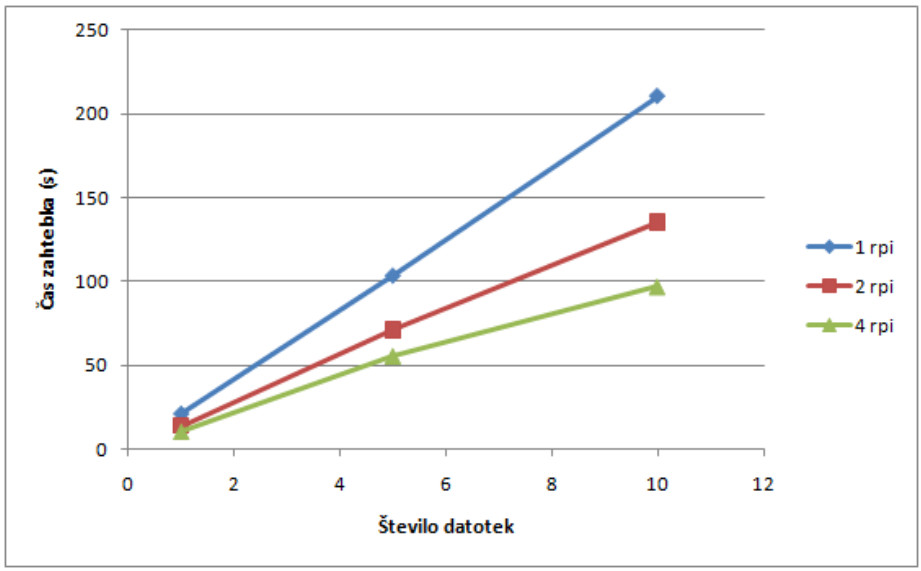
Slika 4.6: Graf časov izvajanja obdelave podatkov.

RPIs	N[f] = 1	N[f] = 5	N[f] = 10
1	21,85	102,38	208,68
	20,70	104,35	203,89
	21,73	103,75	218,35
2	14,50	70,95	126,58
	14,08	70,32	150,61
	13,08	71,55	129,25
4	10,67	55,44	98,99
	10,66	55,93	95,94
	11,06	55,80	95,94

Tabela 4.5: Tabela izmerjenih vrednosti celotnega izvajanja zahtevka. N[f] pomeni število zvočnih datoteke dolžine 5 sekund, RPIs pa število vozlišč v gruči.

4.3.6 Izmerjeni časi izvajanja celotnega zahtevka: datoteka dolga 5s, različno število datotek v zahtevku

Iz grafa 4.7 vidimo, da je čas izvajanja približno linearno odvisen od števila datotek. To je razumljivo, saj gre za več ponovitev enakega zahtevka. Če bi število datotek zelo povečali, bi najbrž prišlo do velike upočasnitve, ko bi začelo zmanjkovati spomina na gruči, vendar se ob tem skoraj istočasno tudi gruča sesuje, ker je zaradi majhnega zunanjega pomnilnika (SSD kartica) zelo malo swap prostora, kar pomeni, da zmanjka celotnega pomnilnika skoraj istočasno kot delovnega pomnilnika. Glej poglavje 4.4.

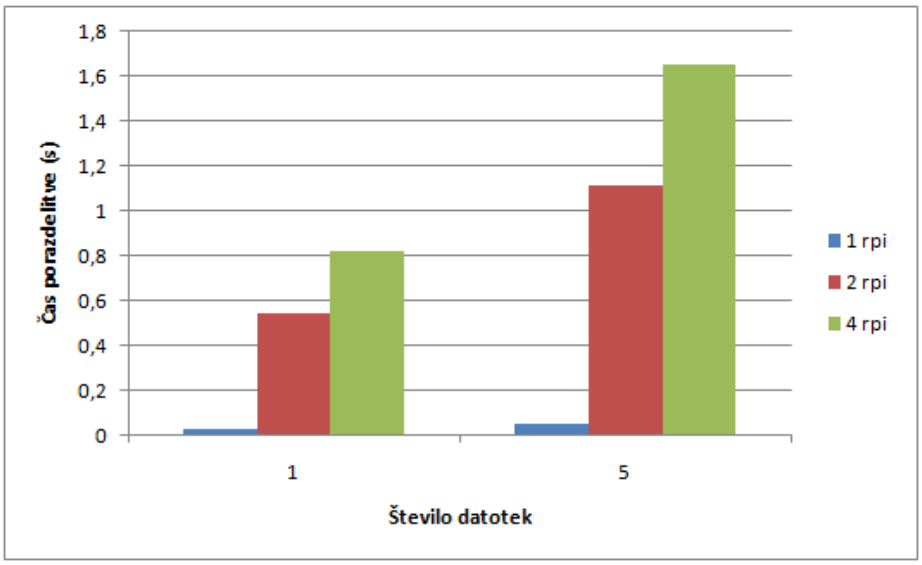


Slika 4.7: Graf časov izvajanja celotnega zahtevka za več datotek.

4.3.7 Izmerjeni časi porazdelitve podatkov po gruči: datoteka dolga 5s, različno število datotek v zahtevku

RPIs	N[f] = 5	N[f] = 10
1	0,024	0,047
	0,024	0,048
	0,025	0,048
2	0,542	1,10
	0,548	1,15
	0,549	1,09
4	0,841	1,64
	0,816	1,63
	0,810	1,69

Iz grafa 4.8 vidimo, da so časi porazdelitve podatkov po gruči (MPI scatter in gather funkciji) zelo majhni in predstavljajo reda 1% celotnega časa izvajanja. Zanimivo je, da je razlika v času prenosa majhna pri prehodu iz dveh vozlišč na 4, zelo velik pa je pri prehodu iz enega na dve. To je razumljivo, saj v primeru enega vozlišča podatkov ni treba nikamor razpošiljati, pri večih pa mora podatke poslati prek lokalnega omrežja na druge računalnike.



Slika 4.8: Graf časov pošiljanja porazdelitve podatkov po gruči za več datotek.

RPIs	N[f] = 5	N[f] = 10
1	93,73	192,83
	93,73	186,84
	92,90	186,84
2	45,90	91,97
	46,30	96,69
	47,64	92,45
4	23,47	47,19
	23,23	46,48
	23,42	48,38

Tabela 4.6: Graf časov izvajanja FFT. F[f] pomeni število 5 sekundnih datotek, RPIs pa število vozlišč v gruči.

4.3.8 Izmerjen čas izvajanja alogritma FFT: datoteka dolga 5s, različno število datotek v zahtevku

Podatki kažejo na linearno pohitritev glede na število vozlišč in linearno povečanje časa glede na število datotek.

4.3.9 Izmerjena hitrost obdelave podatkov v kB/s: datoteka dolga 5s, različno število datotek v zahtevku

Maksimalna hitrost obdelave podatkov je veliko manjša pri več majhnih datotekah kot pri eni veliki. Zanimiva je primerjava s tabelo 4.3.

RPIs	N[f] = 5	N[f] = 10
1	18861	18443
	18789	18996
	19118	18133
2	32230	31850
	31792	30810
	31148	31847
4	47518	47360
	47514	47364
	47544	43176

4.3.10 Izmerjeni časi izvajanja celotne storitve: več klientov, ena datoteka, dolga 5 sekund

Meritev celotnega časa izvajanja storitve od trenutka, ko klient začne pošiljati podatke, do trenutka, ko se do konca prenese odgovor strežnika.

RPIs	2 klienta	4 klienti	8 klientov
4	17,04	30,76	56,21
	16,20	31,40	45,14
	16,83	30,89	nuspešno

Tabela 4.7: Tabela meritev časa izvajanja celotne storitve pri več klientih.

Namen teh meritev je bila predvsem ugotoviti, kdaj se bo strežnik sesul. Pri 8 ali več klientih začne zapirati nekatere povezave, nato pa se sesuje. Klienti se sesujejo, kadar je povezava prekinjena. Meritve so povprečni rezultati vseh uspešnih klientov. Pri tretji meritvi z 8 klienti se je strežnik sesul, zato meritve ni.

4.4 Komentarji in zaključek

4.4.1 Algoritem FFT in naša implementacija

Ker za dovolj velik n velja $n * log(n) > n * log(n/m) * m$, je hitrost reševanja problema večja, če rešujemo več manjših problemov, kot enega velikega. To je razlog, da ne porazdeljujemo po gruči po drevesu razcepa podatkov, ki ga izvaja FFT, ampak razdelimo vse podatke na m delov in nato obdelujemo vsakega posebej, pri čemer je m število računalnikov v gruči. To je slaba rešitev za zelo velik m ali zelo majhne zahtevke, saj bi se lahko poznala izguba kvalitete zvoka, ki jo to prinese. Razlika se na normalnih zahtevkih in našem m sicer ne pozna. Takšen pristop se uporablja tudi v praksi, predvsem kadar se izvaja transformacije v realnem času (sproti).

4.4.2 Maksimalna obremenjenost gruče pred sesutjem

Z eno datoteko nam ni uspelo sesuti gruče, je pa pri zvočnih datotekah, dolgih 3 minute, čas izvajanja narasel nad mejo uporabnega.

Poskusili smo sesuti gručo tudi z več datotekami, poslanimi z enega klienta. Izkazalo se je, da je naš program premalo požrešen, da bi na gruči zmanjkalo delovnega spomina, zato nam tudi tako ni uspelo sesuti gruče.

Uspešno smo sesuli strežnik tako, da smo nanj povezali 10 klientov hkrati. Vsak klient je poslal eno datoteko. Strežnik je datoteke sprejemal, nato pa je nekatere povezave zaprl, nekatere pa so se izvedle uspešno. Kot nadaljnje delo bi bilo zanimivo ugotoviti, zakaj se je to zgodilo in kako bi to preprečili, saj za razlog nismo prepričani. Sumimo, da je zaradi neuspešnega klica sistemskega klica malloc (rezervacija spomina) prišlo do pisanja po nedovoljenem spominu.

4.4.3 Ugotovitve

Na podlagi vseh izmerjenih podatkov smo ugotovili, da je Beowulf gruča, sestavljena iz računalnikov Raspberry Pi, ugodna rešitev glede na razmerje med ceno in zmogljivostjo. Zaradi zelo majhnega delovnega pomnilnika (512MB, model 2 verzija B+ pa 1GB) ni primerna za reševanje velikih problemov. Pri določenih problemih, na primer našem, je ta težava rešljiva s pazljivim programiranjem, pri mnogih pa ni.

Ker je pri uporabi gruče vedno prisotna režija (predvsem porazdeljevanje ali kopiranje podatkov po gruči), je en zelo zmogljiv sistem boljši od gruče majhnih računalnikov, vendar na račun visoke cene. Razlika je očitna - na enem računalniku je čas režije 20- do 40-krat manjši kot na gruči (glej graf 4.8).

Ko smo testirali z več instancami klienta hkrati, so se pojavile mnoge težave na našem strežniku. Mnogo klientov je prekinil, pri 10 ali več klientih pa se je strežnik sesul. To nam pove, da naša konfiguracija ni primerna za uporabo v realnem svetu kot oddaljena storitev. To je sicer težava naše implementacije in ne gruče ali računalnikov Raspberry Pi.

Poglavje 5

Testiranje zmogljivosti PLC-jev

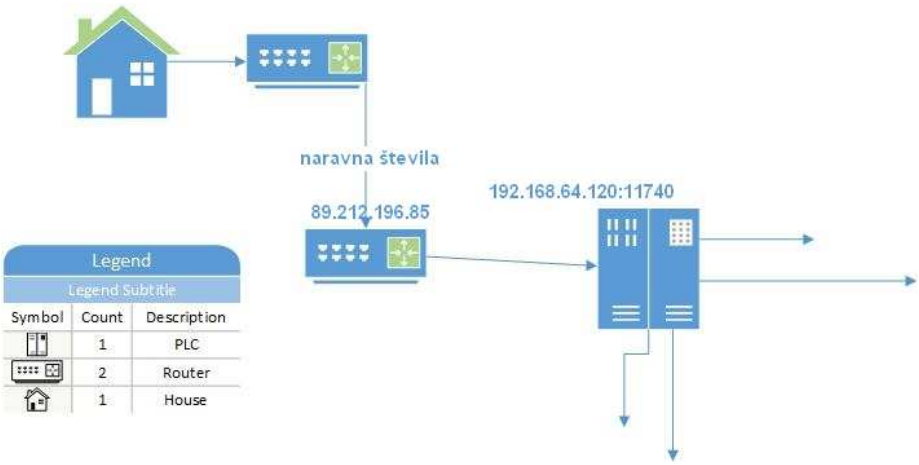
Silvester Jakša, Luka Pirnat, Marko Tavčar

listings

5.1 Ideja testiranja

Ker nas vse v skupini bolj zanima nizkonivojski del računalništva in ker smo imeli na voljo PLC-naprave, smo se odločili, da bomo naredili benchmark PLC-ja. Odločili smo se za JanzTec emVIEW 6/A500.

Testirali smo njegovo zmogljivost, pri čemer smo si pri oblikovanju testov pomagali z virom [61]. Preko protokola TCP smo testirali ethernetni vmesnik PLC-ja hkrati z internetno povezavo. S testom smo želeli simulirati realno situacijo, ko je PLC na neki lokaciji, mi pa do njega dostopamo prek omrežja in ga bodisi nastavljamo bodisi beremo kake podatke z njega ali kaj drugega. Tako smo ga tudi mi povezali v internet, potem pa mu prek njega pošiljali umetni model bremena (slika 5.1). Tega smo sestavili iz vrste vhodnih podatkov, ki jih je PLC sprejemal in prikazoval na zaslonu. Med prejemanjem je PLC opravljal tudi druge naloge. Želeli smo namreč izmeriti sposobnost opravljanja vrste nalog v realnem času med prejemanjem sporočil, ta sposobnost pa nam je potem predstavljala tako imenovani benchmark same naprave. Konkretno - merili smo Round Trip Time (RTT) - čas od trenutka, ko je paket poslan, do trenutka, ko prejmemo potrditev. Naš cilj je bil ugotoviti, ali na RTT v večji meri vpliva zmogljivost PLC-ja ali omrežje - njegova kakovost in razdalja.



Slika 5.1: Shema omrežja, v katerem smo izvajali testiranje.

5.2 Namen PLC-naprave

PLC - Programmable Logic Controller - je digitalni računalnik, ki se ga uporablja za avtomatizacijo tipično industrijskih elektronsko-mehanskih procesov, npr. za nadzor strojev za sestavljanje izdelkov, vlivanje umetnih mas ipd.
PLC-ji imajo veliko število analognih in digitalnih vhodov in izhodov. Delujejo lahko pod različnimi pogoji - visoke/nizke temperature, električni šum ipd. in so odporni na vibracije in udarce. Programi za kontrolo strojev so navadno shranjeni v pomnilniku s pomožnim napajanjem ali pa v statičnem pomnilniku. PLC je dober primer realnočasovnega sistema, saj morajo biti rezultati izračunani kot odziv na vhodne pogoje v omejenem času, sicer pride do neželenih akcij strojev, ki jih kontrolira. Zaradi te lastnosti je PLC tudi idealen sistem za naše testiranje [62].

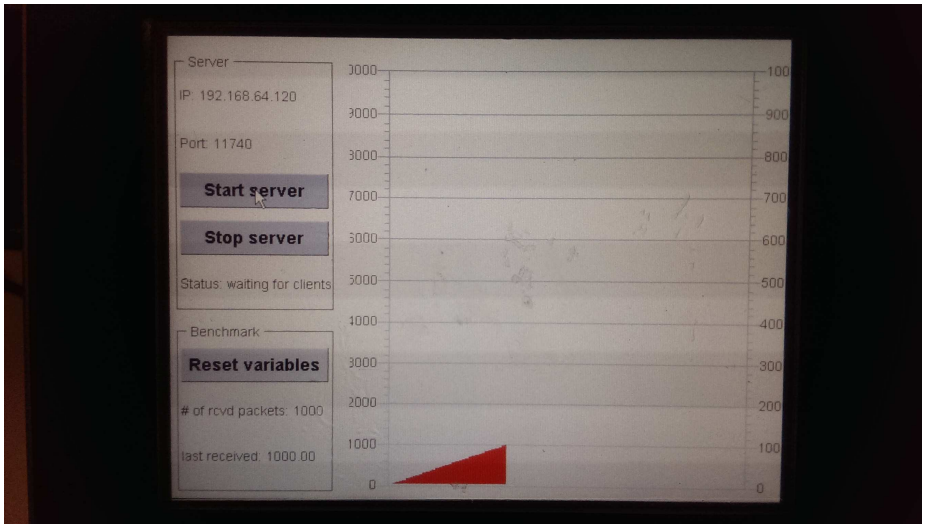
V našem primeru je PLC predstavljal enostaven TCP-strežnik, ki sprejema podatke in jih potem lahko obdela in pošlje nazaj ali pa izvede kako akcijo, glede na to, kakšne podatke je prejel. To je gotovo uporaben primer uporabe PLC-ja, saj ga lahko tako krmilimo oz. mu pošiljamo poljubne podatke iz kjerkoli na svetu.

5.3 Strojna oprema

Pri testiranju smo uporabljali naslednjo strojno opremo:

- PLC JanzTec emVIEW 6T/A500/R3 (sliki 5.2 in 5.3):
  - ta PLC je v resnici Human Machine Interface (HMI), ki sestoji iz PLC-ja emPC in rezistivnega na dotik občutljivega LCD-zaslona z

ločljivostjo 600×480 pikslov
- cena: okrog 1300 eur
- dimenzije v mm: $194 \times 159 \times 72$ (š $\times$ v $\times$ g)
- temperature delovanja: 0 - 50° C (vklopljen), -20 - 75° C (izklopljen)
- napajanje: 14 - 32 V enosmerne napetosti
- hlajenje: brez ventilatorja
- CPE: 600 MHz Freescale i.MX515
- dinamični pomnilnik - RAM: 256 MB DDR2
- statični pomnilnik: compact flash card
- operacijski sistem Windows CE6
- vmesniki: VGA, 2-krat USB 2.0, 2-krat 10/100 Mbps Ethernet, 3-krat RS232, CAN
- natančneje v viru [63]



Slika 5.2: Janztec emVIEW 6T/A500/R3 med izvajanjem testa.

- Usmerjevalnik/modem Iskratel Innbox V50-U, priklopljen na bakreno parico, hitrost povezave 15/1 Mbps,
- Usmerjevalnik Linksys WRT160NL, priklopljen na optično omrežje, hitrost povezave 100/10 Mbps,
- Prenosni računalnik ASUS K75VJ,
- Prenosni računalnik ASUS X75,
- Prenosni računalnik HP ProBook 6560b.

Slika 5.3: Janztec emVIEW 6T/A500/R3, zadnji del s strani.

5.4 Programska oprema

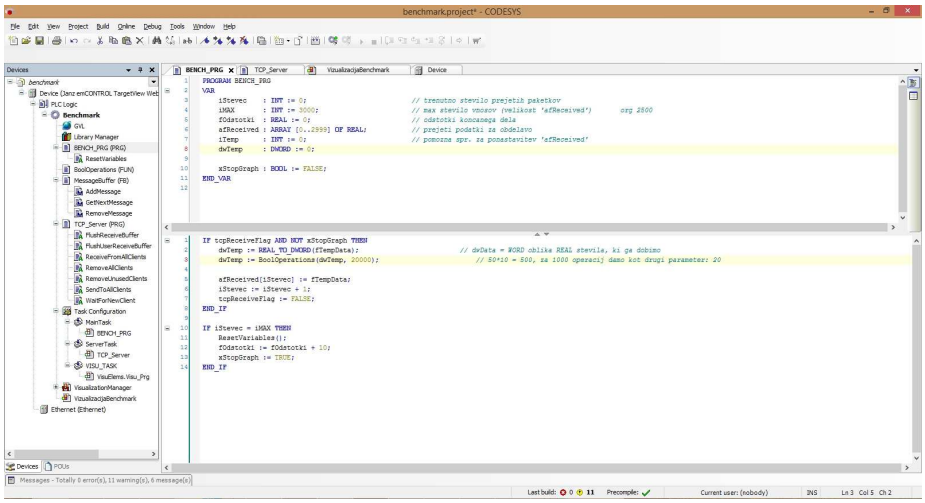
Pri testiranju smo uporabljali naslednje programe:

- CodeSys 3.5.1 (slika 5.4)
- Programski jezik ST [64] - temelji na jeziku Pascal
- Program na PLC-ju - predstavlja strežnik. Temelji na kodi PLC-Chat iz CodesysStore [65]
- Testni program 'ReceiverSender.cpp' - testni program na strani klienta

5.5 Testiranje

5.5.1 Opis testov

Testi so bili sestavljeni iz različnih bremen. Breme se je nanašalo predvsem na količino logičnih bitnih operacij oz. boolovih operacij, po viru [66] se namreč PLC-je testira z bitnimi logičnimi operacijami. PLC jih je izvajal zatem, ko je prejel sporočila, in preden ga je odposlal nazaj. Teste smo izvajali preko



Slika 5.4: Razvojno okolje CodeSys.

ethernetne povezave na različnih lokacijah, natančneje iz Ljubljane, Velikih Lašč, iz Krakova na Poljskem in lokalno v Ivančni Gorici. Razlog za to, da smo iz Krakova pošiljali le 1000 števil je, da je to počel kolega, ki pa je imel omejen čas, zato smo morali teste skrajšati. Pri pošiljanju iz Velikih Lašč pa je šlo le za primerjavo s 'pošiljanjem v dvoje' in nismo potrebovali tako veliko testov.

- *Test A*: 1000 števil iz Krakova (Poljska) brez obremenitve,
- *Test B*: 1000 števil iz Krakova (Poljska) z obremenitvijo 1 milijona boolovih operacij,
- *Test C*: 3000 števil iz Ljubljane brez obremenitve,
- *Test D*: 3000 števil iz Ljubljane z obremenitvijo 1 milijona boolovih operacij,
- *Test E*: 3000 števil lokalno brez obremenitve,
- *Test F*: 3000 števil lokalno z obremenitvijo 1 milijona boolovih operacij,
- *Test G*: 3000 števil lokalno v Ljubljani z obremenitvijo 2,5 milijonov boolovih operacij,
- *Test H*: 3000 števil lokalno v Ljubljani z obremenitvijo 5 milijonov boolovih operacij,
- *Test I*: 3000 števil lokalno v Ljubljani z obremenitvijo 10 milijonov boolovih operacij,

- *Test J*: 3000 števil lokalno v Ljubljani z obremenitvijo 20 milijonov boolovih operacij,
- *Test K*: 1000 števil iz Bežigrada v Šiško, hkrati pa še 1000 iz Velikih Lašč v Šiško z obremenitvijo 1 milijon boolovih operacij,
- *Test L*: 1000 števil iz Velikih Lašč z obremenitvijo 1 milijona boolovih operacij.

5.5.2 Potek testiranja

Iz različnih oddaljenih lokacij (Ljubljana, Krakov in Velike Lašče) smo se povezali direktno na PLC, ki je bil v času testov lociran v Ivančni Gorici. Vsakič je bil klient v omrežje povezan prek brezžične povezave, PLC pa je bil povezan z UTP-kablom (Wi-fi vmesnika sploh nima). Na lokaciji je bil PLC priključen direktno na modem/usmerjevalnik Innbox in je imel statično določen lokalni IP-naslov 192.168.64.120, poslušal pa je na portu 11740 (slika 5.1). Da so podatki pritekali do PLC-ja, smo ustrezno nastavili port forwarding. Drugi dan testiranja pa se je PLC nahajal v Ljubljani v Šiški z istimi omrežnimi nastavitvami (testi G, H, I, J, K in L).

Na PLC smo naložili program (slika 5.2), ki konstantno posluša, ali je dobil kakšen TCP-paket, ki ga po potrebi (odvisno od testa) obdela ali pa samo izpiše na graf, ki se prikazuje na zaslonu PLC-ja, in potem odpošlje vsem povezanim odjemalcem. Sicer smo nameravali strežnik implementirati tako, da bi posameznemu pošiljatelju odgovoril le z njemu namenjenim paketom, a nam je za to zmanjkalo časa.
Če želimo podatek obdelati in ne samo sprejeti ter poslati nazaj, kličemo funkcijo, ki opravi N boolovih operacij (programska koda 5.1). Obdelava vhodnih podatkov nam predstavlja obremenitev PLC-ja, saj smo ga skušali še dodatno zaposliti in tako povečati RTT.

```
1  FOR iCounter := 0 TO iLoopMultiplier DO
2    // 10 bool operacij
3    wLowWord := fbData.wLowWord AND fbData.wHighWord;
4    wHighWord := fbData.wLowWord XOR fbData.wHighWord;
5    xBit0 := NOT fbData.xBit0;
6    xBit2 := NOT fbData.xBit1;
7    xBit4 := NOT fbData.xBit2;
8    xBit6 := NOT fbData.xBit3;
9    xBit8 := NOT fbData.xBit4;
10   xBit10 := NOT fbData.xBit5;
11   xBit12 := NOT fbData.xBit6;
12   xBit14 := NOT fbData.xBit7;
13
14   // 10 bool operacij, skupno 20
15   xBit16 := NOT fbData.xBit16;
16   xBit18 := NOT fbData.xBit14;
17   xBit20 := NOT fbData.xBit13;
18   xBit22 := NOT fbData.xBit12;
```

```
19    xBit24 := NOT fbData.xBit11;
20    xBit26 := NOT fbData.xBit10;
21    xBit28 := NOT fbData.xBit9;
22    xBit30 := NOT fbData.xBit8;
23    byLLByte := (fbData.byHHByte OR fbData.byLHByte) XOR (fbData.
        byHHByte);
24
25    // 10 bool operacij, skupno 30
26    byHHByte := (fbData.byLLByte XOR fbData.byLHByte) AND (fbData.
        byHLByte OR NOT fbData.byHLByte);
27    xBit1 := fbData.xBit0 AND fbData.xBit10;
28    xBit5 := fbData.xBit2 OR fbData.xBit12;
29    xBit9 := fbData.xBit4 AND fbData.xBit14;
30    xBit11 := fbData.xBit0 OR fbData.xBit10;
31    xBit15 := fbData.xBit2 AND fbData.xBit12;
32    xBit19 := fbData.xBit4 OR fbData.xBit14;
33
34    // 10 bool operacij, skupno 40
35    xBit3 := fbData.xBit20 XOR fbData.xBit21;
36    xBit7 := fbData.xBit30 XOR fbData.xBit31;
37    xBit13 := fbData.xBit20 AND fbData.xBit21;
38    xBit17 := fbData.xBit10 AND fbData.xBit11;
39    xBit21 := fbData.xBit20 XOR fbData.xBit21;
40    xBit23 := fbData.xBit30 XOR fbData.xBit31;
41    xBit25 := fbData.xBit10 AND fbData.xBit21;
42    xBit27 := fbData.xBit20 XOR fbData.xBit31;
43    xBit29 := fbData.xBit10 AND fbData.xBit21;
44    xBit31 := fbData.xBit20 XOR fbData.xBit31;
45
46    // 10 bool operacij, skupno 50
47    byLHByte := (NOT(fbData.byHLByte XOR fbData.byHHByte)) OR (fbData
        .byLLByte AND fbData.byLHByte);
48    byHLByte := (fbData.byLHByte XOR fbData.byHLByte) OR (NOT(fbData.
        byLLByte AND fbData.byHHByte));
49    wHighWord := fbData.wLowWord XOR fbData.wHighWord;
50    wLowWord := fbData.wLowWord OR fbData.wHighWord;
51  END_FOR
```

Programska koda 5.1: For zanka, v kateri se izvede 50 naključnih boolovih operacij. Zanka je del funkcije *'BoolOperations(dwDataIn, iLoopMultiplier)'*, v kateri iLoopMultiplier predstavlja, kolikokrat naj se zanka izvede in s tem kolikokrat 50 boolovih operacij želimo izvesti.

Število N je bilo v naših testih sprva 1 milijon, saj je to že precej visoko število logičnih operacij. Potem pa nas je zanimalo, koliko logičnih bitnih operacij lahko naredimo lokalno v času, ki ga sicer paket porabi za pot do Krakova in nazaj. Zato smo test lokalno pognali še pri obremenitvi PLC-ja z 2,5 milijona, 5 milijonov, 10 milijonov in 20 milijonov logičnih bitnih operacij, tokrat v Ljubljani, pri čemer je bil PLC priklopljen na usmerjevalnik Linksys WRT160NL.

PLC-ju smo podatke pošiljali z omenjenim programom *'ReceiverSender.cpp'* (slika 5.5), ki smo ga sprogramirali v C++, da smo imeli program, ki je zadostoval našim potrebam. Program za pošiljanje deluje tako, da se poveže na PLC in nato od nas zahteva parametre:

- začetno število,
- inkrement,
- število paketov in
- časovni interval pošiljanja paketov.

Prva dva parametra sta zanimiva samo za prikaz na PLC-ju, medtem ko sta naslednja dva povezana s samim testom. Število paketov nam predstavlja koliko paketov želimo poslati na PLC v obdelavo, časovni interval pa koliko časa (v ms) naj preteče med prejetim odgovorom iz PLCja ter naslednjim poslanim paketom iz programa.
Pošiljanje poteka tako, da program v zanki pošilja PLC-ju pakete s celim številom, zatem pa v še eni zanki čaka na pravi paket - enak integer, kot je bil poslan (če imamo več klientov PLC pošlje vsem vse, kot že omenjeno, zato moramo preverjati, kateri paket je za nas). Medtem merimo čas, ki preteče med oddajo paketa in sprejetjem potrditve iz PLC-ja oz. RTT (programska koda 5.2 in slika 5.5).

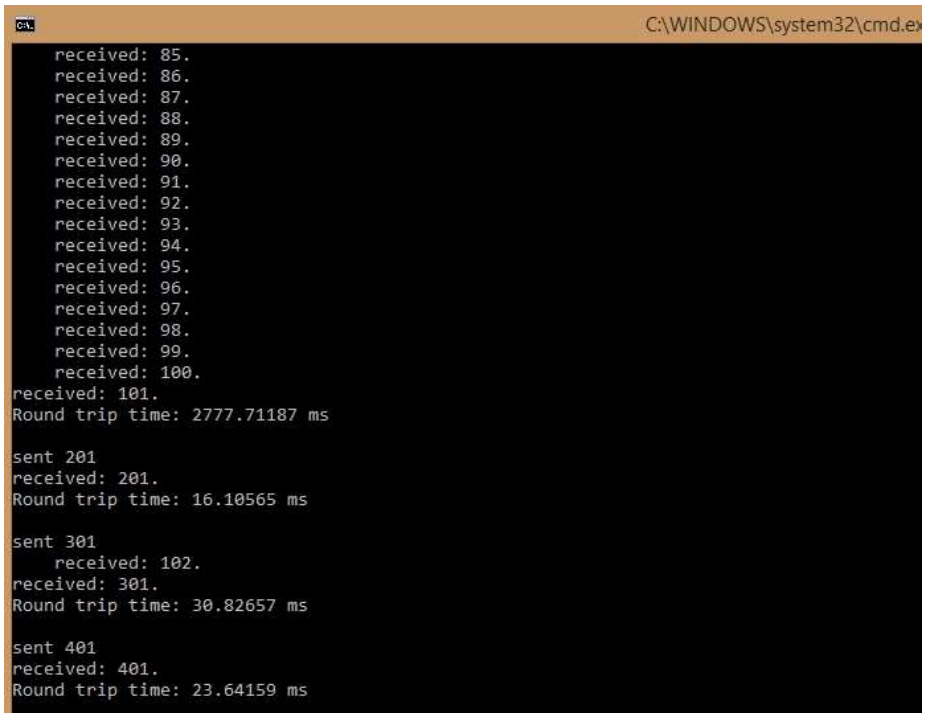
```
while (i > -1 && N > sendPackIndex) {
    length = intLength(i) + 1;
    sprintf_s(temp, "%d", i);

    // merjenje RTT-ja
    double start = getRealTime();
    double end = 0.0f;
    if ( (send(sockfd, (char *)&temp, length, 0)) == SOCKET_ERROR )
     {
      error = WSAGetLastError();
      ...
      return error;
    }
    printf("sent %s\n", temp);

    // cakaj na pravi paket
    while (i > -1) {
      if ( (recv(sockfd, buffer, sizeof(buffer), 0)) ==
    SOCKET_ERROR ) {
        error = WSAGetLastError();
        ...
        return error;
      }
      strncpy_s(temp2, buffer, length);
      temp2[length] = '\0';
      if ( strcmp(temp, temp2) == 0 ) {
        printf("received: %s.\n", buffer);
        break;
      }
      else
        printf("    received: %s.\n", buffer);
    }
    // merjenje RTT-ja
    end = getRealTime();
```

```
33      ...
34  }
```

Programska koda 5.2: While zanka, v kateri se pošiljajo in prejemajo paketi v odjemalskem programu.



Slika 5.5: Pošiljanje in prejemanje paketov v testnem odjemalskem programu. Na sliki sta na PLC povezana dva odjemalca, zato je na zgornjem delu ta odjemalec prejemal le pakete drugega, potem pa je spet začel svoje. To se dogaja izmenično.

Parametri, ki smo jih uporabljali v programu pri večini testov (če ni navedeno drugače) za pošiljanje, so sledeči:

- začetno število: 1,
- inkrement: 1,
- število paketov: 3000 (Ljubljana in lokalno), 1000 (Krakov in Velike Lašče),
- časovni interval pošiljanja paketov: 0 ms.

5.5.3 Rezultati

Lokacija	Št. testov	Št. paketov	avg min RTT	avg RTT	avg max RTT
Krakov, Poljska	10	1000	67,497	101,006	1415,129
Ljubljana	10	3000	19,642	51,051	944,547
Ivančna Gorica (lokalno)	10	3000	5,155	18,674	188,387

Tabela 5.1: Rezultati testov brez obremenitve (testi A, C in E). Vsi časi so v milisekundah.

Lokacija	Št. testov	Št. paketov	avg min RTT	avg RTT	avg max RTT
Krakov, Poljska	10	1000	75,241	149,201	1510,610
Ljubljana	10	3000	21,805	68,985	1007,702
Velike Lašče	3	1000	9,667	28,312	73,333
Ivančna Gorica (lokalno)	10	3000	6,060	25,296	306,991

Tabela 5.2: Rezultati testov z obremenitvijo 1 milijona boolovih operacij (testi B, D, L in F). Vsi časi so v milisekundah.

Št. testov	Št. paketov	Št. boolovih operacij	avg min RTT	avg RTT	avg max RTT
10	3000	1 milijon	6,060	25,296	306,991
2	3000	2,5 milijona	4,030	32,799	192,462
1	3000	5 milijonov	4,555	48,630	463,724
1	3000	10 milijonov	4,013	97,233	7129,141
1	3000	20 milijonov	4,406	174,957	11193,911

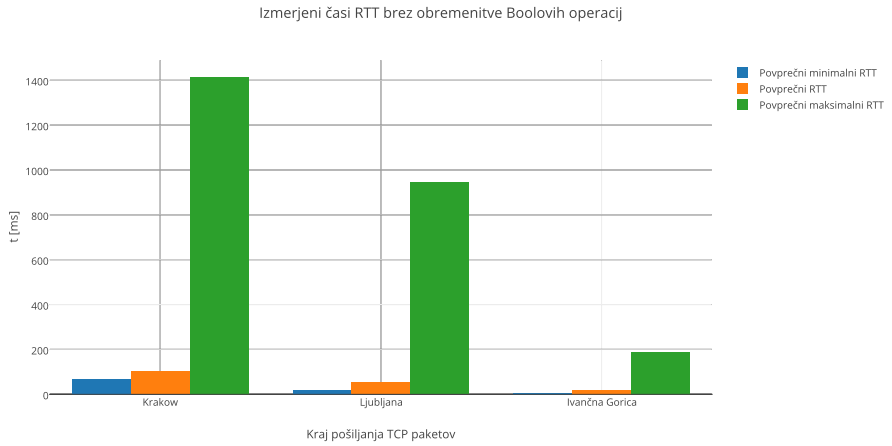
Tabela 5.3: Rezultati testov z različnim številom boolovih operacij (testi F, G, H, I in J). Vsi časi so v milisekundah.

5.5.4 Komentar rezultatov

S testi A, C in E (tabela 5.1 in slika 5.6) smo preverjali, koliko časa potrebuje v povprečju paket, da pride do PLC-ja in da pride nazaj še PLC-jev odgovor. Iz testa E bi lahko sklepali, da PLC rabi v povprečju približno 18 ms, da odgovori, saj smo bili nanj povezani lokalno. Vendar pa je bil vmes še brezžičen prenos (Wi-fi) in usmerjevalnik, tako da ta čas gotovo zajema še kar nekaj milisekund latence v povezavi.

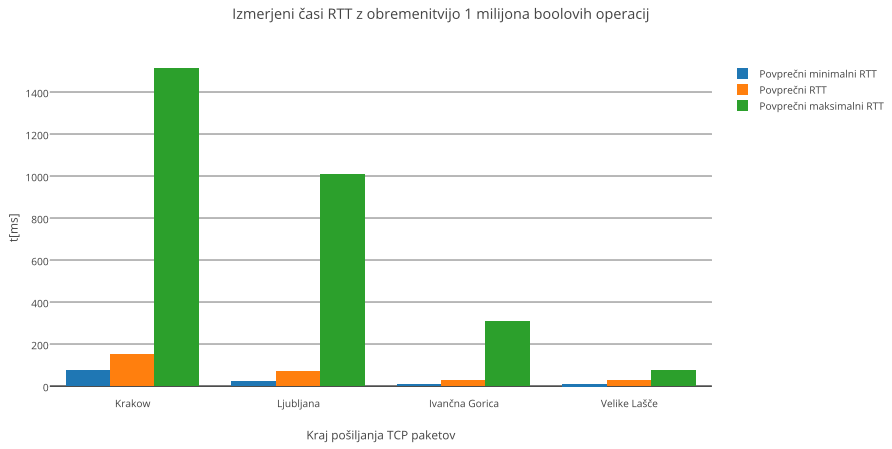
Pri testih B, D, F in L (tabela 5.2 in slika 5.7) smo uporabili še funkcijo za obremenitev PLC-ja in prav tako merili povprečen čas za odgovor PLC-ja, le da je bil še rahlo obremenjen (1 milijon logičnih operacij). Pričakovali smo, da se bodo časi podaljšali več, kot so se potem v praksi. Podaljševali so se za približno 1,35-krat.

Pri testu L smo testirali povezavo iz Velikih Lašč do Šiške v Ljubljani. Presenetljivo smo ugotovili, da so povprečni časi skoraj enaki testu F, izvedenem



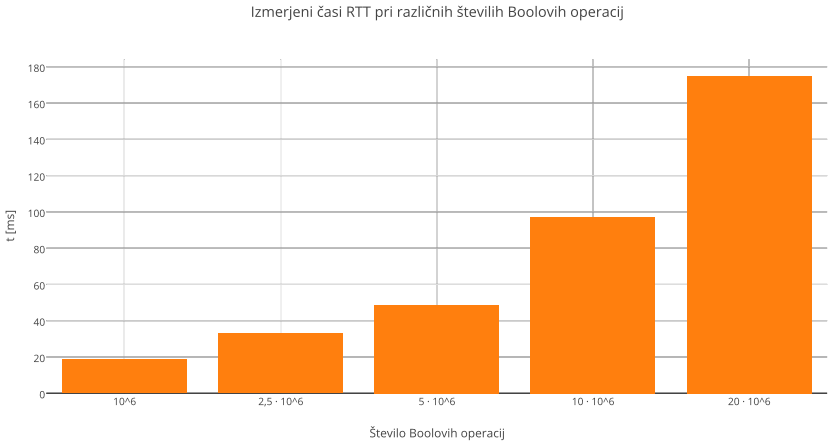
Slika 5.6: Rezultati testov brez obremenitve.

lokalno, in to kljub 30 km zračne razdalje [67] (v primerjavi z nekaj metrov lokalno). Razlog za to je verjetno, da je šla celotna komunikacija pri testu L po hitrem optičnem omrežju, hkrati pa je bil PLC priklopljen na zmogljivejši namenski usmerjevalnik in ne na 'multipraktični' modem. Poleg tega je treba še povedati, da sta bila klient in ta modem/usmerjevalnik pri lokalnih testih eno nadstropje narazen, med njima je bil torej strop in kar nekaj sten, kar je gotovo pripomoglo k slabšim rezulatom.



Slika 5.7: Rezultati testov z 1 milijonom boolovih operacij.

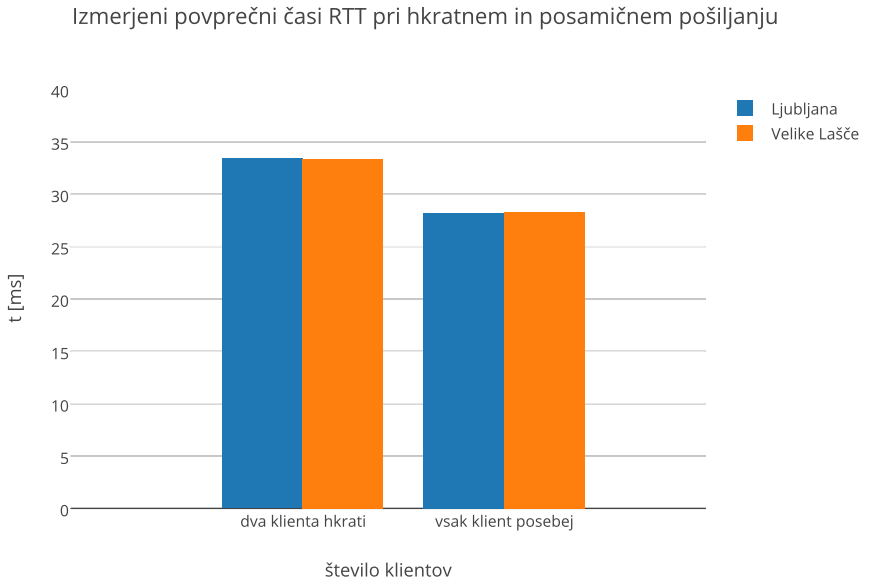
Ker nas je, kot že omenjeno, zanimalo, koliko boolovih operacij bi lahko opravili v času, ki ga paketi porabijo za pot do Krakova (okrog 600 km zračne razdalje), smo se lotili še lokalnega testiranja z močno povečanim bremenom (testi G, H, I in J (tabela 5.3 in slika 5.8)).



Slika 5.8: Rezultati testov z različnim številom boolovih operacij.

Ugotovili smo, da dobimo ekvivalenten povprečni RTT - okrog 100 ms - pri 10 milijonov boolovih operacij, kar je kar 10-krat več (9 milijonov logičnih operacij več). Tako smo zelo jasno videli, da je v našem primeru največji problem razdalja oz. latenca povezave in ne procesorska moč.
Preizkusili smo tudi hkratno pošiljanje dveh odjemalcev (slika 5.9). Iz rezultatov je po pričakovanjih razvidno, da se RTT pri hkratnem pošiljanju nekoliko poveča, a je temu verjetno tako predvsem zaradi tega, ker klient čaka na pravi paket in ne veliko zaradi obremenitve PLC-ja.
Pri čisto vseh testih smo opazili zelo velika odstopanja (max in min RTT, ki v nekaterih primerih odstopata tudi za več sekund), kar je lahko posledica dveh stvari. Prva je, da se izgubljajo paketi na poti in mora potem TCP pošiljati še enkrat, zmanjšati 'congestion window' in hitrost prenosa (TCP Tuning [68]). To je možno na daljših poteh (npr. v Krakov), saj je vmes veliko aktivnih naprav, vendar pa je na kratkih poteh zelo malo verjetno. To nas pripelje do drugega vzroka, ki je, da se na trenutke 'zamaši' in upočasni naš PLC - strežnik. Ker so precej velika odstopanja tudi pri lokalnem testiranju, je to precej verjeten razlog. Razlog, da se to dogaja, pa je bodisi pomanjkljivost v programu na PLC-ju bodisi pomanjkanje računskih zmogljivosti.

Faktor razlike med testom brez obremenitve in z obremenitvijo pri pošiljanju iz Ljubljane v Ivančno Gorico je:



Slika 5.9: Pošiljanje dveh odjemalcev hkrati (eden iz Velikih Lašč, drugi iz Ljubljane). PLC je bil v Ljubljani, obremenitev je bila 1 milijon boolovih operacij.

$$\frac{68,9}{51,1} \approx 1,35 \tag{5.1}$$

Enako velja za teste, ki so bili izvedeni lokalno:

$$\frac{25,3}{18,7} \approx 1,35 \tag{5.2}$$

5.6 Zaključek

Naučili smo se, da je povezava kmalu ozko grlo pri prenosu podatkov in da na to zelo vpliva njena kakovost. Tako se zelo pozna, če je ta optična, namesto bakrena. Zaključimo torej lahko, da je manj verjetno, da je ozko grlo na strežnikovi (v našem primeru PLC) ali pa na odjemalčevi strani, saj je omrežje veliko bolj omejujoče. Sploh če gre za prenos podatkov na daljše razdalje in če je vmes tudi brezžično omrežje.

Prav tako smo spoznali, da je velik problem pri hitrosti prenosa tudi, če program ni napisan optimalno, saj se v tem primeru hitrost prenosa močno zmanjša. Na začetku smo imeli namreč zaradi nepoznavanja jezika ST in PLC-jev na sploh

velike težave z našim strežnikom, ki je bil precej počasen. Te težave smo potem, ko smo si povsem razjasnili posebnosti jezika in celotno programsko kodo, odpravili, tako da smo lahko podatke pošiljali veliko hitreje.

Poglavje 6

Analiza hitrosti oblačne storitve Amazon S3

Andrej Česen, Denis Božović, Anže Lazar

6.1 Uvod

V tem članku smo analizirali oblačne storitve Amazon S3. Zanimale so nas hitrosti prenosa različnih bremen, odvisnosti hitrosti prenosa od vrste povezave in različnih internetnih ponudnikov ter shranjevanje podatkov na porazdeljenih strežnikih.

Oblačne storitve so v današnjih časih že globoko zasidrane v svetovnem spletu, vendar še vedno v visokem porastu. Kot ponudnika storitev smo si izbrali Amazon, saj je eden od največjih igralcev na tem področju in ponuja zanesljive in kvalitetne rešitve.

6.2 Oblačna spletna storitev Amazon S3

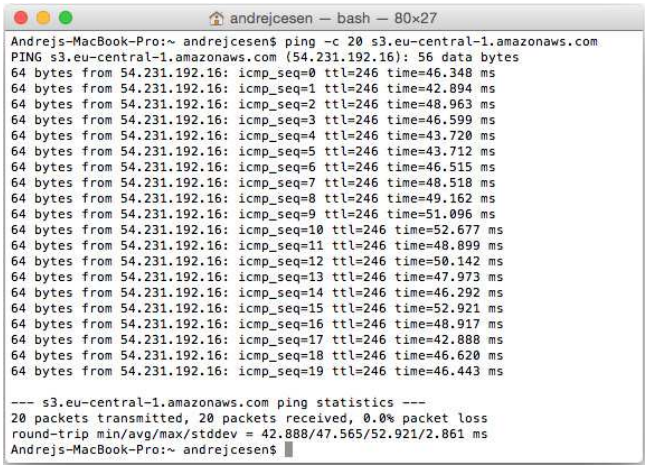
Amazon Simple Storage Service (Amazon S3) je oblačna spletna storitev za hranjenje podatkov. Primarno je namenjena razvijalcem programske opreme in IT kadru za hrambo objektov. Amazon S3 je del širše ponudbe Amazon Web Services (AWS) [69].

Testiranje smo izvajali na brezplačni različici AWS Free Tier, ki nudi uporabnikom 5GB prostega prostora in je omejen na 20.000 GET in 2.000 PUT HTTP zahtev za dostop do podatkov [70].

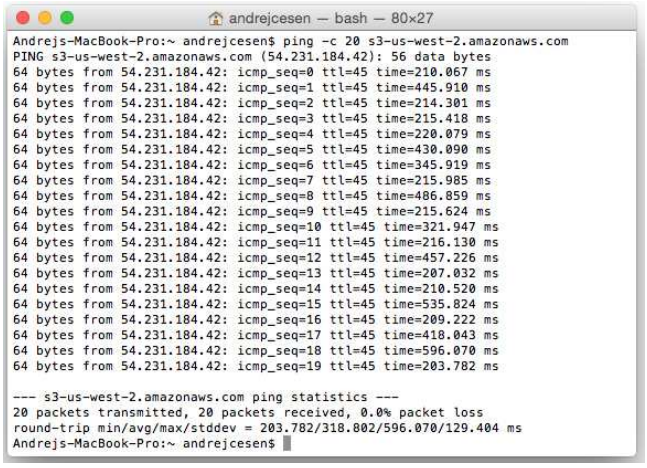
Po registraciji uporabnika je potrebno ustvariti vedro (bucket), kamor se bodo objekti (datoteke) nalagali. Vedro poleg naslovnega prostora definira tudi

fizično lokacijo hrambe objektov, zato se pri ustvarjanju le-tega izbere regijo, kjer bodo objekti gostovani.

Da bi omrežno zakasnitev minimizirali, smo za regijo gostovanja izbrali Frankfurt EU (eu-central-1). Izmerjen povprečen čas dostopa do strežnika v Frankfurtu na dostopni točki `s3.eu-central-1.amazonaws.com` je znašal **47,57ms** (slika 6.1), medtem ko je izmerjen čas dostopa za US West (Oregon) regijo na dostopni točki `s3-us-west-2.amazonaws.com` znašal **318,80ms** (slika 6.2).



Slika 6.1: Izvedba ukaza ping na URL `s3.eu-central-1.amazonaws.com`.



Slika 6.2: Izvedba ukaza ping na URL `s3-us-west-2.amazonaws.com`.

Razlike je torej približno za **271ms** v prid regiji EU, kar nam omogoča merjenje časa prenosa, na katerega bistveno manj vplivata čas vzpostavitve povezave in morebitne mrežne anomalije.

6.3 Breme

Izbrano breme je sintetičen model bremena - prenašanje datoteke v smeri iz oblačne storitve proti uporabniku. Datoteka je slika JPEG velikosti 1.037.183B (slika 6.3), ki predstavlja tipičen element bremena pri dostopu do spletnih strani. Kasneje smo ugotovili, da prenosi trajajo predolgo, zato smo v prid kvalitetnejšim rezultatom testirali še z enakim bremenom velikosti 100KB in 10KB.

Slika 6.3: Breme - slika JPEG velikosti 1.037.183B [71]

6.4 Testna skripta

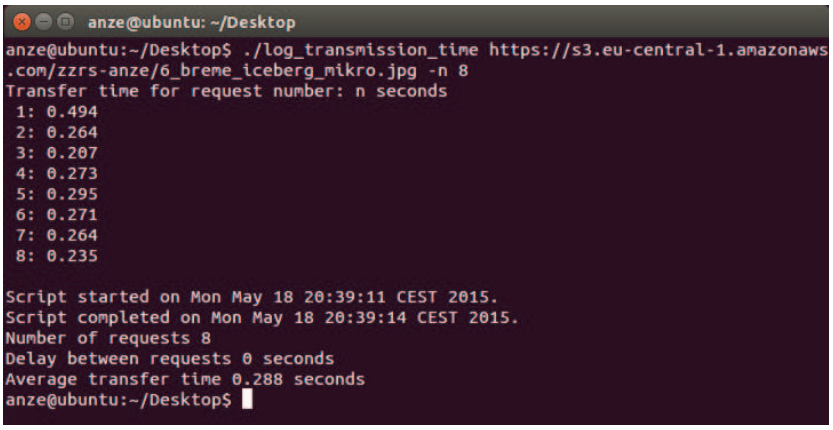
V namene testiranja časa prenosa smo implementirali skripto za ukazno lupino Bash. Obvezen argument pri proženju skripte je URL spletnega vira, kot opcijske argumente pa sprejme zakasnitev med posameznimi zahtevami v sekundah (stikalo `-d`) ter število zahtev (stikalo `-n`). Na standardni izhod zabeleži čas proženja skripte, celoten čas izvedbe posamezne zahteve, čas zaključka izvajanja, vrednosti podanih parametrov in povprečen čas izvedbe vseh zahtev. Rezultat proženja skripte je prikazan na sliki 6.4.

```
1  #!/usr/bin/env bash
2
3  ############     Parsing input parameters     ############
4  show_help() {
5      printf 'Usage: %s [options] <url>\n' "$(basename $0)"
6      printf -- '-d  Delay between requests in seconds.\n'
7      printf -- '-n  Number of requests.\n'
8  }
9
10 # Reset all variables that might be set.
11 URL=
12 delay=0
13 requests_count=1
14
15 while [ "$#" -gt 0 ]; do
16     case $1 in
17         -h|-\?|--help)   # Call a "show_help" function to display a
       synopsis, then exit.
```

```
18              show_help
19              exit
20              ;;
21          -d)
22              if [ "$#" -gt 1 ]; then
23                  delay=$2
24                  shift 2
25                  continue
26              else
27                  echo 'ERROR: Must specify a non-empty "-d DELAY"
    argument.' >&2
28                  exit 1
29              fi
30              ;;
31          -n)
32              if [ "$#" -gt 1 ]; then
33                  if (( $2 < 100 )); then
34                      requests_count=$2
35                      shift 2
36                      continue
37                  else
38                      echo 'ERROR: NUMBER_OF_REQUESTS must be less than
    100.' >&2
39                      exit 1
40                  fi
41              else
42                  echo 'ERROR: Must specify a non-empty "-n
    NUMBER_OF_REQUESTS" argument.' >&2
43                  exit 1
44              fi
45              ;;
46          --)             # End of all options.
47              shift
48              break
49              ;;
50          -*)
51              printf 'WARN: Unknown option (ignored): %s\n' "$1" >&2
52              ;;
53          ?*)             # Main parameter - URL
54              URL=$1
55              ;;
56          *)              # Default case: If no more options then break out of
     the loop.
57              break
58      esac
59
60      shift
61  done
62
63  if [ ! "$URL" ]; then
64      echo 'ERROR: "URL" not given. See --help.' >&2
65      exit 1
66  fi
67
68  ############    Measuring transmission time     ############
69  printf 'Script started on %s.\n\n' "$(date)"
70
71  export LC_NUMERIC="en_US.UTF-8"   # use decimal period
72
73  time_total=0
74  for (( i=0; i < requests_count; i++ )); do
75
76      # measure the total time, in seconds, that the operation (GET
     request) lasted
77      time_request=$(curl "$URL" -so /dev/null -w "%{time_total}")
78
79      printf 'Transfer time for request %2d: %.3f seconds\n' "$(( i+1 ))"
    "$time_request"
```

```
80
81      # add up the request time to the total request time
82      time_total=$(bc -l <<< "$time_total + $time_request")
83
84      # if the request is not the last, sleep for the specified delay
85      if (( i < requests_count-1 )); then
86          sleep "$delay"
87      fi
88  done
89
90  time_average=$(bc -l <<< "$time_total / $requests_count")
91
92  # print results to stdout
93  printf '\nScript completed on %s.\n'             "$(date)"
94  printf 'Number of requests: %d\n'               "$requests_count"
95  printf 'Delay between requests: %d seconds\n'   "$delay"
96  printf 'Average transfer time: %.3f seconds\n' "$time_average"
```

Programska koda 6.1: Izvorna koda testne skripte



Slika 6.4: Primer proženja skripte. Izpis je prilagojen za analizo rezultatov v MS Excelu.

6.5 Izvedba testov

Testno breme smo naložili na Amazon S3. Nastavili smo mu javne pravice dostopa, da ga lahko vsi prenašajo brez avtentikacije odjemalca. Tako smo lahko testirali prenos iste datoteke po želji tudi preko različnih lokalnih ponudnikov interneta, Amazon S3 pa je potreboval manj časa za obdelavo naše zahteve.

6.5.1 Dostop preko različnih internetnih ponudnikov

Teste smo izvajali preko različnih internetnih ponudnikov z različnimi povezavami:

- T-2 - optična povezva, 10/10 Mbps
- Siol - VDSL povezava, 20/5 Mbps
- Amis - ADSL povezava, 10/1 Mbps

Merili smo potovalne čase paketov (od nas do Amazonovih strežnikov in nazaj) s programom ping. Ponudnik **T-2** ima povprečen čas **16 ms**, ponudnik **Siol** ima **34 ms** in ponudnik **Amis** ima **48 ms**. Ker se hitrosti ponudnikov razlikujejo, se nekateri testi med seboj ne morejo absolutno primerjati, saj je prihajalo tudi do pol-sekundnih razlik (razlika je bolj očitna na večjih bremenih/datotekah).

6.5.2 Analiza meritev prenosov

Meritve smo izvedli v dveh sklopih:

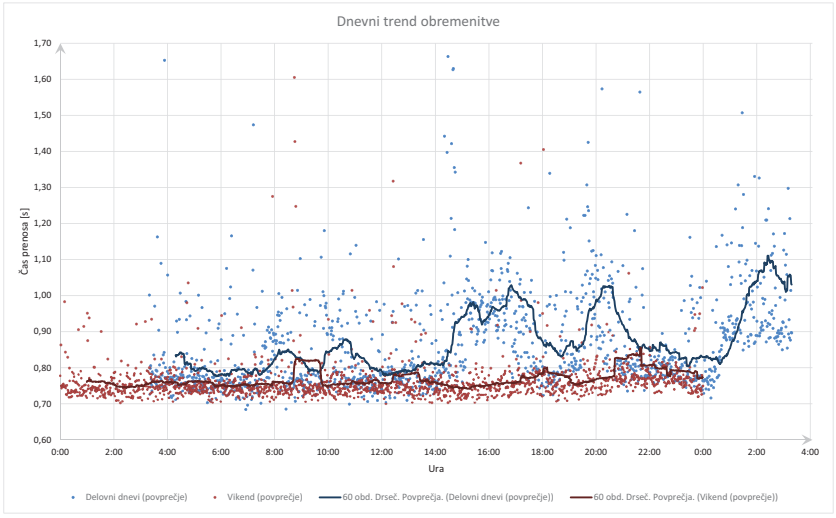
- izvajanje meritev na eno minuto, za iskanje dnevnega trenda obremenitve glede na uro v dnevu,
- izvajanje meritev z različnimi bremeni in frekvencami, za ugotavljanje prisotnosti predpomnenja.

Testiranje dnevnega trenda obremenitve

Test smo izvajali dva delovna dneva in čez vikend. Naše hipoteze so bile, da obstaja nek trend obremenitve čez dan in da so strežniki čez vikende manj obremenjeni, kot med tednom.

Na grafu 6.5 je s polno modro črto predstavljen trend obremenitve Amazonove storitve čez delavnik. Strežniki so najbolj obremenjeni med 15. in 17. uro, 19. in 21. uro ter 1. in 3. uro. Popoldne predvidevamo, da so storitve obremenjene zaradi povečanega obiska na njihovih strežnikih, ponoči pa se po navadi izvajajo vzdrževanja. To lahko predvidevamo, ker smo izbrali evropske strežnike za opravljanje meritev, ki strežejo zahtevam pretežno iz Evrope. Če bi izbrali strežnike v ZDA, tega ne bi mogli predvidevati, saj ti strežniki strežejo zahtevam iz celega sveta (saj so privzeta in najcenejša izbira).

S polno rdečo črto je predstavljen trend obremenitve Amazonove storitve čez dan vikenda. Zaznati je možno minimalne obremenitve, ki pa se ne morejo primerjati s tistimi med tednom.

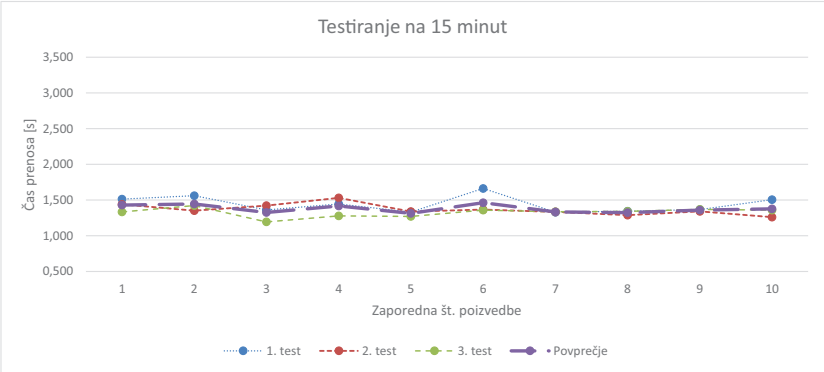


Slika 6.5: Iskanje trenda dnevne obremenitve med vikendom in med delavnikom (ISP: Siol).

Testiranje predpomnilnika - Testi na 15 in 10 minut

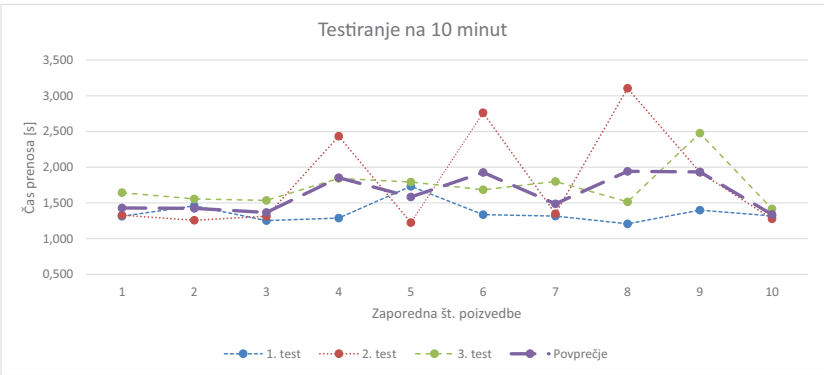
Izvedli smo različne teste prenosa bremena (sliki 6.6 in 6.7), da bi ugotovili ali frekvenca dostopov do vira vpliva na čas prenosa bremena oziroma na čas procesiranja zahteve. Vse meritve smo izvajali trikrat in rezultate povprečili.

Mislili smo, da bi moralo pogostejše poizvedovanje vplivati na hitrost dostopa in posledično prenos bremena. V prvem testu (slika 6.6) nismo opazili nič nenavadnega, saj smo take rezultate pričakovali.



Slika 6.6: Iskanje prelomne točke za prenos v predpomnilnik, test na 15 minut (ISP: Amis).

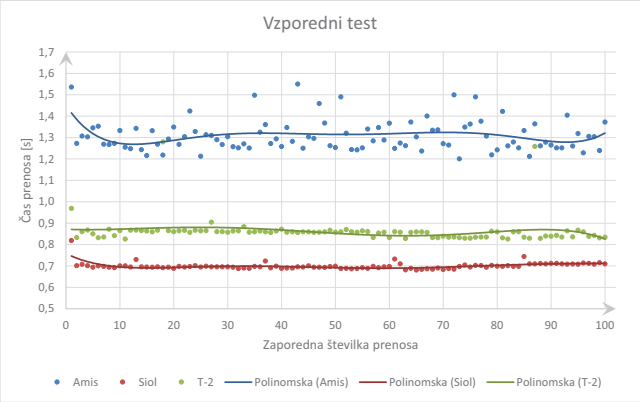
Frekvenco dostopa do bremena smo zmanjšali in pričakovali podobne rezultate (slika 6.7), kot pri prvem testiranju (slika 6.6). Odkrili pa smo, da včasih rezultati (časi dostopa) začnejo variirati.



Slika 6.7: Iskanje prelomne točke za prenos v predpomnilnik, test na 10 minut (ISP: Amis).

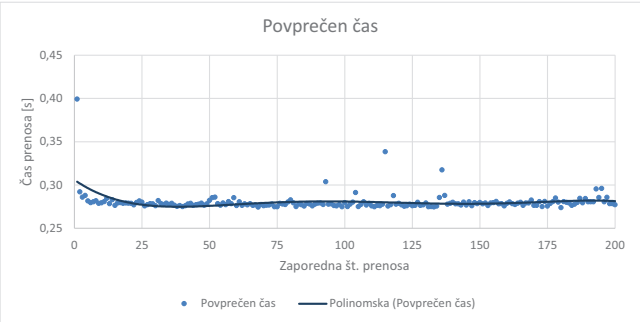
Testiranje predpomnilnika - Vzporedni testi

Na podlagi rezultatov, ki smo jih dobili iz prejšnjih testiranj, smo se odločili, da poskusimo strežnik čimbolj obremeniti. Pričakovali smo, da bo po dovoljšnjem številu prenosov slika prenešena v predpomnilnik, kar naj bi se poznalo v nenadnem zmanjšanju časa prenosa. Za tovrstni test smo ob istem času začeli prenašati sliko. Vsakega izmed odjemalcev posebej prikazuje barva na grafu 6.8.



Slika 6.8: Iskanje prelomne točke za prenos v predpomnilnik, vzporedni test iz treh različnih računalnikov, vsak pri drugem ponudniku internetnih storitev.

Zaradi različnih hitrosti naših lokalnih povezav se časi prenosa med seboj dokaj razlikujejo, vendar pa ni moč opaziti nikakršne nenadne pohitritve prenosov, zato smo se odločili za dodaten test z desetkrat manjšo sliko (velikosti 100kB), katerega rezultati so na grafu 6.9. Na grafu se na začetku vidi rahel padec časa prenosa, kar bi morda lahko namigovalo na predpomnilnik. Rahlo izstopajoča krivulja ponudnika Amis je hkrati vzbudila zanimanje, ali morda brezžična internetna povezava lahko v večji meri vpliva na pridobljene rezultate.



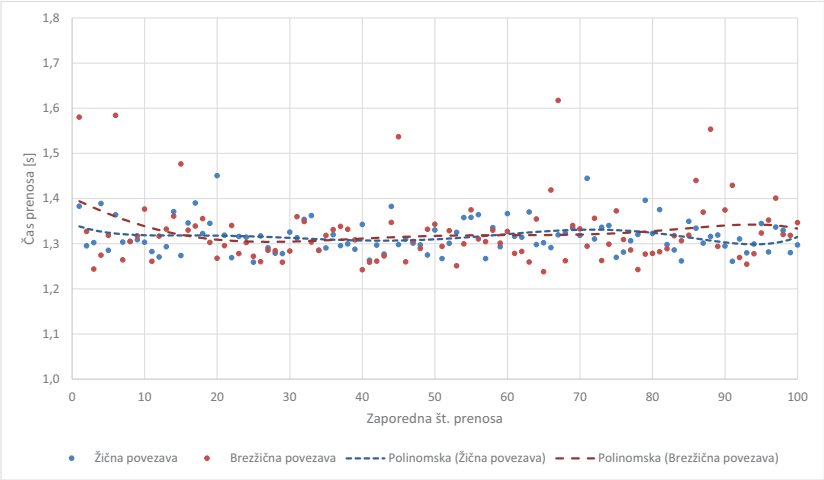
Slika 6.9: Iskanje prelomne točke za prenos v predpomnilnik z desetkrat manjšim bremenom. (ISP: Siol)

Primerjava brezžične in žične povezave v lokalnem omrežju

Zaradi nekonsistentnih časov prenosa na omrežnem priključku Amis preko brezžične povezave, smo oblikovali hipotezo, da visoko razpršenost časov prenosa povzročajo lokalne motnje na brezžičnem omrežju. Da bi hipotezo preverili, smo ustvarili naslednji testni scenarij:

1. Najprej smo izklopili vse odjemalce priklopljene na omrežni priključek.
2. Nato smo povezali računalnik z usmerjevalnikom preko žične povezave (Ethernet).
3. Naredili smo 5 ponovitev testiranj; 100 prenosov testnega bremena na izbran računalnik z ukazom `log_transmission_time -n 100`.
4. Nato smo povezali računalnik z usmerjevalnikom preko brezžične povezave (Wi-Fi, protokol 802.11n, zračna oddaljenost približno 5m).
5. Naredili smo 5 ponovitev testiranj; 100 prenosov testnega bremena na izbran računalnik z ukazom `log_transmission_time -n 100`.

Ob vsakem prenosu smo zabeležili čase prenosa, rezultati pa so prikazani na grafu 6.10.

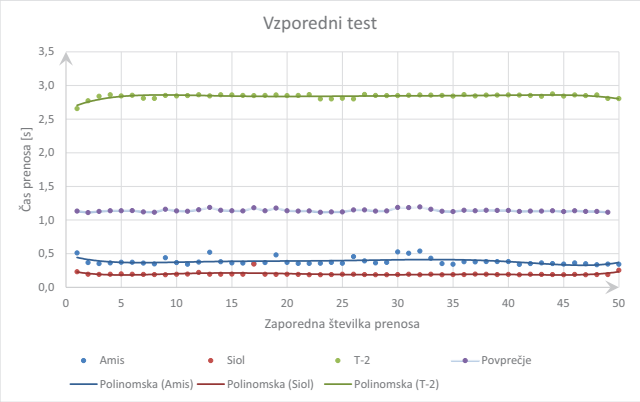


Slika 6.10: Podrobnejše testiranje z omrežnega priključka Amis.

Časi prenosa po brezžični povezavi imajo pri določenih točkah visoka odstopanja od povprečja navzgor. To lahko pripišemo lokalnim motnjam na brezžičnem omrežju (potreba po ponovnem pošiljanju paketov). Razlika je še posebej opazna, če primerjamo čase prenosa s tistimi po žičnem omrežju. Pri žični povezavi prihaja do precej manj odstopanj in povprečje je tudi bolj stabilno. Torej lahko potrdimo domnevo, da je do visokih deviacij pri časih prenosa prihajalo zaradi lokalnih motenj na brezžični povezavi - naša hipoteza je potrjena.

Testiranje predpomnilnika - Vzporedni testi na zelo majhnem bremenu

Na koncu smo testirali še z izredno majhno sliko velikosti 10kB. Rezultati testa so na grafu 6.11. Iz grafa je razvidno, da je povprečje časov prenosa bolj ali manj konstantno, zato domnevamo, da imamo nek vir (bodisi računska zmogljivost, odzivnost strežnika, hitrost prenosa, ...) omejen. Še vedno ni mogoče opaziti skokov v hitrost, ki bi jih lahko povezovali s predpomnilnikom.



Slika 6.11: Iskanje prelomne točke za prenos v predpomnilnik, vzporedni test iz treh različnih računalnikov, vsak pri drugem ponudniku internetnih storitev.

6.6 Metrika zmogljivosti

Merili smo povprečen čas prenosa datoteke iz oblaka k uporabniku. Torej skupen čas $T(T1 + T2 + T3)$, ki ga sestavljajo:

- $T1$: čas dostopa do storitve
- $T2$: čas procesiranja zahteve
- $T3$: čas prenosa datoteke iz oblaka k uporabniku

$T1$ smo v grobem ocenili glede na čas, ki ga potrebujejo paketi ukaza `ping s3.eu-central-1.amazonaws.com -t` v ukaznem pozivu (terminalu), da prepotujejo pot do strežnika in nazaj. Ker se internetni ponudniki med seboj razlikujejo, smo pričakovali tudi različne čase dostopa do strežnika. Najdaljši povprečni čas dostopa je trajal 48ms, tako da smo ta rezultat prepolovili in tako določili $T1 = \mathbf{24ms}$.

Nas je zanimal predvsem čas $T2$ - koliko časa preteče preden se vzpostavi prenos datoteke. Opazovali smo, ali frekvenca povpraševanja po datoteki vpliva na čas procesiranja zahteve.

Ker lahko merimo le skupen čas T, smo iz tega morali sklepati na čas $T2$. Čas $T3$ lahko približno ocenimo, zato smo čas dostopa do strežnika in prenos same datoteke (1MB / zakupljena hitrost prenosa pri ISP) odšteli od skupnega časa prenosa T in tako dobili okviren čas: $T2 = T - (T1 + T3)$.

Meritve smo izvajali večkrat, da smo lahko izračunali povprečen čas prenosa datoteke (ta znaša **1,7s** pri Amisu, **0,7s** pri Siolu in **2,8s** pri T-2), ki smo ga upoštevali kot metriko zmogljivosti oblačne storitve.

Za potrebe testiranja smo porabili okoli 20.000 zahtevkov, od tega je bilo približno $\frac{1}{4}$ drugih testiranj, ki smo jih porabili za vzpostavljanje testnih scenarijev.

6.7 Zaključek

Analizirali smo oblačno storitev Amazon S3. Na začetku nas je zanimalo, ali storitev uporablja predpomnjenje za hitrejše delovanje, med testiranjem pa sta se nam porodili še dve vprašanji; Ali obstaja trend dnevne obremenitve in ali način povezave v lokalno omrežje vpliva na hitrost prenosov bremena.

Pri testiranju predpomnilnika spreminjanje frekvence povpraševanja ni imelo opaznega vpliva na čas prenosa bremena. Iz tega lahko sklepamo na eno izmed naslednjih trditev:

1. Na brezplačnem računu imamo omejene vire.
2. AWS S3 nima funkcionalnosti predpomnjenja.
3. Breme se je premaknilo v predpomnilnik, toda zaradi prevelikega časa prenosa to ni bilo zaznano.

Ugotovili smo tudi, da obstaja dnevni trend obremenitve. Med tednom je storitev med dolo?enimi urami bolj obremenjena, kot sicer. Med vikendom je storitev enakomerno in manj obremenjena.

Testi so pokazali, da je način povezave v lokalno omrežje pomemben. Preko brezžične povezave prihaja do večjih dostopnih časov, ker je povezava nekoliko manj stabilna.

Poglavje 7

Testiranje zanesljivosti in zmogljivosti spletne trgovine v oblačni storitvi

Uroš Marolt, Gregor Stopar, Dejan Jusufović

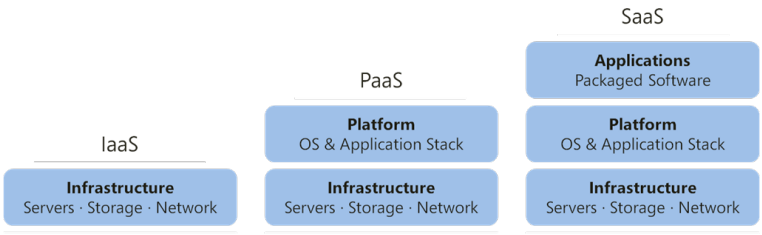
7.1 Uvod

Skozi naslednje razdelke je predstavljena zanesljivost in zmogljivost spletne trgovine, ki uporablja oblačno storitev za svoje delovanje. Oblačna storitev za razliko od tradicionalnega modela, kjer organizacija sama poskrbi za nakup in vzdrževanje opreme, omogoča deljenje opreme oz. resursov v lasti ponudnika oblačne storitve, ter njihovo učinkovito izrabo. Projekt prav tako uporablja aplikacijski server, s čimer sta prezentacijski del in poslovna logika popolnoma ločena. Za ta namen smo razvili preprosto aplikacijo spletne trgovine, preko katere bomo testirali zanesljivost in zmogljivost ponudnika oblačne storitve. Kljub preprostosti aplikacije, le-ta vsebuje vse osnovne lastnosti spletne trgovine, ki jih uporabnik pričakuje. V nadaljevanju so predstavljeni podrobni testi oblačne storitve in kako le-ti vplivajo na uporabniško izkušnjo.

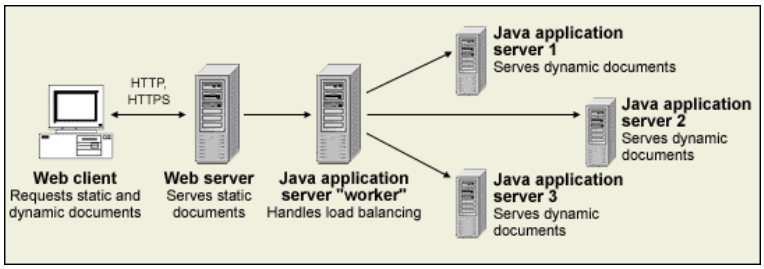
7.2 Uporabljena tehnologija

7.2.1 Oblačna storitev

Oblak omogoča učinkovito rabo in deljenje računalniških resursov s strani več uporabnikov. Resursi niso samo deljeni med uporabniki, ampak se tudi dodeljujejo po potrebi. Ta premik v načinu organizacije aplikacij prinaša veliko prednosti, pa tudi slabosti. Ker organizacijam ni potrebno več skrbeti za nakup in vzdrževanje opreme, se večino bremena preusmeri v sam razvoj aplikacij in prilagajanje trenutnemu trgu. slabost pa se kaže v tem, da organizacija nima lokalno prisotnih podatkov ter kontrole nad infrastukturo aplikacije, kar pomeni da se za nemoteno delovanje popolnoma zanaša na ponudnika oblačne storitve. Oblačne storitve niso med seboj enakovredne (za različne tipe glej sliko 7.1). Pospešen razvoj v zadnjih letih je izoblikoval tri možne tipe.



Slika 7.1: Tipi oblačnih storitev.



Slika 7.2: Vloga aplikacijskega strežnika.

Kot ponudnika oblačne storitve smo izbrali Openshift Online Public PaaS [72], ker ima naša skupina z njim največ izkušenj. Brezplačni načrt ponudnika nam omogoča uporabo treh malih koles. Ponudnik OpenShift definira malo kolo kot varen ekosistem, ki ima dodeljeno procesno enoto, pomnilnik in določeno mrežno zmogljivost. Mala kolesa omogočajo razvoj več aplikacij ali pa razvoj ene same, kjer se dodatna kolesa (največ tri) dodelijo po potrebi, glede na povečan promet. Lokacija našega strežnika: 54.166.211.18 - United States - Virginia - Ashburn (Amazon Technologies Inc.) Tehnične specifikacije posameznega kolesa so sledeče:

- 512MB delovnega pomnilnika,
- 100MB swap prostora,
- 1GB trajnega prostora,
- Red Hat linux plaforma, ki jo poganjajo na Amazon EC2 strežnikih.

7.2.2 Aplikacijski strežnik

V projektu bomo uporabili tudi aplikacijski strežnik, saj nam omogoča ločitev poslovne logike od prezentacije in je pogost pristop pri izdelavi spletnih aplikacij, kot je spletna trgovina. Aplikacijski strežnik izpostavlja poslovno logiko za uporabo client aplikacijam, preko različnih protokolov, med drugim tudi HTTP, medtem ko se spletni strežnik primarno uporablja za pošiljanje HTML-ja brskalnikom. Vizualni prikaz vloge aplikacijskega stežnika je prikazan na sliki 7.2.

Uporabljeni strežnik v projektu je WildFly, pretežno zaradi njegove razširjenosti in stabilnosti ter proste dostopnosti. WildFly se je v preteklosti imenoval JBoss ter je v celoti napisan v programskem jeziku Java. Med drugim tudi implementira Java EE specikacijo in teče na večih platformah, npr. Windows in Unix.

7.2.3 Podatkovna baza

Uporabljena podatkovna baza v projektu je MySql 5.5. Izbrali smo jo, ker je prosto dostopna ter zelo razširjena relacijska podatkovna baza. Kljub nekaterim slabostim kot sta npr. slabša stabilnost in slabša zmogljivost pri velikem prometu, so prednosti te relacijske baze veliko večje, kot njene slabosti.

7.2.4 Spletna trgovina

Aplikacijo spletne trgovine smo razvili sami in temelji na REST arhitekturi. Tako imenovane RESTful aplikacije uporabljajo HTTP zahteve za kreiranje, posodabljanje in brisanje podatkov. Odločitev za to arhiteturo izhaja iz dejstva, da je neodvisna od platforme, neodvisna od uporabljenih programskih jezikov in je standardizirana saj uporablja HTTP.

7.3 Opis testiranja

7.3.1 Aplikacija

Najbolj pomembni parametri, ki jih bomo testirali pri delovanju aplikacije so:

- število hkratnih uporabnikov/odprtih košaric oz. nakupov (količina uporabnikov),
- hitrost obdelave podatkov/posameznih zahtev (hitrost odziva),
- razliko v odzivnosti pri tem, ko se produkti nahajajo v bazi ter ko so v pomnilniku.

Za ta namen se ponuja več orodij (kot je npr. SoapUI), nekaj testov pa smo napisali sami.

7.3.2 Strežnik

Pri strežniku nas bodo zanimali sledeči parametri:

- obremenitev delovnega pomnilnika,
- obremenitev procesne enote,
- obremenitev omrežja.

Pri tej nalogi bomo uporabili sledeča orodja:

- Autobench,
- Server Benchmark.

7.4 Model aplikacije

Kot je prikazano na sliki 7.3 vidimo, da imamo entitete aplikacije razdeljene v tri skupine: Products, Users in Orders.

7.4.1 Products

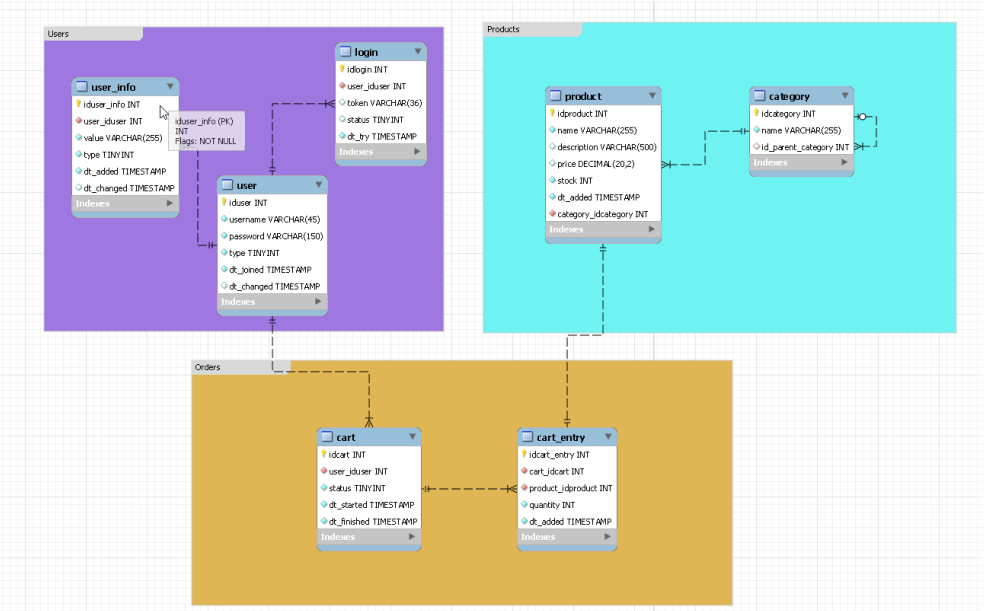
V skupini products imamo entiteti product in category. Kot že imeni povesta, hranita id in ime produktov ter kategorij in nadkategorij, v product pa imamo še podatke o ceni, zalogi, datumu ter opisu produkta.

7.4.2 Users

V skupini users se nahajajo entitete user_info, login in user. User_info hrani id, informacije, vrednost ter datum, ko je bil uporabnik dodan in nazadnje spremenjen. Tabela Login hrani id logina, id uporabnika, status in čas prijave. User hrani id, uporabniško ime, geslo, tip, datum prijave ter datum spremembe uporabnika.

7.4.3 Orders

Skupina orders vsebuje cart in cart_entry. Entiteta cart vsebuje id uporabnika ter id, status, čas začetka in čas konca košarice. Cart_entry vsebuje id košarice ter id košarice in id produkta, število le-tega ter čas, ko je bil produkt dodan.



Slika 7.3: Model postavljene aplikacije.

7.5 Implementacija

Naslednja podpoglavja vsebujejo opis REST storitev, ki jih uporablja aplikacija. Naslov spletne aplikacije: `http://webstore-tposi.rhcloud.com/WebstoreWeb`.

7.5.1 Login API

Logiranje

Pot: /api/login
Tip: POST
Sprejme: application/xml
Vrne: application/xml
Primer zahtevka:

```
<loginRequest>
  <username>admin</username>
  <password>adminpass</password>
</loginRequest>
```

Primer odgovora:

```
<loginResponse>
  <token>48965dd4-bb52-423f-8824-91672a087f5c
```

```
  </token>
</loginResponse>
```

Registriranje novega uporabnika

Pot: /api/login/register
Tip: POST
Sprejme: application/xml
Vrne: application/xml
Primer zahtevka:

```
<registerRequest>
  <username>Janez</username>
  <password>Novak</password>
  <email>janez.novak@gmail.com</email>
</registerRequest>
```

Primer odgovora:

```
<registerResponse>
  <token>48965dd4-bb52-423f-8824-91672a087f5c
  </token>
</registerResponse>
```

Izpis obstoječega uporabnika

Opis: Token podan v primeru spodaj predstavlja token, ki ga vrne vpis.
Pot: /api/login/logout
Tip: POST
Sprejme: application/xml
Vrne: application/xml
Primer zahtevka:

```
<request>
  <token>e5a3af93-1fee-4f99-ad95-908a9fe6ee1c
  </token>
</request>
```

Primer odgovora:

```
<response>
  <message>Logout successful!</message>
</response>
```

7.5.2 Profile API

Prikaz Profila

Pot:/api/profile
Tip: POST
Sprejme: application/xml
Vrne: application/xml
Primer zahtevka:

```
<request>
  <token>c0cfa34a-1113-46f1-bd84-e97738917ecd
  </token>
</request>
```

Primer odgovora:

```
<profileResponse>
  <username>Janez</username>
  <dtJoined>28-11-2014 01:11:48</dtJoined>
  <userInfos>
    <value>janez.novak@gmail.com</value>
  </userInfos>
</profileResponse>
```

Spreminjanje Profila

Pot: /api/profile/change
Tip: POST
Sprejme: application/xml
Vrne: application/xml
Primer zahtevka:

```
<profileChangeRequest>
  <token>c0cfa34a-1113-46f1-bd84-e97738917ecd
  </token>
  <type>contact\_number</type>
  <value>041564789</value>
</profileChangeRequest>
```

Primer odgovora:

```
<response>
  <message>contact\_number added!</message>
</response>
```

Dovoljeni tipi sprememb:
address_street, address_country, address_
zipcode, name, address_town, email,
contact_number, lastname, address_streetnr, password

7.5.3 Category API

Povpraševanje po kategorijah

Opis: Storitev vrne kategorije spletne trgovine.
Pot: /api/category
Tip: GET
Vrne: application/xml
Primer zahtevka:

```
http://webstore-tposi.rhcloud.com/WebstoreWeb/api/
    category
```

Primer odgovora:

```
<categoryResponse>
  <category>
    <idcategory>1</idcategory>
    <name>Računalništvo</name>
    <idParentCategory/>
  </category>
  <category>
    <idcategory>2</idcategory>
    <name>Avdio-video</name>
    <idParentCategory/>
  </category>
  <category>
    <idcategory>3</idcategory>
    <name>Foto</name>
    <idParentCategory/>
  </category>
  <category>
    <idcategory>4</idcategory>
    <name>Telefonija</name>
    <idParentCategory/>
  </category>
  <category>
    <idcategory>5</idcategory>
    <name>Komponente</name>
    <idParentCategory>1</idParentCategory>
```

```
  </category>
  <!-- and more... -->
</categoryResponse>
```

7.5.4 Product API

Dodajanje produkta v košarico

Pot: /api/product/{idCategory}
Tip: GET
Vrne: application/xml
Primer zahtevka:

```
http://webstore-tposi.rhcloud.com/WebstoreWeb
/api/products/7
```

Primer odgovora:

```
<productResponse>
  <product>
    <idProduct>44</idProduct>
    <name>Asus namizni računalnik (M12AD) i7-</name>
    <price>979.00</price>
    <stock>3</stock>
  </product>
  <product>
    <idProduct>45</idProduct>
    <name>DELL namizni računalnik Optiplex
    9020MT i7-</name>
    <price>1092.00</price>
    <stock>4</stock>
  </product>
  <product>
    <idProduct>46</idProduct>
    <name>HP namizni računalnik 490PD
    MTG2, i7-4790, 8 GB, 1 TB</name>
    <price>884.00</price>
    <stock>6</stock>
  </product>
  <product>
    <idProduct>47</idProduct>
    <name>HP namizni računalnik 400PD
    MTG2 i3 4150 4GB 500GB</name>
    <price>451.00</price>
    <stock>7</stock>
```

```
28   </product>
29 </productResponse>
```

7.5.5 Cart API

Izpis trenutne košarice

Pot: /api/cart
Tip: POST
Sprejme: application/xml
Vrne: application/xml
Primer zahtevka:

```
1 <request>
2   <token>f12b0164-1b14-4296-826e-116400f46927</token>
3 </request>
```

Primer odgovora:

```
1  <cartResponse>
2    <cartEntry>
3      <idProduct>44</idProduct>
4      <itemName>Asus namizni računalnik (M12AD) i7-</
     itemName>
5      <price>979.00</price>
6      <quantity>1</quantity>
7    </cartEntry>
8    <cartEntry>
9      <idProduct>45</idProduct>
10     <itemName>DELL namizni računalnik
11     Optiplex 9020MT i7-</itemName>
12     <price>1092.00</price>
13     <quantity>1</quantity>
14   </cartEntry>
15   <priceSum>2071.00</priceSum>
16 </cartResponse>
```

Dodajanje artikla v košarico

Pot: /api/cart/add
Tip: POST
Sprejme: application/xml
Vrne: application/xml
Primer zahtevka:

```
<addToCartRequest>
  <token>f12b0164-1b14-4296-826e-116400f46927</token>
  <idproduct>44</idproduct>
  <quantity>1</quantity>
</addToCartRequest>
```

Primer odgovora:

```
<response>
  <message>Added 1 products with product
  id 44 to cart!</message>
</response>
```

Odstranjevanje artikla iz košarice

Pot: /api/cart/remove
Tip: POST
Sprejme: application/xml
Vrne: application/xml
Primer zahtevka:

```
<removeFromCartRequest>
  <token>f12b0164-1b14-4296-826e-116400f46927</token>
  <idproduct>44</idproduct>
  <quantity>1</quantity>
</removeFromCartRequest>
```

Primer odgovora:

```
<response>
  <message>Removed 1 products with id 44 from cart!</
   message>
</response>
```

Zaključek košarice

Pot: /api/cart/checkout
Tip: POST
Sprejme: application/xml
Vrne: application/xml

Primer zahtevka:

```
<request>
```

```
  <token>f12b0164-1b14-4296-826e-116400f46927</token>
</request>
```

Primer odgovora:

```
<response>
  <message>Finished!</message>
</response>
```

Izpis starih zaključenih košaric

Pot: /api/cart/history
Tip: POST
Sprejme: application/xml
Vrne: application/xml
Primer zahtevka:

```
<request>
  <token>f12b0164-1b14-4296-826e-116400f46927</token>
</request>
```

Primer odgovora:

```
<checkoutHistoryResponse>
  <finishedCart>
    <cartEntries>
      <idProduct>44</idProduct>
      <itemName>Asus namizni računalnik
      (M12AD) i7-</itemName>
      <price>979.00</price>
      <quantity>0</quantity>
    </cartEntries>
    <cartEntries>
      <idProduct>45</idProduct>
      <itemName>DELL namizni računalnik
      Optiplex 9020MT i7-</itemName>
      <price>1092.00</price>
      <quantity>1</quantity>
    </cartEntries>
    <priceSum>1092.00</priceSum>
  </finishedCart>
  <finishedCart>
    <cartEntries>
      <idProduct>44</idProduct>
      <itemName>Asus namizni računalnik
```

```
23        (M12AD) i7-</itemName>
24        <price>979.00</price>
25        <quantity>1</quantity>
26      </cartEntries>
27      <priceSum>979.00</priceSum>
28    </finishedCart>
29  </checkoutHistoryResponse>
```

7.6 Benchmarks

7.6.1 Referenčni GET dostop s povezavo na bazo

Namen je primerjava pri referenčnem GET dostopu, ko pri prvem dostopu naložimo šifrant v pomnilnik in ko do le-tega dostopamo v bazi. Merili bomo odzivni čas. Povezava: http://webstore-tposi.rhcloud.com/WebstoreWeb/api/category
Benchmark utility: SoapUI LoadTest
Trajanje benchmarka: 60 sekund

Hipoteza

Predpostavimo, da dobimo s šifrantom v pomnilniku boljše rezultate, saj je dostopni čas do pomnilnika manjši.

Rezultati

	Minimum	Maximum	Average	# of tries	tries per second	bytes	bytes per second
Šifrant v bazi	144	388	161,73	329	5,45	986013	16346
Šifrant v pomnilniku	130	343	141,95	329	5,47	986013	16398

Komentar

Pri rezultatih vidimo, da se je naša napoved uresničila in smo dejansko dobili boljše čase pri primeru, ko je šifrant v pomnilniku. V povprečju je dostop hitrejši za malo manj kot 20ms, minimalni čas je bil hitrejši za 14ms, maksimalni pa za 45ms.

7.6.2 Maximalno število prijav na sekundo

Povezava: http://webstore-tposi.rhcloud.com/WebstoreWeb/api/login

Za benchmark smo napisali svojo kodo. Zgenerirali smo 1500 testnih uporabnikov, ki smo jih nato uporabili pri testiranju zmogljivosti (benchmarku). Koda požene več niti, vsaka nit pa pošilja prijavne zahteve na strežnik. Hitrost kontroliramo z določanjem števila niti in časom spanja niti po vsakem zahtevku.

Testiranje se je izvajalo na:
Intel i5-4440 CPU @ 3,1Ghz
8GB RAM
64-bit Windows 8.1
10/10 Mbit optična povezava

Koda benchmarka se nahaja na:
https://github.com/BizzyIzDizzy/ZZRS_Webstore/blob/master/WebstoreTest/src/com/webstore/test/UserFrameworkTest.java

Rezultati in komentar

Pri testiranju smo ugotovili, da je zgornja meja, pri kateri aplikacija še deluje približno 30 zahtevkov na sekundo. Če jih je več JavaEE strežniku (Wildfly) zmanjka upravljanih povezav do MySQL baze. Na strežniku je nastavljeno, da upravlja z maksimalno 10-imi povezavami do baze, kar je priporočljivo za Openshift z 1GB pomnilnika in eno jedrnim 1Ghz procesorjem. Pri vseh nadaljnjih testiranjih z različnimi velikostmi bazena povezav (connection pool) smo prišli do podobnih številk, čeprav pri nekaterih testiranjih ni bilo več omejitve povezav na bazo (s preko 50-imi upravljanimi povezavami), nam še vedno ni uspelo preseči meje 30 zahtevkov na sekundo. Prihajalo je do raznih napak na strežniku (Broken pipi - Wildfly-MySQL, java OutOfMemory itd.)

7.6.3 Test zmogljivost CPE in pomnilnika s šifrantom v pomnilniku

Glede na to, da nam server WildFly in naša postavljena aplikacija zasedeta približno 160MB pomnilnika, 100 kategorij z 2000 produkti pa 251MB, nam na voljo za uporabniške interakcije ostane 101MB.

Amazon ima povprečno vrednost produkta 15$ in povprečno vrednost naročila 47,31$. Iz tega lahko izračunamo povprečno število produktov v košarici: 3

Iz statistične analize je razvidno tudi to, da ima v povprečju kupec en artikel več v košarici preden opravi nakup.

Testne akcije(breme) smo zato določili tako, da izvedemo 4x dodaj v košarico, 4x preveri košarico, 1x odstrani iz košarice in 1x preveri stanje.

V povprečju povečajo delovni spomin aplikacije iz 411MB na 414MB - delta=3MB (thread overhead (thread stack) + local thread data (heap data specific to thread) = 0,5MB + 2,5MB).

Uporabljena orodja so bila ps, top in /proc direktorij na linux sistemu.
Koda benchmarka se nahaja na naslovu:
https://github.com/BizzyIzDizzy/ZZRS_Webstore/blob/master/WebstoreTest/src/com/webstore/test/CartFrameworkTest.java

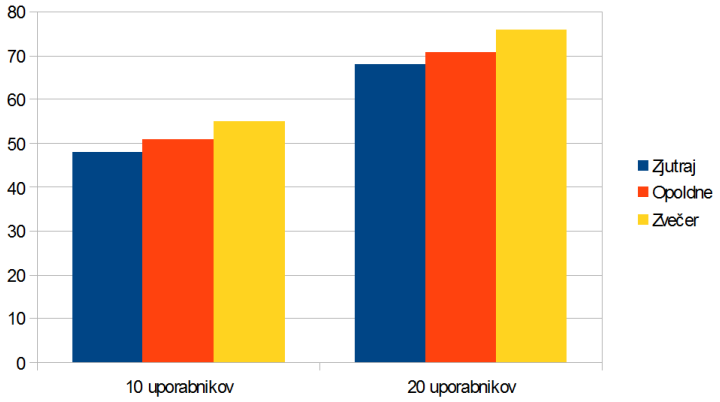
Hipoteza

Torej če predpostavimo, da povprečen uporabnik porabi 3MB delovnega spomina ko opravlja nakup, pomeni da lahko strežemo 33im uporabnikom naenkrat preden nam Openshift resetira strežnik (če smo več kot X časa na meji pri 512MB samodejno resetirajo vse aplikacije na našemu cloudu, kjer je X spremenljivka ki se zmanjšuje v odvisnosti od bremena na njihove strežnike/posamezne gruče).

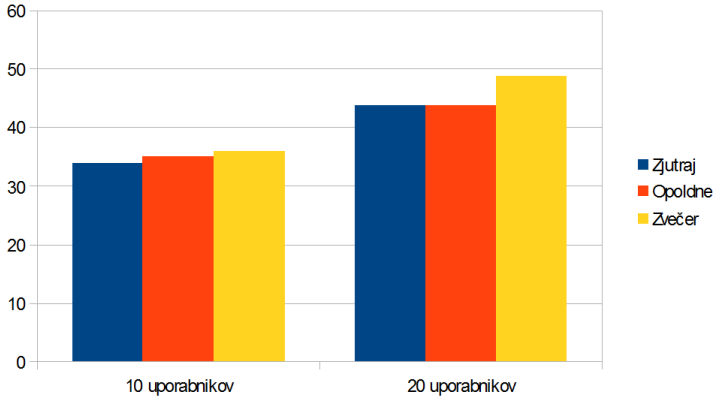
Pripravili smo benchmark, ki testira ravno to. Zasnovan je tako da N uporabnikov naenkrat opravlja nakup. Hkrati testiramo tudi breme na procesor.

Rezultati

Št. testa	Št. sočasnih uporabnikov	poraba(MB)	zasedenost CPU(%)	skupni delovni pomnilnik (MB)	Čas med zahtevki(ms)	Datum in čas
1	10	34	26,7	445	200	2. Maj 2015 - 6:08
2	10	48	49,3	459	100	2. Maj 2015 - 6:12
3	20	44	46,7	455	200	2. Maj 2015 - 6:16
4	20	68	66,3	479	100	2. Maj 2015 - 6:23
5	10	35	25,3	446	200	2. Maj 2015 - 12:04
6	10	51	48,7	462	100	2. Maj 2015 - 12:09
7	20	44	45,9	455	200	2. Maj 2015 - 12:16
8	20	71	51,3	482	100	2. Maj 2015 - 12:19
9	10	36	26,5	447	200	2. Maj 2015 - 20:23
10	10	55	53,3	466	100	2. Maj 2015 - 20:29
11	20	49	49,1	460	200	2. Maj 2015 - 20:36
12	20	76	72,8	487	100	2. Maj 2015 - 20:41



Slika 7.4: Graf porabe pomnilnika pri medprihodnih časih zahtev 100ms.

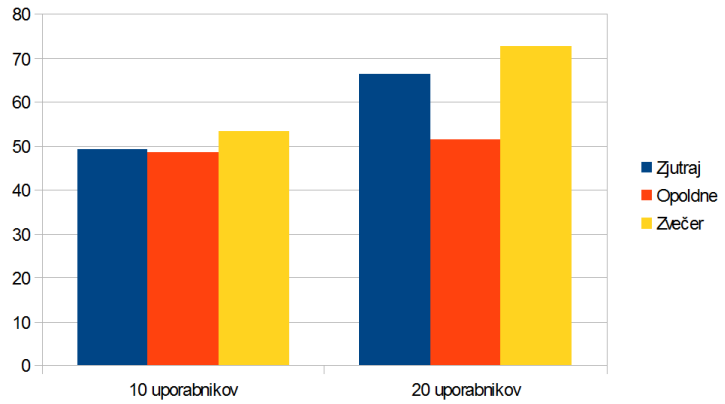


Slika 7.5: Graf porabe pomnilnika pri medprihodnih časih zahtev 200ms.

Pri rezultatih nas seveda zanimata poraba pomnilnika in zasedenost CPE. Za lažjo predstavitev rezultatov imamo spodaj podane štiri grafe in sicer se prva dva nanašata na porabo pomnilnika, druga dva pa na zasedenost CPE. Pri grafih smo imeli prisotne različne obremenitve in sicer smo testirali postavitev z 10 ter 20 uporabniki in z medprihodnimi časi zahtev 100ms ter 200ms.

Komentar porabe pomnilnika

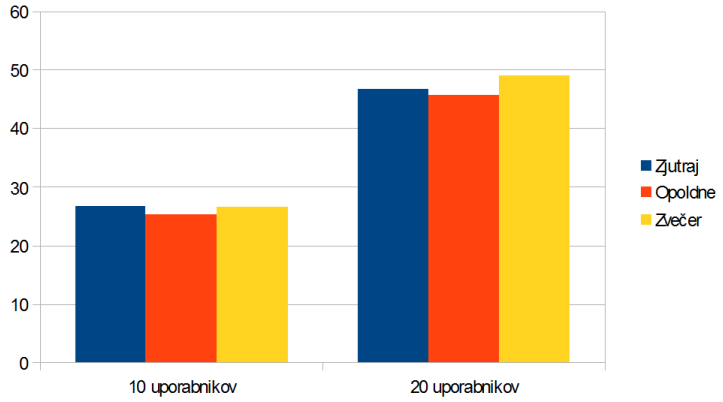
Na sliki 7.4 imamo prikazan graf porabe pomnilnika, ko je bil nastavljen medprihodni čas zahtev 100ms. Na grafu imamo ločen prikaz testa z 10 uporabniki in z 20 uporabniki. Vidimo, da je zasedenost pomnilnika manjša pri testu z 10 uporabniki in sicer za približno 20MB. Opazna je tudi razlika v različnih delih dneva, zjutraj je poraba najmanjša, zvečer pa največja.
Graf, ki prikazuje podatke za nastavitev z medprihodnimi časi zahtev 200ms je prikazan na sliki 7.5. Tudi tukaj vidimo, da je obremenjenjost največja zvečer in najmanjša zjutraj, vendar pa so odstopanja manjša. Manjše so tudi razlike med primerom z 10 uporabniki in primerom z 20 uporabniki, kar je logično zaradi manj pogostega prihajanja zahtev.



Slika 7.6: Graf zasedenosti CPE pri medprihodnih časih zahtev 100ms.

Komentar zasedenosti CPU

Na sliki 7.6 imamo podan graf, ki prikazuje zasedenost CPU pri medprihodnih časih zahtev 100ms. Tudi pri zasedenosti CPU je razvidna dodatna obremenitev, ko imamo 20 istočasnih uporabnikov. Vendar pa so tukaj rezultati veliko bolj nepredvidljivi, saj je procesor uporabljen v gruči in ima tako na naše rezultate vpliv tudi delo ostalih aplikacij. S tem pojasnimo večja nihanja zvečer, ko je bila razlika med primerom z 10 in primerom z 20 uporabniki 20% in pa recimo opoldne, ko je bila zasedenost skoraj enaka pri 50%.
Na sliki 7.7 je prikazan graf z nastavitvijo medprihodnih časov zahtev 200ms. Ponovno opazimo razliko med testom z 10 uporabniki in testom z 20 uporabniki, saj je pri slednjem zasedenost ponovno večja, vendar pa so kot vidimo stolpci zamaknjeni navzdol. Kot lahko razberemo je bila zasedenost pri prvi nastavitvi okoli 25%, pri drugi pa malo nad 45%. Zaradi manj intenzivnega prihajanja zahtev je zasedenost seveda manjša, je pa zato veliko večji razkorak med testom



Slika 7.7: Graf zasedenosti CPE pri medprihodnih časih zahtev 200ms.

z 10 uporabniki in testom z 20 uporabniki, saj je pri drugem skoraj v vseh delih dneva zasedenost 2x večja.

Pojasnila in ugotovitve

Pri testih z zaporedno številko 4,8 in 12 nam je Openshiftova platforma resetirala vse aplikacije, ki oddajajo na izhodne porte (80, 8080 in soležna SSL porta), ker je bilo breme na pomnilnik že očitno preveliko. Težava je, ker delovni pomnilnik wildfly+postavljene aplikacije ni edino kar moramo upoštevati saj zraven teče tudi mysql strežnik. In skupaj sta verjetno pri teh treh poskusih pripeljala porabo delovnega pomnilnika preko 512MB.

Ugotovitve so sledeče:

- *A*: V povprečju je poraba delovnega pomnilnika bila 3,625MB/uporabnika, kar je 0,625MB več kot smo pričakovali, vendar še vedno dovolj blizu, da lahko rečemo, da je prvotni izračun dovolj dober približek,
- *B*: Izkaže se, da imamo na voljo zgolj 482,66MB pomnilnika za našo aplikacijo (wildfly strežnik) in nam ostane le 71,66MB pomnilnika za uporabniške aplikacije. Iz tega lahko sklepamo, da ne moremo postreči 33 uporabnikom hkrati (101MB/3MB), ampak lahko le 19 (71,66MB/3,625MB),
- *C*: Čas med zahtevki precej vpliva na porabo procesorske moči in hkrati tudi na porabo delovnega pomnilnika. Po analizi javanskega procesa, na katerem je tekel Wildfly strežnik, se izkaže, da se porabi veliko časa in prostora za kreiranje in zaključevanje niti (vsak zahtevek teče v svoji niti), kar pa je precej bolj zahtevno, če so časi med zahtevki nizki. V povprečju je bilo pri 200ms času med zahtevki in 10imi sočasnimi uporabniki aktivnih 12 uporabniških niti, pri 100ms pa kar 16. Idealno bi bilo seveda 10

niti, ker imamo 10 sočasnih uporabnikov, vendar strežnik ne uspe zapreti prejšnje niti preden se nova začne, zato imamo overhead niti,

- $\check{C}$: Lokalni čas strežnika vpliva precej na breme na CPU, ker dejansko ne dobimo svojega procesorja ampak dobimo tako imenovani vCore, ki je deljen med vsemi uporabniki gruče. Zaradi tega je precej pomembno, koliko so aktivne ostale aplikacije, ki si delijo isti vCore in v Ameriškem West Coast času je bila zvečer največja poraba procesorja.

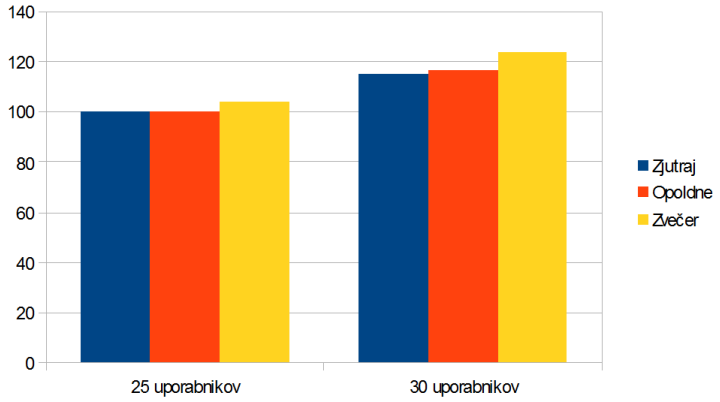
7.6.4 Test zmogljivost CPE in pomnilnika s šifrantom v podatkovni bazi

Za naslednji test smo se odločili, da kategorij in produktov ne bomo prenašali v pomnilnik temveč, da bomo do njih vedno dostopali v bazi. Po novem imamo tako začetno zasedenost pomnilnika 160MB (aplikacija + WildFly server), saj smo z zgornjim ukrepom 251MB sprostili.
Iz prejšnjega benchmarka smo ugotovili, da se zgodi OutOfMemory napaka pri 482,66MB porabe javanskega procesa na katerem teče Wildfly strežnik in naša aplikacija. Pravtako smo izvedeli, da je popvprečna poraba delovnega pomnilnika na uporabnika 3,625MB.

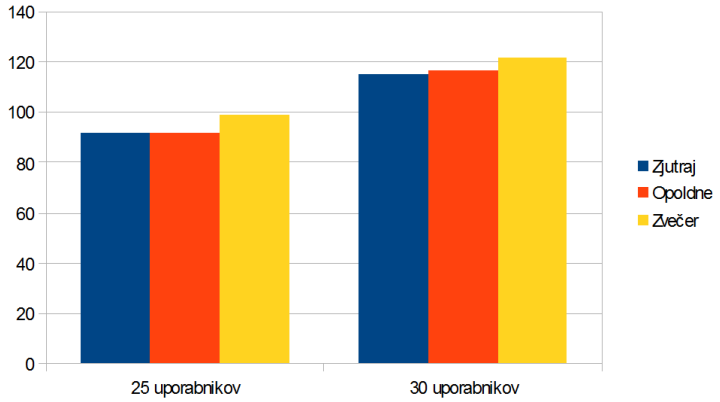
Hipoteza

Iz teh podatkov lahko torej sklepamo da imamo na voljo 322,66MB pomnilnika za uporabnike, sedaj ko nimamo več šifranta shranjenega v pomnilniku. To torej nanese približno 89 sočasnih uporabnikov.
Vemo tudi, da strežnik Wildfly ne zmore postreči več kot 30 zahtevkov/sekundo, zaradi omejitve povezav na bazo v "bazenu"povezav. V prejšnjem benchmarku pa smo se ustavili že pri 19ih sočasnih uporabnikom, torej lahko sklepamo da sedaj, ko ni pomnilnik na aplikaciji več omejen, lahko strežemo 30im sočasnim uporabnikom.

Št. testa	Št. sočasnih uporabnikov	poraba(MB)	zasedenost CPU(%)	skupni delovni pomnilnik (MB)	Čas med zahtevki(ms)	Datum in čas
1	25	92	60	252	200	10. Maj 2015 - 7:11
2	25	100	65	259	100	10. Maj 2015 - 7:23
3	30	115	64	275	200	10. Maj 2015 - 7:31
4	30	115	66	275	100	10. Maj 2015 - 7:44
5	25	92	61	252	200	10. Maj 2015 - 13:01
6	25	100	64	260	100	10. Maj 2015 - 13:33
7	30	117	65	277	200	10. Maj 2015 - 13:41
8	30	117	65	277	100	10. Maj 2015 - 13:51
9	25	99	66	259	200	10. Maj 2015 - 21:03
10	25	104	71	265	100	10. Maj 2015 - 21:14
11	30	122	70	282	200	10. Maj 2015 - 21:32
12	30	124	71	284	100	10. Maj 2015 - 21:41



Slika 7.8: Graf porabe pomnilnika pri medprihodnih časih zahtev 100ms.



Slika 7.9: Graf porabe pomnilnika pri medprihodnih časih zahtev 200ms.

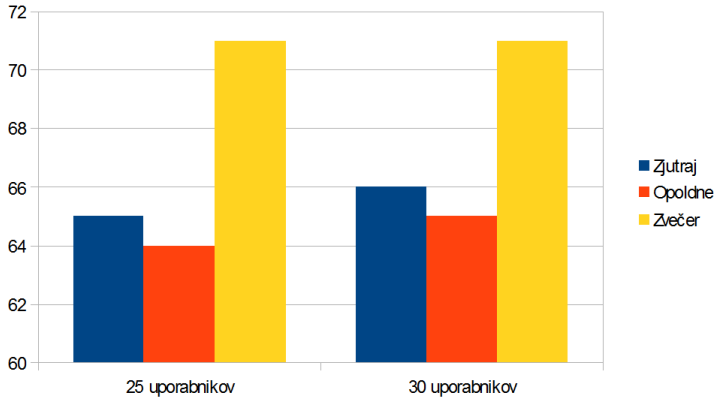
Komentar porabe pomnilnika

Kot lahko takoj razberemo iz slik 7.8 in 7.9 vidimo, da je poraba pomnilnika večja od tiste pri testih s šifrantom v pomnilniku, kar je logično saj tukaj testiramo 25 in 30 uporabnikov.
Na sliki 7.8 imamo graf, kjer je prikazana poraba pomnilnika pri testu z medprihodnimi časi zahtev 100ms. Vidna je razlika v porabi pri testih s 25 uporabniki in testih s 30 uporabniki saj opazimo, da je pri slednjih poraba 15-20% višja. Kot smo opazili že v prejšnjem razdelku pri testih s šifrantom v pomnilniku, je

poraba nekoliko višja zvečer.
Na sliki 7.9 z grafom porabe pomnilnika pri medprihodnih časih zahtev 200ms opazimo skoraj identično sliko, le da so stolpci pri 25 uporabnikih za malo pomaknjeni navzdol. Pomnilnik je zaradi manjše intenzivnosti prihajanja zahtev seveda manj obremenjen, a je razlika precej manjša od tiste v prejšnjem razdelku, ko imamo šifrant v pomnilniku.

Komentar zasedenosti CPE

Na slikah 7.10 in 7.11 imamo grafa, ki prikazujeta zasedenost CPE pri 25 in 30 uporabnikih. Ponovno lahko vidimo razliko v testih opravljenih zvečer ter testih opravljenih zjutraj in opoldne. Medtem, ko se slednja na obeh grafih razlikujeta zgolj za odstotno točko, je zvečer videti povečano zasedenost višjo za do 6 odstotnih točk. Kakor je bilo pojasnjeno že v prejšnjem razdelku je to zaradi delitve procesorja v gruči, saj je največ ostalih aplikacij aktivnih ravno zvečer.

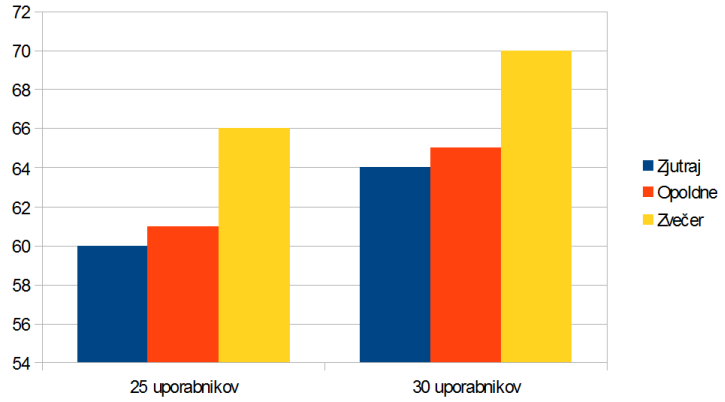


Slika 7.10: Graf zasedenosti CPE pri medprihodnih časih zahtev 100ms.

Ugotovitve

Ugotovitve so sledeče:

- *A*: Zaradi večjega števila povezav na bazo se povprečna poraba delovnega pomnilnika na uporabnika poveča na 3,93MB (prej 3,625MB),
- *B*: Izkaže se, da je sedaj večja obremenitev na CPU, saj je sedaj vedno velika obremenitev (vedno več kot 50%), med tem ko je prej obremenitev le redko prešla 50



Slika 7.11: Graf zasedenosti CPE pri medprihodnih časih zahtev 200ms.

- *C*: Pri testih, kjer je naenkrat nakupovalo 30 uporabnikov, smo dobili vmes kar nekaj napak zaradi pomanjkanja povezav na bazo (iz Wildfly bazena povezav). Po preverjanju logov na strežniku, se je število povezav na bazo povečalo za faktor 5, saj sedaj pri skoraj vsakem zahtevku izvedemo vsaj še en select na bazo (zaradi šifranta, ki ni več v pomnilniku). Izkaže se da 30 uporabnikov na enkrat ni več realna ocena, da bi vse potekalo brez napak. Pri 25ih uporabnikih napak nismo zasledili. Po dodatnem testiranju se izkaže da je 26 uporabnikov naenkrat verjetno najbolj varna ocena da do napak ne bo več prihajalo.

7.7 Zaključek

Če povzamemo vsa testiranja lahko sklepamo:

- da bi vse potekalo brez napak bi morali omejiti (verjetno na load balancerju) število hkratnih povezav na strežnik na 26,
- povprečen dostopni čas zahtevka s povezavo na bazo je 161,73 ms, saj smo odstranili šifrant iz pomnilnika,
- na Openshift cloud platformi je (vsaj za Java EE aplikacije in aplikacijske strežnike) zelo pomembna poraba pomnilnika, saj zelo hitro platforma avtomatično resetira našo aplikacijo, če presežemo limit; glede na porabo procesorja v zadnjem testu se nam ni treba bati, da bi nam resetirali aplikacijo saj je potrebno, da je procesor na 100% porabi vsaj 1 uro, preden resetirajo aplikacijo,
- zastonjska verzija Openshift platforme ni ravno primerna za težko kategorne aplikacijske strežnike kot je Wildfly (ki vsebuje celotno JavaEE

implementacijo), saj zelo hitro napihnejo število niti (omejitev 250 niti) in porabo pomnilnika (omejitev cca. 482,66MB).

Poglavje 8

Analiza zmogljivosti oblačne storitve Openshift

Andrej Premrn, Anej Budihna, Jan Markočič

8.1 Ideja testiranja

Ker se bolj nagibamo k oblačniškim storitvam, smo se odločili bolj podrobneje pregledati benchmarke na CloudHarmony [73]. Opazili smo, da benchmarki pokrivajo tako teste strojne opreme, kot teste omrežja. Benchmarki pokrivajo nekaj najbolj popularnih ponudnikov oblačnih storitev kot so Amazon, Google, Microsoft itd.

Najprej smo pomislili, da bi testirali že obstoječo oblačno storitev. Možnosti je veliko, saj naprimer Google v oblaku gosti 4 miljone aplikacij. Odločili smo se, da bi bilo lažje testirati storitev, ki bi jo sami sprogramirali in skonfigurirali. S tem zagotovimo dobro poznavanje aplikacije in dostop do programske kode. Zasnovali smo štiri ideje, kaj in kako bi lahko testirali:

1. Aplikacija: Client naloži sliko na strežnik, nato sproži prenos iste slike s strežnika [74].

   Kaj testiramo: Public network; hitrost prenosa, hitrost nalaganja.

   Breme: Množica zahtev, istočasnih/zaporednih nalaganj (in prenosov) različno velikih datotek.

   Prednosti:

   - enostavno,

- benchmark teče na naših računalnikih.

Slabosti:

- public network, ne pa private/internal network.

2. Aplikacija: Client pošlje sql poizvedbno na strežnik, ta pa vrne ustrezno sliko/e.

   Kaj testiramo: Hitrost odziva strežnika (hitrost odzivanja na poizvedbe) in download.

   Breme: Množica zahtev, istočasnih/zaporednih povpraševanj in prenosov različno velikih datotek.

   Prednosti:

   - enostavno,
   - benchmark teče na naših računalnikih,
   - testiramo tudi nekako hitrost strežnika (poizvedovanje).

   Slabosti:

   - public network, ne pa private/internal network.

3. Aplikacija: Client naloži sliko, strežnik vrne n podobnih slik.

   Kaj testiramo: Public network; hitrost nalaganja, hitrost prenosa. Hitrost procesiranja slik oz. hitrost iskanja podobnih slik.

   Breme: Množica zahtev, istočasnih/zaporednih uploadov, downloadov (+ obremenitev procesorja, ki mora ugotoviti podobnosti).

   * isto kot prejšnja, bolj zahtevna, bolj izrazito testira cpe, iste tudi pomanjkljivosti

4. Aplikacija: Lokalno testiranje cpeja/pomnilnika na strežniku [75].

   Kaj testiramo: CPE/pomnilnik

   Breme: Množica zahtev za procesiranje ali računanje/prenašanje množice podatkov.

Po pogovoru s profesorjem smo se odločili za 1. možnost, ki bi jo po potrebi še razvili in nadgradili.

8.2 Izbira objektov

8.2.1 Ponudnik

- Openshift [76]: Zaradi že preteklih izkušenj in podpore vsem izbranim tehnologijam smo se odločili za ponudnika Openshift. Openshift je PaaS (Platform as a Service), kar pomeni, da ima predpripravljeno „vso“ infrastrukturo, ki bi jo morebiten uporabnik potreboval. Izbrali smo zastonjsko

(small) verzijo, ki nam ponuja 1 CPU, 512 MB pomnilnika, 1 GB prostora na disku in neomejen prenos podatkov. Če je potrebno lahko našo instanco skaliramo na 1,5 GB pomnilnika in 3 GB prostora na disku.

Ponudnik navaja, da se brezplačna instanca nahaja na vzhodu ZDA. Bolj natančno lokacijo smo pridobili s pomočjo orodja *tracert*. S pomočjo spletne strani, ki določi lokacijo glede na IP naslov smo ugotovili, da se naša instanca nahaja v zvezni državi Virginia. V tracert izpisu se je pojavljala tudi beseda *amazon*. Po nadaljnem raziskovanju smo ugotovili, da Openshift ustvarja instance na Amazonovi oblačni storitvi [77] in tam naloži vso potrebno programsko opremo in opravi konfiguracijo, ki drugače ni ravno enostavna.

8.2.2 Tehnologije

- PHP [78]: Zaradi enostavnosti in dobrega poznavanja smo za oblačno aplikacijo izbrali programski jezik PHP.
- MySQL [79]: MySQL je odprtokodni sistem za upravljanje s podatkovnimi bazami, ki nam omogoča enostavno shranjevanje slik.
- Java [80]: V Javi je izdelana naša namizna aplikacija, ki nam pomaga pri testiranju. Javo smo izbrali, ker je jezik dokaj enostaven in, ker lahko aplikaciji hitro dodamo tudi grafični vmesnik.

8.3 Orodja

8.3.1 Aplikacija na oblaku

Aplikacija pri ponudniku služi kot galerija slik. Preko spletne strani lahko naložimo slike, jih prikažemo v galeriji, pregledamo seznam slik, jih prenesemo itd. Stran na dnu prikaže tudi čas, ki je bil potreben za prikaz slike. Ta čas nam je pomagal pri ugotavljanju pravilnega delovanja namizne aplikacije s katero dopolnjujemo našo oblačno aplikacijo.

Openshift aplikacija je dostopna na `http://php-myspwebsite.rhcloud.com/uploadmultiple.php`.

8.3.2 Javanska namizna aplikacija

Javanska aplikacija nam pomaga in olajša testiranje. Glavna prednost je, da lahko teste popolnoma avtomatiziramo in jih sprogramiramo v aplikacijo. Njena glavna funkcionalnost je, da meri čas (T2), ki je potreben, da nam oblačna storitev pripravi slike za prenos. V principu čas T2 izmerimo po naslednjih korakih:

1. vzpostavimo povezavo s podatkovno bazo,
2. začnemo meriti čas,

3. poženemo poizvedbo,
4. končamo meriti čas pri začetku prenosa,
5. upoštevamo čas zakasnitve.

Rezultate lahko beležimo v tekstovno datoteko, kar nam omogoča hiter in enostaven uvoz v Google Sheets [81], kjer lahko rezultate prikažemo v obliki grafov in krivulj.

Nekaj ostalih funkcionalnosti:

- nalaganje ene ali več slik,
- prenos ene ali več slik,
- prenos slik glede na zaporedno lokacijo (na začetku, na koncu, naključno),
- N-kratna ponovitev prenosa.

Aplikaciji smo izdelali tudi grafični vmesnik, ki je prikazan na sliki 8.1.

8.3.3 Network Monitor (Firefox)

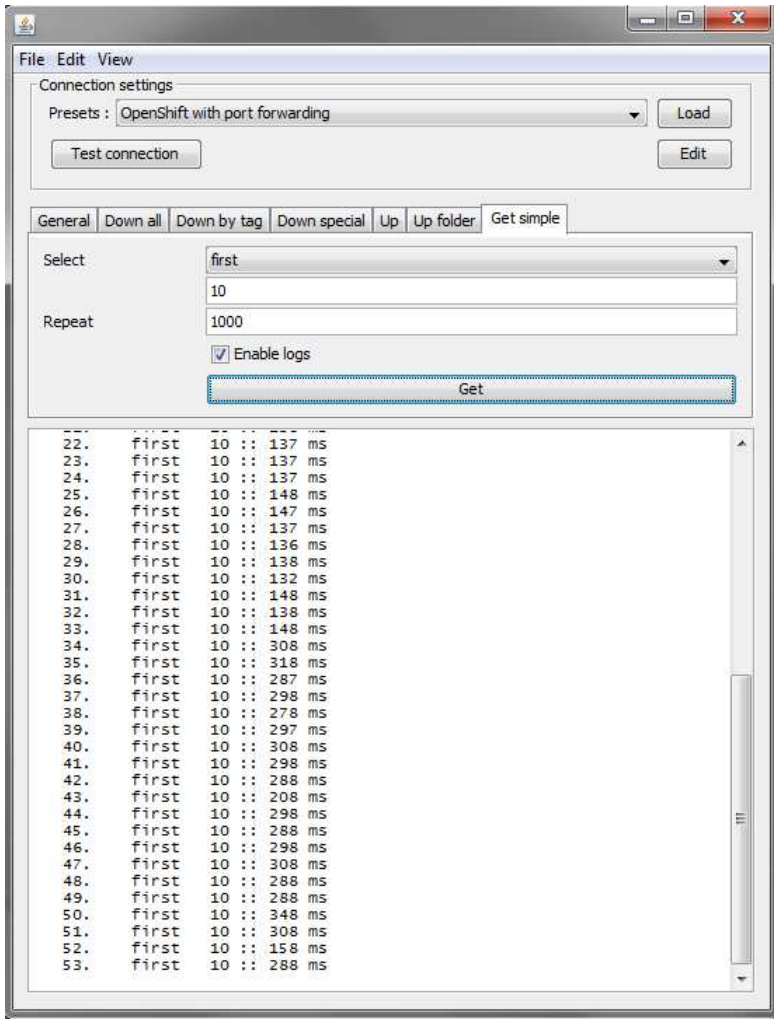
Firefoxov Network Monitor [82] smo uporabili, ker omogoča merjenje različnih časov GET zahtevka (slika 8.2). Za nas je zanimiv čas „waiting“, ki je ravno čas T2. Orodje smo uporabili v navezi z našo spletno aplikacijo, da smo preverili pravilno delovanje naše namizne aplikacije.

8.4 Realizacija eksperimenta

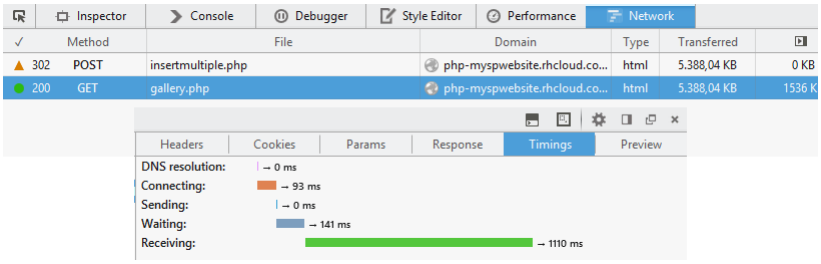
Na Openshiftu smo odprli nov račun. Z vodičem smo si pomagali skonfigurirati platformo in pripraviti vsa potrebna orodja, ki jih potrebujemo za delo s ponudnikom [83]. Dodali in vzpostavili smo MySQL podatkovno bazo v velikosti 1 GB, ki služi za hrambo slik [84]. Na spletu smo poiskali primer PHP programske kode za galerijo slik in jo prilagodili našim potrebam. V naslednjem koraku smo že razmišljali o testiranju. Prva različica našega benchmarka je bila naslednja: Aplikacija na lokalnem računalniku pošlje na strežnik (oblak) množico slik. Tam tamkajšnja storitev poskrbi za prenos določenega števila slik nazaj na lokalni računalnik.

Začeli smo s programiranjem avtomatizacije prenosa slik. Storitev na strežniku smo nadgradili z možnostjo nalaganja množice slik v enem koraku. Ta funkcionalnost bo ključna pri testiranju hirosti nalaganja (T1) in v končni fazi tudi hitrosti prenosa (T3). Zbrali smo tudi slike neba - med seboj podobne slike, ki so pri isti resoluciji približno enake velikosti tudi v bajtih. Ta podobnost (konsistentnost slik) je pomembna pri objektivnem ocenjevanju zmogljivosti oblačne storitve.

Na tej točki smo že dobili nove ideje o aplikaciji in kako bi potekalo testiranje. Nova ideja je bila, da bi z odjemalca začeli z nalaganjem večjega števila slik.



Slika 8.1: Namizna javanska aplikacija.



Slika 8.2: Firefoxov Network Monitor. Vsota vseh časov je slabih 1,4 s.

Ob začetku nalaganja bi začeli meriti čas. Ob končanem nalganju slik na oblak, bi uporabnika preusmerili na galerijo slik. Čas prikaza slike v galeriji je enak času, ki bi ga potrebovali za dejanski prenos slike s strežnika na računalnik. Ob končanem prikazu vseh slik v galeriji bi ustavili merjenje časa. Iz tega podatka bi nato lahko dobili T2. T2 je čas, ki preteče od trenutka, ko naša oblačna aplikacija dobi zahtevo za akcijo, do trenutka, ko se sproži prenašanje slik. Torej T2 je čas, ki ga storitev porabi za pripravo odgovora na našo zahtevo.

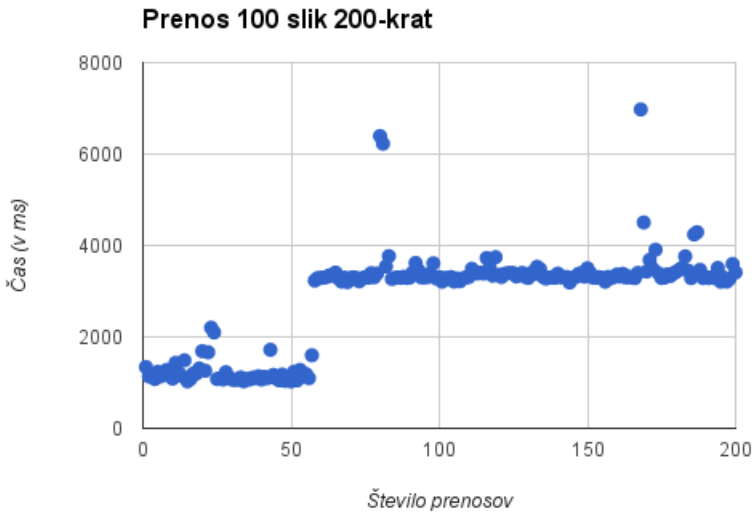
Po še nadaljnem razmišljanju smo se odločili, da bi merjenje časa in prenos datotek lažje nadzorovali in upravljali z namizno aplikacijo, ki bi jo sprogramirali v Javi. Naslednji korak je bil izdelava javanske aplikacije.

Po izdelani namizni aplikaciji je prišel čas, da smo začeli razmišljati o testiranju oblačne storitve. Na oblak smo naložili 8 slik. Merili smo čas, ki ga je aplikacija potrebovala, da je uspešno prenesla te slike na lokalni računalnik. V našem primeru je ta čas znašal okrog 2,4 s. Isto meritev smo naredili tudi v brskalniku z izborom zavihka „gallery“, saj se v tem primeru prav tako izvede prenos vseh slik v polni velikosti. Na dnu strani smo prikazali čas, ki ga je strežnik potreboval od pridobitve zahtevka do končane priprave podatkov. Ta čas je znašal zelo blizu 1 s. Z orodjem brskalnika Firefox (Network Monitor), ki omogoča vpogled v čase, ki so bili potrebni za izvedbo določenih korakov, smo ugotovili, da prenos omenjenih slik na lokalni računalnik traja približno 1,4 s (slika 8.2). Čas, ki smo ga pridobili z aplikacijo (2,4 s) smo si razlagali kot:

- čas, ki ga je izmeril strežnik za pripravo podatkov (slik): 1 s,
- čas, ki je bil potreben za prenos slik na lokalni računalnik: 1,4 s.

Med testiranjem smo opazili, da ima naša namizna aplikacija določene pomanjkljivosti. Namreč, ko program prenese slike s strežnika, te shrani na disk. Naša štoparica upošteva tudi omenjeno akcijo, kar je narobe, saj s tem naš benchmark zajema tudi hitrost lokalnega diska.

Ponovno smo pregledali izvorno kodo namizne aplikacije. Popravili smo način merjenja. Slik ne shranjujemo več na disk in tako tudi ne upoštevamo več časa, ki je potreben za zapis slik. Čas, ki ga izmeri aplikacija, smo izolirali na čas T2. Aplikaciji smo dodali še dodatno funkcionalnost, da lahko izbiramo ali hočemo prenesti slike z zaporednih pomnilniških lokacijah ali z naključnih. Ker je za pridobitev verodostojnih rezultatov potreben dober vzorec množice



Slika 8.3: Graf priprave 200-kratnega prenosa 100 slik.

meritev, smo dodali tudi možnost N-kratnega zaporednega avtomatskega ponavljanja akcije.

Na tej točki smo imeli vse pripravljeno za postavljanje hipotez in izvajanje testov.

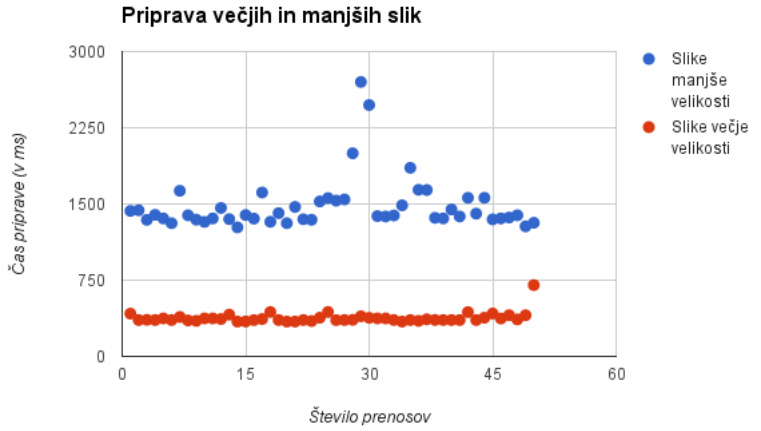
8.5 Testiranje, meritve in analiza

Testiranje poteka po principu, da najprej postavimo neko hipotezo, nato opravimo eksperiment oziroma meritve, dobljene podatke analiziramo in na koncu potrdimo ali ovržemo hipotezo.

1. Hipoteza: Sistem oblačne storitve bo prepoznal zahteve po isti sliki in jo naredil hitreje dostopno, kar se pokaže po hitrejšem času priprave.

   Test: Z zaporednim povpraševanjem 200-krat zahtevamo isto sliko. Povpraševanja so zaporedna, brez vmesnega čakanja.

   Ugotovitve: Pri prenosu 1, 2 in 10 slik nismo opazili spremembe časa (tudi pri 3000 zaporednih prenosih). Pri prenosu 50 in 100 slik, se je čas priprave pri določenem zaporednem prenosu podvojil. Podvojitev se je pri 50 slikah pojavila pozneje kot pri 100 slikah. Slika 8.3 prikazuje čase odziva oblačne storitve v odvisnosti od števila prenosov pri prenosu 100 slik. Hipotezo smo zavrnili, saj čas priprave ostane relativno enak ali pa se celo podvoji.



Slika 8.4: Graf primerjave priprave slik večje in manjše velikosti.

2. Hipoteza: Čas prenosa slike bo večji, če prenesemo večje število manjših slik kot če prenesemo majhno število velikih slik.

   Test: Najprej prenesemo 97 datotek velikosti okoli 415 kB, katerih skupna velikost znaša 39,1 MB. Nato pa prenesemo še 23 datotek povprečne velikosti 1755 kB, katerih skupna velikost je skoraj ista kot velikost prejšnje skupine slik (39,3 MB). Test ponovimo 3-krat, brez vmesnega čakanja.

   Ugotovitve: Trikratno prenašanje večjega števila majhnih slik je trajalo 3 minute in 22 sekund, kar je praktično enako časa kot trikratno prenašanje manjšega števila večjih slik, ki je trajalo 3 minute 20 sekund. Torej hipotezo ovržemo. Kot kaže, priprava slik traja zanemarljivo malo časa, saj priprava velike množice slik ni naredila razlike v času v primerjavi z pripravo manjše množice slik. Ta ugotovitev nas pripelje do naslednjega testa. Bolj natančna meritev samega časa priprave slik različnih velikosti.

3. Hipoteza: Čas priprave slike bo večji, če zahtevamo večje število manjših slik kot če zahtevamo majhno število velikih slik.

   Test: Najprej od serverja zahtevamo, da nam pripravi 97 datotek velikosti okoli 415 kB, katerih skupna velikost znaša 39,1 MB. Nato pa zahtevamo še pripravo 23 datotek povprečne velikosti 1755 kB, katerih skupna velikost je skoraj ista kot velikost prejšnje skupine slik (39,3 MB). Test ponovimo 50-krat, brez vmesnega čakanja.

   Ugotovitve: Priprava 96 majhnih slik (velikosti 415 kB) je trajala v povprečju 1484 ms, priprava 23 velikih slik (velikosti 1755 kB) je pa trajala v povprečju 378 ms. Hipotezo potrdimo, saj je razlika v času zelo očitna. Očitno je pri prejšnjem testu čas priprave slik zanemarljivo majhen in zato neopazen pri rezultatih. Slika 8.4 priazuje pridobljene rezultate.

Slika 8.5: Graf primerjave povprečnega časa priprave slik na naključnih in zaporednih lokacijah.

4. Hipoteza: Čas priprave slike je neodvisen od njene velikosti.

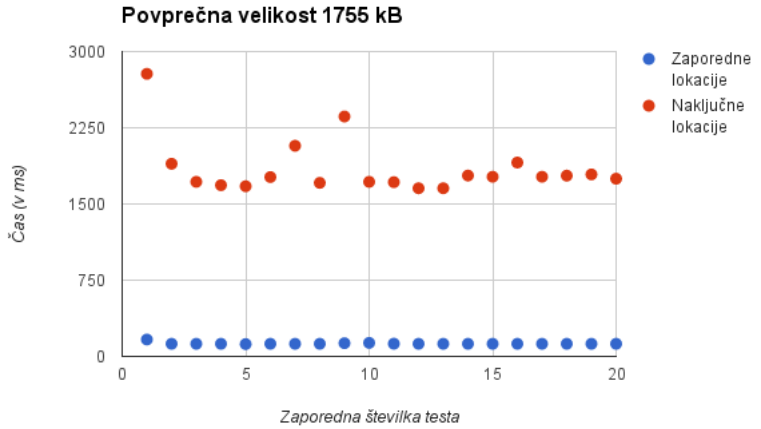
   Test: Od oblaka zahtevamo, da pripravi 1, 50, 100 naključnih slik (niso vedno iste). Test ponovimo za skupine slik, katerih povprečna velikost je 13 kB, 415 kB, 1100 kB in 1755 kB. Kombinacije testov ponovimo 20-krat.

   Ugotovitve: Pri vseh kombinacijah testov so bili rezultati precej podobni. Priprava slik, ne glede na njihovo število in velikost, vedno traja med 1,8 s in 2,7 s. Rezultati testov so bili precej nekonsistentni. Razlog je po vsej verjetnosti mehanizem oblaka, ki upočasni delovanje storitev ob previsokem številu dostopov. Za boljšo verodostojnost testov in rezultatov, smo med posameznimi testi počakali nekaj minut, da bi minimizirali verjetnost, da se bo sprožil omenjen mehanizem upočasnjevanja.

5. Hipoteza: Čas priprave naključne slike bo daljši kot v primeru, ko zahtevamo vedno isto sliko.

   Test: Od oblaka zahtevamo, da pripravi prvih 5 in 5 naključnih slik. Test ponovimo za skupine slik, katerih povprečna velikost je 13 kB, 415 kB, 1100 kB in 1755 kB. Kombinacije testov ponovimo 20-krat.

   Ugotovitve: Povprečni čas priprave naključnih slik je v vseh primerih daljši kot povprečni čas priprave vedno istih slik. Hipotezo torej potrdimo. Razlike v časih so bile največje pri majhnih slikah (slika 8.5) (povprečna velikost 13 kB) in najmanjše pri velikih slikah (slika 8.6) (povprečna velikost 1755 kB).



Slika 8.6: Graf primerjave povprečnega časa priprave slik na naključnih in zaporednih lokacijah.

8.6 Zaključek

Postavili smo aplikacijo, ki temelji na prenosih slik in dobro simulira delovanje povprečne moderne aplikacije.

Kljub temu, da smo uporabljali brezplačno storitev, smo ob normalni uporabi ugotovili, da je zmogljivost oblačne storitve precej dobra razen ob velikih obremenitvah, kjer je prišlo do precejšnje omejitve hitrosti. Zanimivo bi bilo iste teste ponoviti tudi na plačljivi instanci ali oblaku in preveriti, če se omejitve pojavijo kasneje, oziroma če sploh se.

Večina rezultatov, ki smo jih pridobili so bili pričakovani. Ugotovili smo, da se pri prenosu slik večji del časa porabi za iskanje in pripravo slik in da je dejansko prenašanje podatkov hitro. Od vseh rezultatov nas je najbolj presenetila ugotovitev o pomankanju predpomnjenja podatkov, v primeru, ko se pogosto dostopa do istih podatkov. Predpomnenje bi zelo pohitrilo pripravo podatkov.

Poglavje 9

Raziskovalno testiranje oblačne storitve

Urh Svetičič, Matevž Žugelj, Nejc Zupan

9.1 Uvod

Oblačne storitve ne prinašajo samo novih poslovnih priložnosti, temveč imajo tudi velik vpliv na testiranje in vzdrževanje programske opreme. Vpliv se pokaže, ko se pogovarjamo o testiranju programske in strojne opreme v oblaku kot storitev (Testing as a Service - TaaS).
Ko govorimo o TaaS je potrebno poznati odgovore na sledeča vprašanja [85]:

- Kaj je oblačno testiranje?
- Kakšne tipe testiranj je potrebno izvesti za oblačne storitve?
- Kakšne so glavne razlike med običajnim testiranjem programske opreme in testiranjem programske opreme v oblaku?
- Katere so posebne zahteve in lastnosti pri testiranju oblačnih storitev?
- Kakšni so problemi, izzivi in potrebe pri oblačnem testiranju?

9.1.1 Razumevanje testiranja oblačnih storitev

Oblak nam ponuja stroškovno učinkovit in fleksibilen način skozi katerega se skalabilni računalniški resursi ponudijo kot storitve uporabnikom, kjerkoli in kadarkoli to želijo.

Kaj je oblačno testiranje?

Na kratko povedano, oblačno testiranje programske opreme pomeni merjenje nekih aktivnosti na dani infrastrukturi (oblaku) z uporabo oblačnih tehnologij in programskih rešitev. Ima 4 večje cilje:

- zagotoviti kvaliteto aplikacij v oblaku;
- validirati storitev kot SaaS (Software as a Service), vključno z zmogljivostjo, skalabilnostjo in varnostjo;
- preveriti že obstoječe funkcijske, podporne oblačne storitve, s katerimi upravljamo resurse;
- testirati kompatibilnost oblaka in medsebojno komunikacijo med SaaS in aplikacijami v oblaku, na primer preverjanje če API-ji naših SaaS lahko med seboj komunicirajo.

Oblike testiranja programske opreme v oblaku

Poznamo 4 različne oblike testiranja programske opreme v oblaku:

- **Testiranje SaaS v oblaku**. Zagotovimo kvaliteto samega SaaS, preverimo njegovo funkcionalnost, da je skladna z zahtevami.
- **Zunanje testiranje oblaka**. Glede na specificirane zmogljivosti in funkcionalnosti ovrednotimo stanje oblaka.
- **Notranje testiranje oblaka**. Zanimajo nas povezave med različnimi SaaS v oblaku, nadzor storitev, kapacitet, upravljanje z oblakom.
- **Testiranje med oblaki**. Testiramo aplikacijo v različnih oblakih.

9.1.2 Kaj je novega pri oblačnem testiranju?

V nasprotju s konvencionalnim testiranjem spletne programske opreme ima oblačno testiranje nekaj edinstvenih ciljev, zahtev in značilnosti.

Zahteve in značilnosti pri oblačnem testiranju

Pri oblačnem testiranju, se v primerjavi s konvencionalnim pojavijo 4 nove zahteve in značilnosti:

- Za testiranje potrebujemo oblačno testno okolje, ki mora nuditi vse potrebne vire.

- Ker odjemalcem nudimo različne storitve, jim moramo zagotoviti kvaliteto le teh. To dosežemo z dogovorom (Service Level Agreement - SLA). V dogovorih so definirane naslednje količine: kvaliteta, dostopnost, zanesljivost, varnost in zmogljivost, ki jih moramo zagotoviti.
- Ker ponujamo storitve nekim kupcem, moramo imeti točno določene cenovne modele za storitve, ki jih nudimo. Vedeti moramo, kaj bomo uporabniku zaračunali.
- Naša storitev mora omogočati simulacijo podatkov in prometa, kakor bi to pričakovali v realnem svetu (gledano po količini).

Testiranje kot storitev (Testing as a Service - TaaS)

TaaS predstavlja inovativen koncept, saj gre za statično-dinamično testiranje na zahtevo ("on demand"). Ponuja vse zgoraj naštete vrste testiranja oblaka. Glavna prednost takih storitev je zmanjševanje stroškov podjetij pri testiranju SaaS storitev.

Koncept TaaS je bil prvič predstavljen leta 2009. Od takrat je bil deležen veliko pozornosti zaradi njegovih posebnih značilnosti. TaaS-ov miselni vzorec ("work flow") sestavljajo večji deli:

- upravitelj TaaS procesov, ki ponuja upravljanje testiranja in kontrolo procesov, je neposredno povezan z ostalimi komponentami;
- upravitelj zahtev, ki jim mora storitev zadostovati (Quality of Service requirements management);
- testno okolje oziroma storitev, ki nam je na voljo zmeraj, ko jo potrebujemo;
- nabor in izbor vseh možnih vrst testiranja;
- simulacijsko okolje, ki na zahtevo stori vse potrebno za izvedbo testov;
- storitev, ki na zahtevo izvede uporabniško definiran test;
- sledenje in nadzor testiranju, ki nam nudi pregled rezultatov;
- prodajo TaaS storitev.

9.1.3 Priložnosti, problemi, izivi in potrebe

Da bi karseda učinkovito izvajali oblačno testiranje, je zelo pomembno, da poznamo primarne probleme, potencialne izzive in potrebe:

- Do težav pride pri *postavljanju testnih okolij na zahtevo.*
- *Problem testiranja skalabilnosti in zmogljivosti.* Testiranje skalabilnosti poteka preko vnaprej nastavljenih porazdeljenih sistemov. Zato je testiranje novih funkcionalnosti, kot je dinamična skalabilnost, vprašljivo.

- *Testiranje varnosti v oblaku.* Varnost je že tako vroča tema v računalništvu, ko pa se preselimo v oblak pa še toliko bolj. V oblaku se porajajo slednja vprašanja: Kako lahko zagotovimo varnost naše oblačne storitve v nekem tujem oblaku? Ali obstajajo kakšni standardi? Kako testirati varnost? Kako zagotoviti omejitev polnoletnosti pri uporabi oblačnih storitev?

- *Problem integracijskih testov v oblakih.* V oblaku se inženirji soočajo z integracijo različnih SaaS. Komunikacijski protokoli so različni, zato je zelo veliko možnosti povezav. Testiranje je zato zelo oteženo.

- *Regresijsko testiranje* se uporablja za iskanje napak v kodi oziroma odkrivanje hroščev. Temelji na večkratnem testiranju enake kode s prednastavljenimi parametri. To zna biti problem pri oblačnem testiranju na zahtevo, kjer želimo testirati kodo takoj, ko jo spremenimo in nimamo časa ročno vnašati parametrov za testiranje te vrste.

9.1.4 Indentifikacija ciljnega sistema

Preden se lotimo testiranja, moramo izbrati opazovani sisem. Naš ciljni sistem v okviru našega raziskovanja predstavlja oblačno storitev. Predvidevamo, da bodo meritve bolj nepredvidljive in bomo s tem prišli do zanimivejših rezultatov. Prav tako je to za nas neznano področje in nam predstavlja izziv. Preko spletne strani http://cloudharmony.com/benchmarks, smo prišli do naslednjih večjih ponudnikov oblačnih storitev:

- **Amazon EC2** - (http://aws.amazon.com/ec2/): Pri Amazonu je mogoče brezplačno najeti t2.micro. V to ponudbo je mesečno vključenih 750 procesorskih ur na procesorju Intel Xeon E5-2670 v2 2.50GHz z 1GB RAM. Na oblaku lahko poganjamo operacijski sistem Linux ali Windows.

- **Rackspace Cloud Servers** - (http://www.rackspace.com/cloud/servers/): Ne ponujajo brezplačnih storitev.

- **Microsoft Azure** - (http://azure.microsoft.com/en-us/): Microsoft nam ponuja začetnih 200$ kreditov ki jih nato lahko uporabljamo. Najcenejša instanca z imenom A0 vsebuje 1 jedro, 0,75 GB RAM in 20 GB trdega diska. Obračunski interval za A0 je 0.018 $ na uro.

- **Google Compute Engine** - (https://cloud.google.com/compute/): Google nam ponuja 300$ kreditov, ki pa jih je potrebno porabiti v roku 60 dni.

Izmed zgoraj navedenih ponudnikov oblačnih storitev je najugodnejši Amazon EC2. Ta nam namreč mesečno ponuja 750 zastonjskih delovnih ur, kar bi moralo zadostovati za naše testiranje. Odločili smo se, da bo to naš ciljni sistem.

9.1.5 Teorija metrik

Sedaj, ko smo si izbrali ponudnika oblačnih storitev, je čas da si izberemo metrike testiranja. Na povezavi [86] iz spletne učilnice najdemo trenutno vse aktualne metrike. Zdi se nam pomembno, da bi na našem opazovanem sistemu preverili večino omenjenih metrik. Metrike, ki bi jih v nadaljevanju testirali, so:

- **Zmogljiost CPU**. Pri tem je potrebno preverjati zmogljivost posameznega in večih jeder hkrati.
- **Hitrost branja in pisanja v pomnilnik**.
- **Hitrost branja in pisanja na trdegi disk**.
- **Zmogljivost prenosa podatkov čez omrežje**.

Raziskali smo tudi področje testiranja oblačnih storitev preko orodja Web Framework Benchmarks [87], saj smo želeli na čim bolj realen način testirati zmogljivost sistema. To pomeni obremeniti sistem na način, ki je podoben veliki količini prometa v realnosti. S tem bi testirali zmogljivost CPE, zmogljivost pomnilniških medijev (RAM, HDD) in generirali omrežno breme. Omenjene metrike testi testirajo z JSON serializacijo, različnimi poizvedbami po podatkovni bazi in urejanjem podatkov.

Izkazalo se je, da testi ne zadostujejo zastavljenim zahtevam, saj so preveč splošni. Odločili smo se za bolj algoritmični pristop, ki je opisan v nadaljevanju, ker nam omogoča usmerjeno testiranje zmogljivosti naštetih komponent.

Na spletu nam je uspelo najti še dve spletni strani [88][89], ki ravno tako testirata zmogljivost oblačnih storitev. V obeh se uporabljajo več ali manj enake metrike kot jih nameravamo mi.

9.1.6 Teorija bremen

Za zgoraj izbrane metrike moramo definirati še breme, s katerim jih bomo testirali. Za generiranje bremena bomo uporabljali različne benchmarke. Zaradi lažjega primerjanja rezultatov, bomo verjetno uporabili iste benchmarke, kot so bili uporabljeni na spletni strani cloudharmony [86]. Poskusili bomo tudi uporabiti benchmarke, ki so navedeni v zgornjih dveh povezavah. Za benchmarke v primerjavi z realnim bremenom, smo se odločili zaradi zmožnosti ponovnega generiranja enakega bremena, zato so veliko boljši od real-time testiranja. Kot vemo realno breme ni ponovljivo in s tem neuporabno za testiranje.

Poznamo dva načina testiranja z benchmarki. Prvo je t.i. **pasivno testiranje** [88], kar pomeni, da se benchmark zažene in se ga pusti teči dokler se ne zaključi. Po zaključku se iz benchmarka pobere podatke, ki se jih analizira. Drugi način testiranja je **aktivno testiranje** [88], ki je v osnovi enako kot pasivno, le da se v tem primeru med samim izvajanjem benchmarka opazuje celoten sistem oziroma del, ki se ga testira. Takšno testiranje je veliko boljše od pasivnega, saj lahko med samim testiranjem opazujemo stanje našega sistema.

Če sistem ni v celoti izkoriščen, moramo preveriti ali je napaka v benchmarku ali pa gre za ozko grlo v sistemu.

Benchmarke delimo na dva načina. Prvi način jih deli glede na obseg testiranja. Lahko gre za testiranje posameznih delov sistema ali pa kar za celoten sistem. Druga delitev deli benchmarke glede načina testiranja. Bodisi imamo več različnih testov za določeno metriko bodisi imamo enega.

Delitev po obsegu testiranja

Delitev po obsegu testiranja vsebuje dve možni vrsti benchmarkov. Prvi so t.i. Component-level Benchmarks. Ti generirajo breme, ki obremeni samo želen del strojne opreme, medtem ko je druga vrsta benchmarkov (System-level Benchmarks) namenjena obremenjevanju celotnega sistema.

Delitev po načinu testiranja

Druga delitev je po lastnostih bremena. Tudi tu imamo dve vrsti. Prvi so "Synthetic Benchmarks", ti generirajo breme, ki naj bi simuliralo realno breme. V praksi to pomeni, da je breme sestavljeno iz večih različnih segmentov. Druga vrsta oziroma "Application Benchmarks" pa generirajo samo en segment bremena. Benchmarki te vrste so večinoma "System-level Benchmarks" in testirajo vpliv posamezne komponente na celotno zmogljivost sistema. Ponavadi so veliki in zahtevni za izvršitev.

9.2 Naš poligon

9.2.1 Izbira in določitev metrik in bremen

Centralno procesna enota (CPE)

Pri testiranju CPE se moramo zavedati, da je potrebno vse teste izvesti vsaj dvakrat. Najprej testiramo vse teste na eni niti, nato pa še na vseh razpoložljivih. Teste izvajamo nad naključnimi podatki na disku.

Oznaka GB/sec pri integer operacijah pomeni uporabljeno pasovno širino do glavnega pomnilnika. Za teoretično mejo bi morali pogledati v specifikacijo opazovanega sistema.

Možni testi so:
Zmogljivost **integer operacij** testiramo z zgoščevalnimi algoritmi:

- AES [GB/sec],
- Twofish [GB/sec],
- SHA1 [GB/sec],
- SHA2 [GB/sec].

Zanima nas tudi zmogljivost kompresijskih in dekompresijskih algoritmov:

- BZip2 [MB/sec],
- JPEG [Mpixels/sec],
- PNG [Mpixels/sec],
- Sobel [Mpixels/sec],
- Lua [MB/sec],
- Dijkstra [Mpairs/sec].

Poznamo tudi metriko **Dhrystone** [90]. Ta ima oznako VAX MIPS. VAX predstavlja arhitekturo zbirke ukazov (instructions set architecture), MIPS (Millions Operations per Second) pa mersko enoto milijon operacij na sekundo.

Zmogljivost operacij **plavajoče vejice** lahko preverjamo s sledečimi algoritmi:

- BlackScholes [Mnodes/sec] - predikcija cen delnic;
- Mandelbrot (set) [Gflops] - računanje fraktalov v kompleksni množici;
- Sharpen Filter [Gflops] - izostritev slik;
- Blur Filter [Gflops] - megljenje slik;
- SGEMM (Single-Precision General Matrix Multiplication) [Gflops] - množenje matrik v plavajoči vejici z enojno natančnostjo;
- DGEMM (Double-Precision General Matrix Multiplication) [Gflops] - množenje matrik v plavajoči vejici z dvojno natančnostjo;
- SFFT (Sparse Fast Fourier Transform) [Gflops] - transformacija analognih signalov;
- DFFT (Discrete Fast Fourier Transform) [Gflops] - transformacija analognih signalov;
- N-Body [Mpairs/sec] - Simulacija N teles, na katere vplivajo fizične sile (ponavadi gravitacija);
- Ray Trace (Graphics) [Mpixels/sec] - tehnika za sledenje poti svetlobe za doseganje virtuale realnosti.

Zmogljivost operacij plavajoče vejice lahko ocenimo tudi z metodo **Whetstone** [91]. Whetstone je interpreter, ki je modificiran in uporabljen za zbiranje statistike o opravljenih Whetstone operacijah. Osnovna metrika je predstavljena z MWIPS (Millions of Whetstone Instructions Per Second). Gre za aritmetiko s plavajočo vejico z enojno natačnostjo.

Za testiranje zmogljivosti CPE bomo uporabili **Geekbench benchmark**. Nudi nam vse zgoraj naštete vrste testiranja, razen načinov Dhrystone in Whetstone.

Pomnilnik

Testiranje pomnilnika zna biti težavno. Testi morajo biti načrtovani tako, da smo lahko prepričani, da vsakič dostopamo do pomnilnika. Če tega ne zagotovimo, potem se lahko zgodi dostop do predpomnilnika. Vse te zahteve pokriva orodje Geekbench, ki nam ponuja ocenitev hitrosti pomnilnika po metodi STREAM [92], ki je sestavljena iz 4 testov:

- COPY - a(i) = b(i),
- SCALE - a(i) = q*b(i),
- SUM - a(i) = b(i) + c(i),
- TRIAD - a(i) = b(i) + q*c(i).

Orodje Geekbench deluje na dovolj veliki zbirki podatkov, da pokrije zgoraj omenjene zahteve.

Trdi disk

Za testiranje trdega diska smo si izbrali zastonjsko orodje Fio (flexible I/O tester). Fio v osnovi podpira 19 različnih tipov I/O gonilnikov, prioritet, zagona večih niti. Deluje tako na bločnih napravah, kot na datotekah in ponuja dostop preko omrežja. Je orodje, ki se ga množično uporablja, saj omogoča natančno definicijo bremen in je narejen praktično za vse večje platforme kot so Linux, OS X, Windows, Android in HP-UX [93].

Fio nam omogoča veliko različnih konfiguracij. Na začetku imamo najbolj osnovne, ki služijo opisu testa in ne vplivajo na samo izvajanje:

- **name** - ime posla za lažje ločevanje, ko imamo več poslov,
- **description** - opis posla, ki se izpiše ob začetku izvajanja testa,
- **directory** - ime direktorija, v katerem so bodo nahajale datoteke za pisanje in branje,
- **filenamen** - ime datoteke, ki jo program ustvari za potrebe delovanja,
- **exclusive** - ekskuzivno oziroma sinhrono dostopanje niti do IO virov.

Po osnovni nastavitvi sledi izbira opcij testiranja. Pri tem lahko izbiramo med pisanjem, branjem in kombinacijo obeh skupaj. Potrebno je tudi upoštevati, da se lahko več branj izvaja hkrati, medtem ko za pisanje potrebujemo ekskluzivno pravico. Za določitev izbere testa imamo na voljo naslednje prametre:

- **read** - sekvenčna branja,
- **write** - sekvenčna pisanja,

- **randread** - naključna branja,
- **randwrite** - naključna pisanja,
- **rw** - mešana sekvenčna branja in pisanja,
- **randrw** - mešana naključna branja in pisanja.

Pri mešanem branju in pisanju je s parametroma rwmiread in rwmixwrite mogoče nastaviti procent branj in pisanj. S parametri random_distribution, random, zipf, pareto, randrepeat in use_os_rand določino, po kakšnem ključu se bodo generirala naključna števila za random branja in pisanja.

Sledijo parametri, ki določajo, kako veliko bo naše breme:

- **size** - Program izvaja IO operacije dokler ne prenese določene količine podatkov.
- **fill_device** - Določi velikost na zelo visoko vrednost in izvaja pisalne operacije dokler ne zmanjka pomnilnika na napravi. Parameter je smiseln le, če uporabljamo sekvenčno pisanje.
- **filesize** - Maksimalna velikost datoteke, ki jo lahko fio ustvari.
- **blocksize** - Velikost enega bloka za IO operacije. Nastavi se ločeno za pisanje in branje. Če parametra ne navedemo je parameter privzeto nastavljen na 4 KB.
- **blocksize_range** - Ta parameter je podoben zgornjemu, saj izberemo razpon, znotraj katerega se naključno določajo velikost blokov za IO operacije.
- **bssplit** - Omogoča nam, da določimo koliko procentov blokov bo kakšne velikosti.
- **blockalign** - Na kakšno velikost so poravnani bloki zapisani na disku.
- **nrfiles** - Število datotek, ki jih program uporablja za to nalogo.
- **random** - Izbere datoteko naključno.

Sedaj izberemo ioengine oziroma način izdaje ukaza posla IO napravi. Fio nam omogoča 19 načinov izdaje ukazov. Omenili bomo le najbolj uporabljene. Ti se delijo na sinhrone in asinhrone. Med sinhrone spadajo sync, psync in vsync, med asinhrone pa libaio, posixaio, solarisaio, windowsaio in mmap.

še ostale nastavitve Fia:

- **prio** - Stopnja prioritete, ki jo dodelimo poslu. Prioriteta mora biti na intervalu med 0 in 7.
- **thinktime** - Čas čakanja v milisekundah pred izdajo IO operacije.
- **rate** - Nastavitev zgornje meje pasovne širine.

- **ratemin** - Minimalna pasovna širina, ki jo mora test doseči, če ne se zaključi.
- **rate_iops, rate_iops_min** - Enako kot pri zgornjih dveh, le da tu omejujemo IOPS.
- **max_latency** - Maksimalna latenca, ki jo dovolimo. **č**e je latenca večja, se test zaključi.
- **runtime** - Maksimalni čas izvajanja testa.
- **loops** - Število ponovitev istega posla.
- **clocksource** - Nastavitev osnove za merjenje ure. Potrebno je, da je ura zelo natančna, če želimo meriti latenco.

Po končani konfiguraciji bremena zaženemo program ki nam vrne sledeče rezultate.

- **io** - število megabajtov, ki jih je proces izvedel;
- **bw** - povprečna pasovna širina;
- **runt** - čas izvajanja testa;
- **slat** - minimalna, maksimalna, povprečna in standardna deviacija latence; ta latenca predstavlja čas od nastanka zahtevka pa do njegove oddaje I/O napravi;
- **clat** - minimalna, maksimalna, povprečna in standardna deviacija latence; s to latenco merimo čas med oddajo in zaključitvijo zahteve;
- **bw** - minimalna, maksimalna in povprečna pasovna širina ter procent združene pasovne širine;
- **cpu** - statistika uporabe procesorja;
- **io depths** - porazdelitev I/O globin;
- **io issued** - število izdanih zahtev za branje/pisanje in število zahtev za kratka branja/pisanja;
- **io latencies** - porazdelitev I/O clat latenc; ta porazdelitev je podobna porazdelitvi za globine;
- **ios** - število io ki so jih naredile vse grupe;
- **merge** - število združenj znotraj I/O razporejevalnika;
- **ticks** - število tikov potrebnih za ohranjanje delovanja diska;
- **io_queue** - čas porabljen znotraj diskovne vrste;
- **util** - merjene obremenitve diska;

Omrežna povezava

Pri testiranju omrežne povezave nas zanimajo naslednje metrike:

- Throughput [kb/s],
- Delay jitter,
- Packet loss.

Za testiranje zmogljivosti povezave smo se odločili za uporabo programske opreme Iperf [94]. Program je napisan v programskem jeziku C in je namenjen merjenju zmogljivosti povezave med dvema točkama. Za njegovo uporabo ga je potrebno naložiti na obe končni točki, v našem primeru sta to oblak in računalnik, preko katerega dostopamo do oblačnih storitev. Iperf deluje po principu server-client, zato je potrebno, da ga namestimo na obe končni točki. V našem primeru bo imel naš računalnik vlogo klienta, oblak pa vlogo strežnika.

Program glede na podane parametre vzpostavi UDP ali TCP povezavo med klientom in strežnikom. V primeru uporabe UDP-ja nam Iperf vrne podatke o pretočnosti (troughput) in količini izgubljenih paketov (packet loss). Če se odločimo za uporabo TCP protokola, pa zaradi načina delovanja nimamo izgubljenih paketov, tako da dobimo samo podatek o pretočnosti. V obeh primerih dobimo časovno razdelano poročilo glede na opazovani interval, ki ga nastavimo pred testiranjem. Poročilo je sestavljeno iz zapisov, ki vsebujejo čas, količino podatkov, ki se je prenesla in pretočnost.

Za testiranje naše povezave, Iperf generira breme glede na podane argumente. Argument s katerim se generira breme pri UDP, je hitrost pošiljanja paketov [bits/sec] in čas pošiljanja. Namesto časa lahko specificiramo količino prenešenih podatkov. Pri TCP lahko navedemo maksimalno hitrost s katero želimo pošiljati, vendar zaradi delovanja protokola to ni potrebno, saj sam določa kakšna je največja hitrost pri kateri ne bo izgubljal paketov. Potrebno pa je navesti, koliko časa bo trajal prenos, oziroma koliko podatkov želimo prenesti. Kot argumente lahko navedemo še ali je povezava enostranska ali obojestranska, ter kakšna je velikost bralno-pisalnega medpomnilnika. Dobro je podati še argument, s katerim določimo interval med zapisi v končno poročilo, saj s tem vidimo, do kakšnih nihanj prihaja med celotnim prenosom. Program nam omogoča uporabo še veliko drugih parametrov [95], ki pa za nas niso pomembni.

9.2.2 Nadziranje in sledenje testiranju

Nadziranje in sledenje testiranju je zelo pomembno, saj z njim pravočasno zaznamo morebitne napake ali neželena odstopanja. Potrebno je poskrbeti, da se poraba virov v času testiranja ne bo povečala zaradi sledenja, oziroma bodo vplivi minimalni.

Odločili smo se, da bomo med samim poganjanjem programa Iperf sistem nadzirali s pomočjo orodji nicstat [96] in htop [97]. Z orodjem nicstat bomo opazovali, kaj se dogaja z mrežno kartico, natančneje, koliko paketov je bilo prejetih in kakšna je hitrost. Prav tako nas bo zanimala povprečna velikost

prejetih paketov. Z orodjem htop bomo nadzorovali odziv celotnega sistema na testiranje. S tem mislimo na porabo pomnilnika, obremenitev procesorja, itd.

Za podrobnejši nadzor sistema med izvajanjem benchmarkov bomo uporabili nadzorni sistem **Munin** [98], ki je prostodostopni paket na Linux distribucijah. Omogoča enostavno nadzorovanje sistema, omrežja in aplikacij. Zajete podatke hrani v RRD datotekah, kar omogoča pregled delovanja sistema skozi čas. To je možno preko enostavnega spletnega vmesnika, ki vsebuje grafe meritev na različnih časovnih ravneh.

Munin ima *master-node* arhitekturo kar omogoča minimalno sled na nadzorovanem sistemu in omogoča vzporeden nadzor večjega števila sistemov.

Munin-Master je centralno mesto, kjer se izvajajo vsa opravila. Redno se izvajajo sinhroni klici do nadzorovanih *vozlišč*, od koder se prenašajo vsi zbrani podatki.

Na tem mestu se generirajo vsi grafi, kar zahteva določeno procesorsko moč. Da je obremenjenost sistema kar se da majhna, se to izvaja le na zahtevo uporabnika.

Munin-Node je enostaven agent, ki teče na nadzorovanem sistemu in na zahtevo komunicira z *master* vozliščem.

Munin-Plugin je izvršljiva datoteka, ki zbira podatke o lokalnem sistemu ali pridobiva podatke iz oddaljenega sistema preko *SNMP* protokola.

RRDtool [99] (*round-robin, database tool*) je primeren za obravnavo podatkov časovne narave. V našem primeru bodo to meritve, ki jih bo zajemalo orodje Munin. Podatki so shranjeni v krožnem pomnilniku, pri čemer zasedenost pomnilnika ostaja konstantna skozi čas.

9.2.3 Postavitev testnega sistema

Dostop do virtualne instance amazonovega oblaka *t2.micro* smo pridobili z registracijo na njihovi spletni strani.

Instance virtualnih računalnikov se upravlja preko *EC2 Management Console*, ki je zelo kompleksen spletni upravljalni sistem in omogoča nastavitev plačilnih računov, uporabniških dostopov, dovoljenj, upravljanje instanc računalnikov, mrežnih nastavitev, izenačevanje obremenitve in mnogo drugih funkcionalnosti. Med drugin tudi obveščanje o obremenitvi sistema in porabi denarnih sredstev.

Pred začetkom dela na sistemu, smo želeli spoznati vse potrebne funkcionalnost sistema za upravljanje z virtualnimi računalniki, kar nam je zaradi obsežnosti sistema vzelo kar nekaj časa. Nastavili smo več uporabniških računov z omejenimi dovoljenji, tako da so osebni podatki in podatki o plačilu vidni le lastniku računa. Nastavili smo tudi obveščanja v primeru prekoračene kvote tako, da bomo ob morebitni pomotni prekoračitvi pravočasno obveščeni.

Na voljo je 9 lokacij po celem svetu, kjer je mogoče postaviti instanco. Ker bomo preverjali zmogljvost sistema in ne delovanje spletne storitve, kjer bi se meril tudi dostopni čas strežnika, to na naše rezultate ne bi smelo vplivati. Ker pa smo morali izbrati lokacijo, smo se odločili za eno izmed EU območij.

Izbran sistem (*Amazon EC2 t2.micro*) poganja operacijski sistem Ubuntu. Na voljo ima 1GB naključnega pomnilnika in 8GB zunanjega pomnilnika SSD. Dostop je mogoč preko SSH protokola. Za dostop potrebuje vsak član skupine svoj privatni ključ. Da smo omogočili dostop do SSH in do nekateri drugih storitev, je bilo potrebno v požarnem zidu instance sprostiti vrata 22, 443, 80 in ICMP za potrebe pinga.

Namestitev benchmarkov

Namestitev programa **Geekbench** je še kar zahtevna, če upoštevamo vse prenose datotek. Prenesti datoteko na našo micro instanco namreč ni najlažje zaradi uporabniških pravic. Poleg tega je potrebna namestitev jedrnih 32-bitnih sistemskih knjižnic, da lahko Geekbench izvaja teste 32-bitnih operacij na 64-bitnem procesorju (64-bitni testi so plačljivi).

FIO namestimo z enostavnim ukazom v ukazno vrstico. Za zagon Fia je potrebna skripta, v kateri definiramo breme. Za naš testni primer smo določili, da mora izvršiti za 1 GB pisanj in branj.

Namestitev **Iperf**-a je prav tako trivialna. Za delovanje tega programa je potrebno v požarnem zidu omogočiti port 443 ter namestiti program na lasten računalnik. Za testni preizkus smo na virtualni instanci micro pognali ukaz iperf -443 -s, na našem domačem računalniku pa iperf -p 443 -c IP. V obeh ukazih -p določa številko porta. Na klientovi strani IP določa globalni IP naše virtualne instance.

Namestitev nadzornih programov

Z vnosom ukaza **htop** v ukazno vrstico se nam prikaže enostaven uporabniški vmesnik, ki prikazuje trenutno porabo procesorja, pomnilnika, itd. Poleg tega nam prikaže še ostale zagnane procese in porabo virov.

Namestimo ga z enim ukazom. Uporabljamo ga z vnosom **nictat** v ukazno vrstico. Po vnosu ukaza nam za vsako mrežno kartico prikaže statistiko o paketih (velikosti, število prejetih, število zavrženih, itd.), hitrosti prejemanja in pošiljanja.

Namestitev programa **Munin** [100] je najbolj kompleksna, saj zahteva veliko dodatnih nastavitev. Najprej je potrebno postaviti strežnik Apache in ga primerno nastaviti. Nato je potrebno spremeniti vsebino datotek, ki vsebujejo konfiguracije za program Munin. Za konec je potrebno odpreti še port 80 v požarnem zidu. Po končani namestitvi lahko iz poljubnega oddaljenega računalnika preko spletne strani opazujemo natančne grafe, ki prikazujejo rabo virov našega sistema.

System Information

	Xen HVM domU
Operating System	Ubuntu 14.04.1 LTS 3.13.0-44-generic x86_64
Model	Xen HVM domU
Processor	Intel Xeon E5-2670 v2 @ 2.50 GHz 1 processor
Processor ID	GenuineIntel Family 6 Model 62 Stepping 4
L1 Instruction Cache	32 KB
L1 Data Cache	32 KB
L2 Cache	256 KB
L3 Cache	25600 KB
Motherboard	N/A
BIOS	Xen 4.2.amazon
Memory	992 MB

Slika 9.1: Sistemske specifikacije testnega sistema.

9.3 Rezultati

9.3.1 Specifikacije testnega sistema

Preden začnemo s predstavitvijo rezultatov, bi podobnejše specificirali ciljni sistem. Delno specifikacijo nam je ponudil Geekbench (glej siko 9.1). V specifikaciji ni omenjen trdi disk, ker ga za CPE teste ne potrebujemo, vemo pa, da virtualna instanca razpolaga z 8 GB SSD.

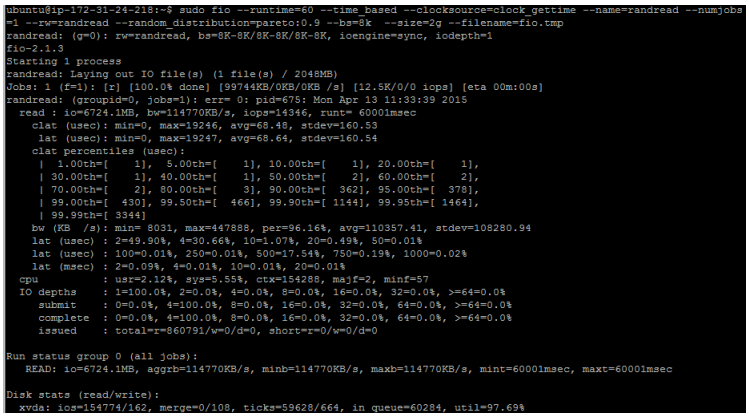
9.3.2 Trdi disk

Zmogljivost trdega diska smo testirali s programom fio (glej sliko 9.2). Pred začetkom testiranja smo ugotovili, da pravzaprav ne vemo kaj dosti o tem, kaj naj bi stalo za SSD diskom, ki nam ga obljublja Amazon. Prav tako smo ugotovili, da ne vemo niti kakšne teste naj pripravimo, da bi testirali določene lastnosti diskov niti katere lastnosti naj testiramo.

Po premisleku nam je ostala edina možnost, da poskusimo s testiranjem, kjer bi spreminjali vrednosti parametrov, za katere menimo, da nam bodo prinesli zanimive rezultate. Na podlagi ugotovitev iz rezultatov začetnih testiranj bomo postavili hipoteze, jih kasneje podrobneje raziskali in jih bodisi ovrgli bodisi potrdili.

Začetnemu testu smo določili, da se bo izvajal na eni niti, na kateri se bodo izvajale naključne bralne operacije blokov velikih 8kB. Skupno bosta prenešena 2GB podatkov. Naključna branja se bodo izvajala po funkciji pareto s potenco 0.9. Test bo trajal 60 sekund. Ukaz s katerim poženemo test z zgoraj omenjeno konfiguracijo:

fio –name= randread –numjobs= 1 –rw= randread –random_distribution= pareto:0.9 –bs= 8k –size= 2g –filename= fio.tmp



Slika 9.2: Prikaz rezultatov dobljenih s programom Fio.

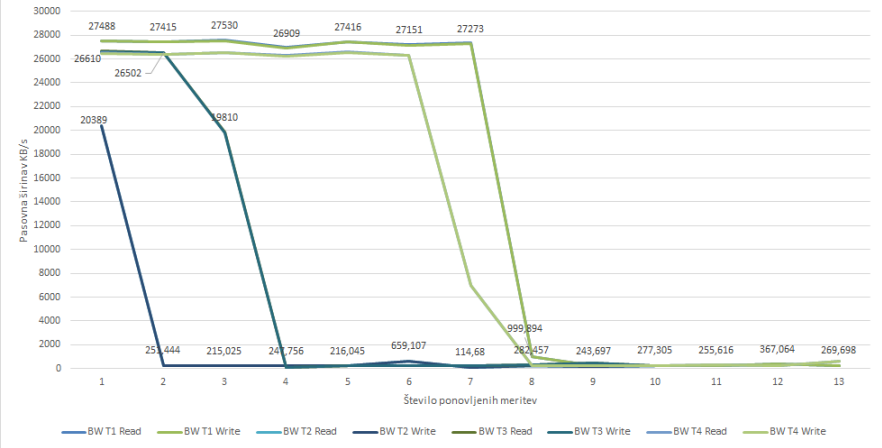
Fio nam vrne veliko različnih rezultatov. Omenili bomo le pomembnejše. Na začetku datoteke imamo navedene specifikacije izvajanja programa, ki jim sledi samo izvajanje. Na koncu je predstavitev rezultatov. Rezultati se začnejo z imenom testa, ki je v našem primeru randread. Sledi nekaj podatkov o sami izvedbi (pid procesa, morebitne napake, čas začetka testa, ...). Nato so tu podatki, ki so za nas relevantni. Začnejo se v sekciji read. V tem testu imamo samo to sekcijo, saj pisanj na disk nismo izvajali. Oznaka io nam pove, koliko bralnih operacij je test izvedel (6724,1 MB). Temu sledi pasovna širina prenosa (114,77 MB/s). Naslednja je glavna metrika trdih diskov. Imenuje se **input/output opreations per second (IOPS)**. Naš sistem je dobil oceno 14346 iops. Če ne bi vedeli kateri disk imamo, bi lahko s pomočjo lastnosti iops in bandwidth ugotovili, da se v našem opazovanem sistemu uporablja SSD in ne klasični HDD. Kot zadnjo pomembno lastnost imamo še runt, ki nam pove koliko časa se je test izvajal, lastnosti clat in lat, pa nam merita čas od oddaje posamezne zahteve za branje, do njene izvršitve. Nadaljnji zapisi vsebujejo predvsem podrobnejše predstavitve prejšnjih rezultatov.

TESTIRANJE 1

Sedaj ko razumemo, kako konfigurirati teste in kaj nam ti vračajo, se lahko lotimo testiranja opazovanega sistema. Ker na začetku nismo vedeli do kakšnih izsledkov nas bodo testi pripeljali, smo se odločili, da bomo naredili več testov z različnimi spremenljivkami. Teste smo najprej razdelili na tri dele, glede na način, kako izvršujejo bralno/pisalne operacije. Prvi način je bil modificiran random, ki za iskanje naslednjega bloka uporablja funkcijo pareto. Drugi način je bil uniformiran random. Zadnji pa je navadni sekvenčni izbor blokov. Znotraj vsakega izmed teh načinov smo izvedli še dodatno delitev, glede na velikost bloka za pisanje/branje. Izbrali smo si 4 različne velikosti blokov: 2KB, 4KB, 8KB in 16KB. Pri vseh testih se bodo izvajale bralno/pisalne operacije. Vse različice testov smo ponovili štrikrat. Teste smo izvajali v dveh delih. Pred izvajanjem testov smo sistemu naložili nekaj ogrevalnega bremena.

pareto									
time stamp	2K				time stamp	4K			
	BW [KB/s]		IOPS			BW [KB/s]		IOPS	
	read	write	read	write		read	write	read	write
29.4.2015 20:19	42,781	42,815	20	20	29.4.2015 20:26	89,981	89,558	21	21
29.4.2015 22:21	35,42	35,571	17	17	29.4.2015 22:11	85,232	85,369	20	20
30.4.2015 21:29	40,086	39,922	19	19	30.4.2015 21:33	88,997	88,916	21	21
30.4.2015 22:29	25,382	24,297	12	11	30.4.2015 22:33	106,864	107,847	26	26
time stamp	8K				time stamp	16K			
	BW [KB/s]		IOPS			BW [KB/s]		IOPS	
	read	write	read	write		read	write	read	write
29.4.2015 20:31	754,323	755,249	92	92	29.4.2015 20:45	0,02	FAIL	0	FAIL
29.4.2015 22:06	64,337	61,215	7	7	29.4.2015 22:01	812,518	812,681	49	49
30.4.2015 21:37	1008,9	1007,7	126	125	30.4.2015 21:50	0,021	FAIL	0	FAIL
30.4.2015 22:37	131,163	128,303	16	15	30.4.2015 22:40	796,37	794,248	48	48
uniform random									
time stamp	2K				time stamp	4K			
	BW [KB/s]		IOPS			BW [KB/s]		IOPS	
	read	write	read	write		read	write	read	write
29.4.2015 21:09	22,341	21,886	10	10	29.4.2015 21:03	57,498	57,417	14	14
29.4.2015 22:26	22,807	22,343	11	10	29.4.2015 22:34	198,059	198,087	48	48
30.4.2015 22:07	25,872	24,932	12	12	30.4.2015 22:03	53,43	51,815	13	13
30.4.2015 22:59	26,929	26,058	13	12	30.4.2015 22:55	0,014	FAIL	0	FAIL
time stamp	8K				time stamp	16K			
	BW [KB/s]		IOPS			BW [KB/s]		IOPS	
	read	write	read	write		read	write	read	write
29.4.2015 20:58	169,287	169,395	20	20	29.4.2015 20:52	402,169	402,931	24	24
29.4.2015 23:09	188,227	188,008	22	22	29.4.2015 23:14	225,116	224,68	13	13
30.4.2015 21:59	126,963	123,809	15	15	30.4.2015 21:56	538,935	538,935	32	32
30.4.2015 22:50	420,707	418,504	51	51	30.4.2015 22:44	153,278	147,095	9	8
sequential									
time stamp	2K				time stamp	4K			
	BW [KB/s]		IOPS			BW [KB/s]		IOPS	
	read	write	read	write		read	write	read	write
29.4.2015 21:15	216,256	216,1	105	105	29.4.2015 21:20	147,944	147,195	36	35
29.4.2015 23:38	205,16	205,385	100	100	29.4.2015 23:32	142,191	141,15	34	34
30.4.2015 22:10	188,252	188,979	91	92	30.4.2015 22:14	152,742	153,149	37	37
30.4.2015 23:04	183,627	193,493	89	89	30.4.2015 23:07	152,882	153,071	37	37
time stamp	8K				time stamp	16K			
	BW [KB/s]		IOPS			BW [KB/s]		IOPS	
	read	write	read	write		read	write	read	write
29.4.2015 21:39	751,21	751,891	91	91	29.4.2015 21:45	336,333	337,802	20	20
29.4.2015 23:27	278,897	277,172	34	34	29.4.2015 23:20	337,999	339,474	20	20
30.4.2015 22:21	248,975	249,876	30	30	30.4.2015 22:25	296,33	290,926	18	17
30.4.2015 23:11	208,024	207,983	25	25	30.4.2015 23:20	737,997	736,441	45	44

Slika 9.3: Prikaz rezultatov prvega testiranja.



Slika 9.4: Prikaz rezultatov pasovne širine v KB/s.

REZULTATI 1

Rezultati (9.3) so nas zelo presenetili. Glede na to, da naj bi bili v našem sistemu uporabljen SSD, bi morala biti pasovna širina in IOPS vsaj 100-krat višja od pridobljenih rezultatov. Po natančnejši analizi smo ugotovili, da smo z ogrevalnim bremenov pokvarili naše meritve. Med izvajanjem ogrevalnega bremena nam je fio namreč vrnil rezultate, ki smo jih pričakovali in opisali v predstavitvi testiranja. Glede na dobljene rezultate smo določili hipotezo za nadaljnje testiranje.

HIPOTEZA 2

Ob prekomerni obremenitvi datotečnega sistema, nam ponudnik zmanjša razpoložljive vire oziroma nam omeji hitrost dostopa do datotečnega sistema.

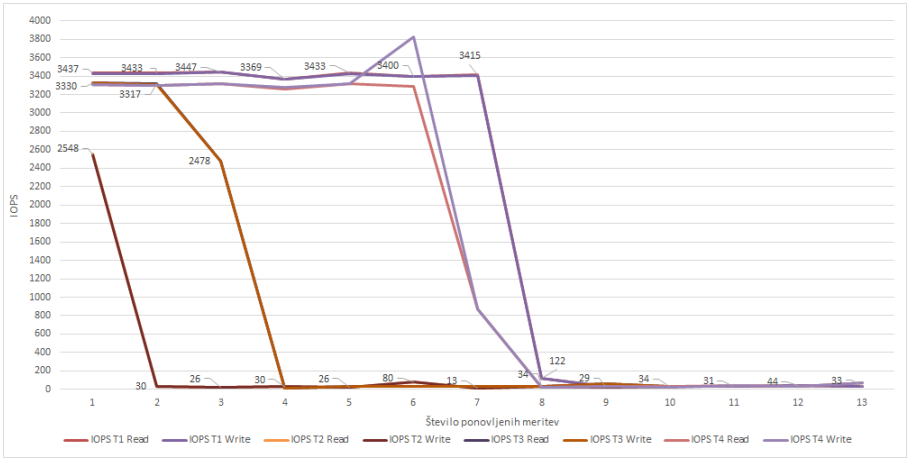
TESTIRANJE 2

Za potrditev naše hipoteze smo oblikovali naslednji sklop testov. Izvajali smo sekvenčne pisalno-bralne operacije z velikostjo bloka 8KB. Za ta test smo se odločili zato, ker so med našim prejšnjim testiranjem te meritve najbolj nihale. Naredili smo 13 zaporednih meritev. Vsaka meritev je trajala dve minuti. Sklop 13 meritev smo izvedli štirikrat. Razen prvega in drugega testa so bili testi izvedeni v različnih dneh.

REZULTATI 2

Glede na rezultate (sliki 9.4 in 9.5) lahko našo hipotezo potrdimo, saj je razvidno, da po nekaj začetnih meritvah pasovna širina in IOPS padejo na zelo majhne vrednosti (okoli 300 KB/s), kar je za današnje razmere v računalništvu zelo malo. Testa, ki zelo odstopata sta BW T2 Read in BW T2 Write. Testa sta bila izvršena v istem dnevu kot testa BW T1 Read in BW T1 Write, vendar s časovnim zamikom sedmih ur.

Iz zgornjih rezultatov lahko opazimo tudi, da sta sliki (9.4 in 9.5) skoraj



Slika 9.5: Prikaz rezultatov IOPS v 4 testiranjih.

identični. To je bilo pričakovano, saj so to pokazali vsi predhodni testi. Prav tako sta v pasovna širina in število vhodno/izhodnih operacij na sekundo (IOPS) v teoriji odvisni ena od druge. Zanimivo se nam je zdelo, da so bile hitrosti vhodnih in izhodnih operacij skoraj vedno enake in z minimalnimi odstopanji. To je presenetljivo, saj sta ti dve hitrosti v praksi in v specifikacijah trdih diskov, ponavadi vedno različni.

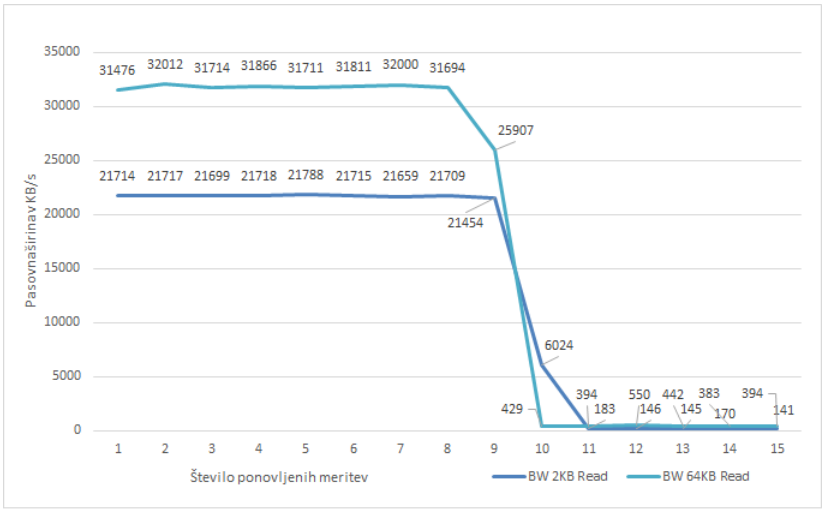
Glede na te rezultate smo postavili novo hipotezo:

HIPOTEZA 3

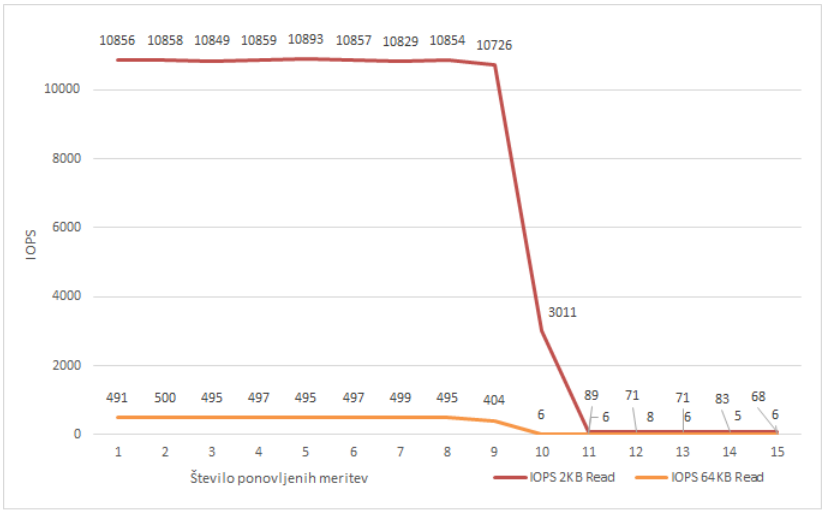
Velikost blokov za pisanje/branje vpliva na odvzem zmogljivosti datotečega sistema.

TESTIRANJE 3

Za potrditev te hipoteze smo izvedli podobne teste kot pri testiranju prejšnje hipoteze, le da smo spremenili velikosti blokov za pisalno-bralne operacije. Za potrebe testa smo izbrali dve skrajni vrednosti. Prva je predstavljala blok velikosti 2 KB, druga pa blok velikosti 64 KB. Posamezna meritev je trajala 2 minuti. Opravili smo 15 zaporednih meritev. V vseh meritvah smo izvajali le bralne operacije, saj smo ugotovili, da med bralnimi in pisalnimi operacijami ni večjih zmogljivostnih razlik.



Slika 9.6: Prikaz rezultatov pasovne širine v KB/s.



Slika 9.7: Prikaz rezultatov IOPS.

REZULTATI 3

Iz rezultatov 9.6 in 9.7 je razvidno, da začne zmogljivost padati po približno 10. meritvi. S padcem pasovne širine in IOPS smo morali hipotezo zavreči. Predvidevali smo, da nam Amazon ne bi zmanjšal zmogljivosti našega testnega sistema, če bi brali večje bloke, saj bi s tem imeli manj dostopov do diska. Iz slike 9.7 lahko opazimo, da smo pri 64 KB meritvah imeli veliko nižji IOPS kot pri 2 KB. Takšni rezultati so popolnoma normalni in ne pomenijo, da smo imeli že na začetku testiranja omejeno hitrost. Do tega pride, ker je za enako hitrost prenosa podatkov potrebno izdati precej manj ukazov za branje 64 KB blokov, kot za branje 2 KB. To ne pomeni, da bi lahko imeli veliko večji prenos podatkov, če bi želeli prebrati več podatkov na enkrat in bi bila tudi izvršitev takega branja daljša. Vendar glede na sliko 9.6 lahko vidimo, da so meritve s 64 KB bloki imele nekoliko višjo pasovno širino kot meritve, v katerih so se uporabljali 2 KB bloki.

Sedaj ko smo ugotovili, da velikost blokov ne vpliva na odvzem zmogljivosti, smo se vprašali, kakšna mora biti obremenitev sistema, da ta še vedno deluje s polno zmogljivostjo.

HIPOTEZA 4

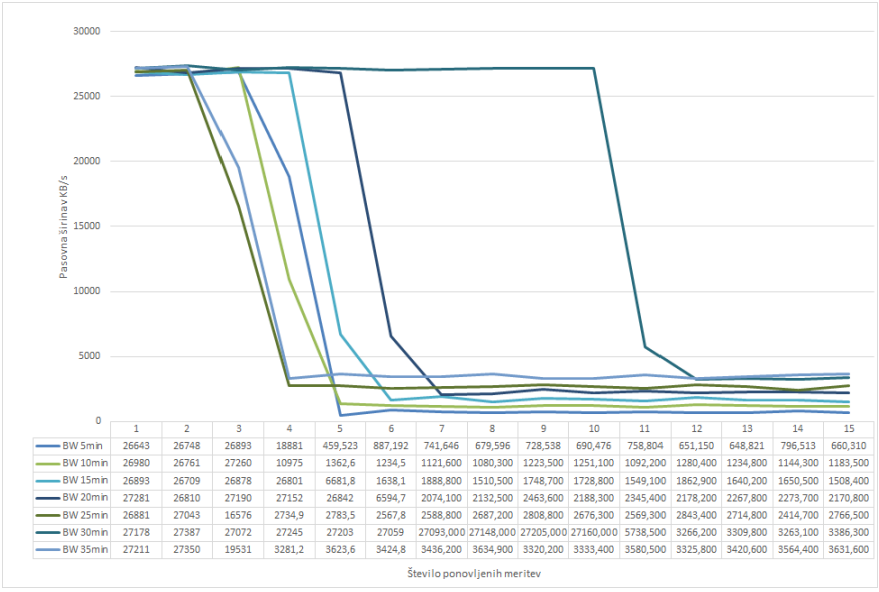
Daljši premori med posameznimi meritvami vplivajo na zmogljivost datotečnega sistema.

TESTIRANJE 4

Hipotezo smo testirali s pomočjo enakih testov, kot pri testiranju 2. hipoteze, le da smo po vsaki meritvi naredli določen premor. Zaradi narave našega testiranja ni mogoče narediti več od enega testa dnevno. Opravili smo 7 testiranj. Začeli smo s 5 minutnim premorom med meritvami, z vsakim nadaljnjim testom pa smo premoru dodali 5 minut. Želeli smo testirati še z daljšimi premori, vendar se je naš zadnji test, zaradi večanja premora med meritvami , izvajal že več kot 9 ur.

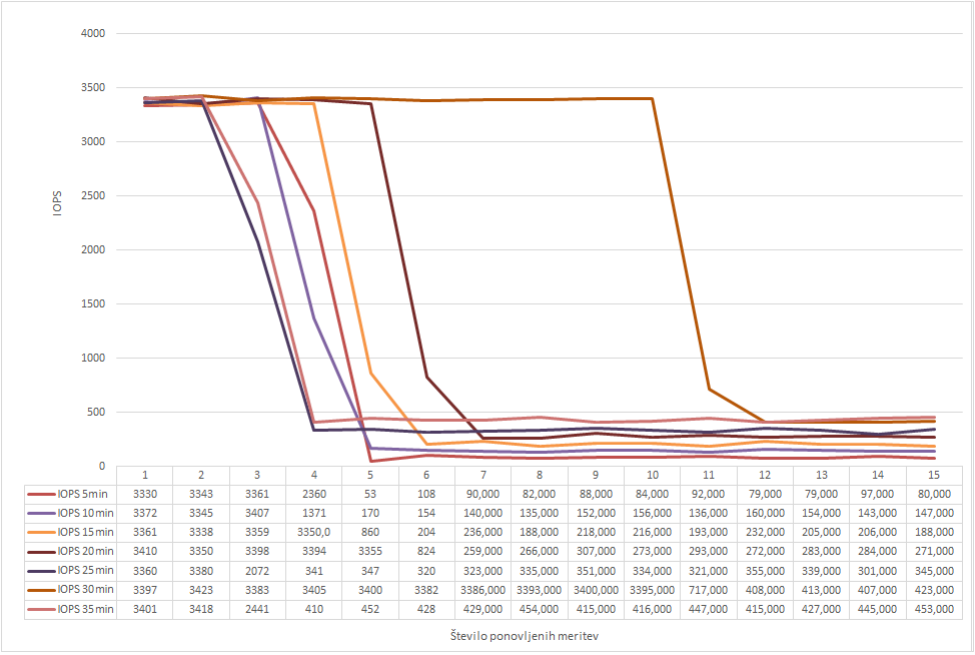
REZULTATI 4

Iz slik 9.8 in 9.9 lahko razberemo, kako premori med meritvami vplivajo na zmogljivost datotečnega sistema in s tem potrdimo hipotezo. Če se osredotočimo samo na sliko pasovne širine (9.8), lahko opazimo, kako pasovna širina raste glede na dolžino premorov med testi. Glede na podatke v tabeli pod sliko 9.8 lahko vidimo, da za vsakih dodatnih 5 minut premora pasovna širina poveča za približno 500 KB/s. Izračunali smo, da bi morali za nemoteno delovanje med posameznimi meritvami čakati približno 5 ur. Te predpostavke z meritvami ne bomo dokazovali, saj za to nimamo dovolj časa. Iz slik 9.8 in 9.9 smo razbrali še, da je test s čakalno dobo 30 minut potreboval veliko več časa do zmanjšanja zmogljivosti. Po daljšem premisleku smo ugotovili, da zato ni bila kriva dolžina premora med meritvami, ampak dolžina premora med izvajanjem posameznih testov (sklop 15 meritev). Teste smo izvajali vsak dan, razen testa s 30-minutno čakalno dobo, pred katerim je minilo več dni zaradi naše odsotnosti. Naše ugotovitve je potrdil tudi test s 35-minutno čakalno dobo, pri katerem se je



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
BW 5min	26643	26748	26893	18881	459,523	887,192	741,646	679,596	728,538	690,476	758,804	651,150	648,821	796,513	660,310
BW 10min	26980	26761	27260	10975	1362,6	1234,5	1121,600	1080,300	1223,500	1251,100	1092,200	1280,400	1234,800	1144,300	1183,500
BW 15min	26893	26709	26878	26801	6681,8	1638,1	1888,800	1510,500	1748,700	1728,800	1549,100	1862,900	1640,200	1650,500	1508,400
BW 20min	27281	26810	27190	27152	26842	6594,7	2074,100	2132,500	2463,600	2188,300	2345,400	2178,200	2267,800	2273,700	2170,800
BW 25min	26881	27043	16576	2734,9	2783,5	2567,8	2588,800	2687,200	2808,800	2676,300	2569,300	2843,400	2714,800	2414,700	2766,500
BW 30min	27178	27387	27072	27245	27203	27059	27093,000	27148,000	27205,000	27160,000	5738,500	3266,200	3309,800	3263,100	3386,300
BW 35min	27211	27350	19531	3281,2	3623,6	3424,8	3436,200	3634,900	3320,200	3333,400	3580,500	3325,800	3420,600	3564,400	3631,600

Slika 9.8: Prikaz rezultatov pasovne širine v KB/s.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
IOPS 5min	3330	3343	3361	2360	53	108	90,000	82,000	88,000	84,000	92,000	79,000	79,000	97,000	80,000
IOPS 10min	3372	3345	3407	1371	170	154	140,000	135,000	152,000	156,000	136,000	160,000	154,000	143,000	147,000
IOPS 15min	3361	3338	3359	3350,0	860	204	236,000	188,000	218,000	216,000	193,000	232,000	205,000	206,000	188,000
IOPS 20min	3410	3350	3398	3394	3355	824	259,000	266,000	307,000	273,000	293,000	272,000	283,000	284,000	271,000
IOPS 25min	3360	3380	2072	341	347	320	323,000	335,000	351,000	334,000	321,000	355,000	339,000	301,000	345,000
IOPS 30min	3397	3423	3383	3405	3400	3382	3386,000	3393,000	3400,000	3395,000	717,000	408,000	413,000	407,000	423,000
IOPS 35min	3401	3418	2441	410	452	428	429,000	454,000	415,000	416,000	447,000	415,000	427,000	445,000	453,000

Slika 9.9: Prikaz rezultatov IOPS.

zmogljivost zmanjšala že po prvih štirih testih.

9.4 Zaključek

Preden smo se lotili ocenjevanja zmogljivosti Amazonove oblačne storitve, si nismo niti približno predstavljali, kakšne bodo naše ugotovitve. Po končanem testiranju smo bili pozitivno presenečeni nad našimi ugotovitvami. Na začetku smo se preko različnih benchmark orodij dokopali do zanimivih pomanjkljivosti, ki smo jih odločili podrobneje raziskati. Odkrili smo, da nam ponudnik oblačnih storitev (Amazon) ob prekomernem obremenjevanju datotečnega sistema omeji (zmanjša) njihovo zmogljivost.

Ko smo to potrdili, nas je zanimal še glavni razlog, zakaj nas Amazon omejuje. Ker smo želeli to raziskati, smo pognali množico testov, ki so se osredotočali na velikost blokov za pisanje in branje in na intenzivnost testiranja. Najprej smo testirali ali velikost blokov vpliva na odvzem zmogljivosti ali ne. Predpostavka se je izkazala za napačno, saj smo pri blokih dolžine 2 KB in 64 KB dobili enake rezultate.

Ker smo ugotovili, da velikosti blokov niso vzrok za omejitev zmogljivosti dostopa, smo vzrok iskali v dolžinah premorov med ponovitvami testov. Po dolgem testiranju smo prišli do ugotovitev, da se z daljšimi premori med testi ne izognemo omejitvi zmogljivosti. Ker pa hitrost vseeno raste s dolžino premora, smo izračunali, da bi za nemoteno delovanje potrebovali premor dolg okoli 5 ur.

Po končanem testiranju in analizi rezultatov smo prišli do zaključka, da nas ponudnik omejuje glede na količino zapisanih podatkov na disk v določenem časovnem obdobju.

Literatura

[1] "Digital Ocean." https://www.digitalocean.com/, April 2015.

[2] "Amazon AWS." http://aws.amazon.com/, April 2015.

[3] "Github Education Pack." https://education.github.com/pack, April 2015.

[4] "AWS Amazon EC2 Pricing." http://aws.amazon.com/ec2/pricing/, April 2015.

[5] "NodeJS." https://nodejs.org/, April 2015.

[6] "Google V8." https://code.google.com/p/v8/, April 2015.

[7] "Node Package Manager (NPM)." https://www.npmjs.com/, April 2015.

[8] "MongoDB." https://www.mongodb.org/, April 2015.

[9] "GeoIP-lite library." https://www.npmjs.com/package/geoip-lite, April 2015.

[10] "MPI for python." http://mpi4py.scipy.org/, April 2015.

[11] "StarCluster." http://star.mit.edu/cluster/, April 2015.

[12] "Iaas." http://en.wikipedia.org/wiki/Cloud_computing#Infrastructure_as_a_service_.28IaaS.29.

[13] "Amazon web services." https://aws.amazon.com/.

[14] "Rasberry pi b+." http://en.wikipedia.org/wiki/Raspberry_Pi.

[15] "Java se 7." http://docs.oracle.com/javase/7/docs/api/java/net/Socket.html.

[16] "Bitbucket." https://bitbucket.org/UNIjostrovtar/monitoring-system.

[17] "Whetstone benchmark." http://en.wikipedia.org/wiki/Whetstone_%28benchmark%29.

[18] "Geekbench 3 benchmark." http://www.primatelabs.com/geekbench/.

[19] "Unixbench." https://github.com/kdlucas/byte-unixbench.

[20] "Big data." http://en.wikipedia.org/wiki/Big_data.

[21] "Cloud storage." http://users.abo.fi/slafond/theses/thesis_Joakim.pdf.

[22] "Numa." http://en.wikipedia.org/wiki/Non-uniform_memory_access.

[23] "Nas." http://en.wikipedia.org/wiki/Network-attached_storage.

[24] "Smb." http://en.wikipedia.org/wiki/Server_Message_Block.

[25] "Nfs." http://en.wikipedia.org/wiki/Network_File_System.

[26] "San." http://en.wikipedia.org/wiki/Storage_area_network.

[27] "Nearline storage." http://en.wikipedia.org/wiki/Nearline_storage.

[28] "Sysbench." https://launchpad.net/sysbench.

[29] "Digitalocean." https://www.digitalocean.com. Accessed: 2015-04-05.

[30] "Phoronix test suite." http://www.phoronix-test-suite.com/. Accessed: 2015-04-05.

[31] "Phoronix test suite." http://phoromatic.com/. Accessed: 2015-04-05.

[32] "Openbenchmarking." http://openbenchmarking.org/. Accessed: 2015-04-05.

[33] "Openssl." https://www.openssl.org/source. Accessed: 2015-04-13.

[34] "Gcc optimizacije." https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Accessed: 2015-04-13.

[35] "Clang optimizacije - stackoverflow." http://stackoverflow.com/questions/15548023/clang-optimization-levels. Accessed: 2015-04-13.

[36] "Gcc optimizacije gentoo." https://wiki.gentoo.org/wiki/GCC_optimization. Accessed: 2015-04-13.

[37] "Clang freebsd manual." https://www.freebsd.org/cgi/man.cgi?query=clang&apropos=0&sektion=1&manpath=FreeBSD+10.1-stable&arch=default&format=html. Accessed: 2015-04-13.

[38] "N-queens problem." http://mathworld.wolfram.com/QueensProblem.html. Accessed: 2015-05-10.

[39] "Ext4 file-system tuning benchmarks." http://www.phoronix.com/scan.php?page=article&item=ext4_linux35_tuning&num=1. Accessed: 2015-05-10.

[40] "Phoronix - tuning ext4." http://www.phoronix.com/scan.php?page=article&item=ext4_linux35_tuning&num=1. Accessed: 2015-04-13.

[41] "Ext4 manual." https://www.kernel.org/doc/Documentation/filesystems/ext4.txt. Accessed: 2015-04-13.

[42] "Tunefs." https://www.freebsd.org/cgi/man.cgi?query=tunefs&sektion=8&n=1. Accessed: 2015-04-13.

[43] "Configuring disks - freebsd manual." https://www.freebsd.org/doc/en/books/handbook/configtuning-disk.html. Accessed: 2015-04-13.

[44] "Threaded io test." http://openbenchmarking.org/test/pts/tiobench. Accessed: 2015-04-05.

[45] "Dd." https://www.freebsd.org/cgi/man.cgi?query=dd%281%29&sektion=. Accessed: 2015-04-13.

[46] "Fio github." https://github.com/axboe/fio. Accessed: 2015-04-13.

[47] "Fio github." http://www.iozone.org. Accessed: 2015-04-13.

[48] "Blogbench." http://openbenchmarking.org/test/pts/blogbench. Accessed: 2015-04-05.

[49] "Phpbench." http://openbenchmarking.org/test/pts/phpbench. Accessed: 2015-05-10.

[50] "ab - apache http server benchmarking tool." http://httpd.apache.org/docs/2.2/programs/ab.html. Accessed: 2015-05-10.

[51] "Iperf." https://iperf.fr. Accessed: 2015-04-13.

[52] "Gcc wikipedia." http://en.wikipedia.org/wiki/GNU_Compiler_Collection. Accessed: 2015-05-10.

[53] "Clang wikipedia." http://en.wikipedia.org/wiki/Clang. Accessed: 2015-05-10.

[54] "Ufs wikipedia." http://en.wikipedia.org/wiki/Unix_File_System. Accessed: 2015-05-10.

[55] "Ext4 wikipedia." http://en.wikipedia.org/wiki/Ext4. Accessed: 2015-05-10.

[56] "Freebsd donors." https://www.freebsdfoundation.org/donate/sponsors. Accessed: 2015-05-10.

[57] J. Read, "Cloud scores," Mar. 2015. [Online; accessed 29-Mar-2015].

[58] "Mpich," Mar. 2015. [Online; accessed 29-Mar-2015].

[59] P. S. Cox, "Steps to make raspberry pi supercomputer," Mar. 2015. [Online; accessed 29-Mar-2015].

[60] E. de Castro Lopo, "Libsnfile," Apr. 2015. [Online; accessed 06-Apr-2015].

[61] "CodeSys - The PLC Benchmark." http://www.codesys.com/index.php?eID=tx_nawsecuredl&u=0&g=0&t=1431377989&hash=dbde25c580c4282f454e33657f2363f834980c99&file=fileadmin/Download/Publikationen/SD_Neufeld_atp_i_1_2007.pdf, maj 2015.

[62] "Wikipedia - Programmable logic controller." http://en.wikipedia.org/wiki/Programmable_logic_controller, maj 2015.

[63] "emVIEW-6T/A500 Hardware manual." https://www.janztec.com/fileadmin/downloads/manuals/emVIEW/manual_emview-6T-A500_hardware.pdf, maj 2015.

[64] "Structured Text (ST)." http://www402.abbext.com/DownloadFile.aspx?file=UserFiles/Answers201210/ST_manual.pdf&filename=ST_manual.pdf, maj 2015.

[65] "CodeSys Store - PLC Chat." http://store.codesys.com/plc-chat.html, april 2015.

[66] "Task Force - Benchmarking." http://www.plcopen.org/pages/tc3_certification/downloads/tc3_benchmarking_v04.pdf, maj 2015.

[67] "Distance Calculator." http://www.daftlogic.com/projects-google-maps-distance-calculator.htm, maj 2015.

[68] "Wikipedia - TCP tuning." http://en.wikipedia.org/wiki/TCP_tuning, maj 2015.

[69] Amazon, "Amazon S3 service." http://aws.amazon.com/s3/, 2015. [Dostopano dne 29. 4. 2015].

[70] Amazon, "Aws free tier." http://aws.amazon.com/free/, 2015. [Dostopano dne 29. 4. 2015].

[71] "Slika bremena (iceberg)." http://www.ivanbrooker.com/wallpaper/mac-os-x-snow-leopard-iceberg-wallpaper.html. [Dostopano dne 29. 4. 2015].

[72] "Platforma Open Shift." https://www.openshift.com//, Marec 2015.

[73] "Cloud benchmarks." https://cloudharmony.com/benchmarks, Mar. 2015.

[74] "Benchmarking personal cloud storage." http://eprints.eemcs.utwente.nl/23674/01/cloud_storage.pdf, Mar. 2015.

[75] "Io benchmarking: How, why and with what." http://www.cuddletech.com/blog/pivot/entry.php?id=820, Mar. 2015.

[76] "Openshift." https://www.openshift.com/, Mar. 2015.

[77] "A clue how openshift really works." https://www.ibm.com/developerworks/community/blogs/c06ef551-0127-483d-a104-cdd02b1cee31/entry/a_clue_how_openshift_really_works_and_my_surprising_conclusion19?lang=en, Apr. 2015.

[78] "Php 5 tutorial." http://www.w3schools.com/php/, Apr. 2015.

[79] "Php mysql database." http://www.w3schools.com/php/php_mysql_intro.asp, Apr. 2015.

[80] "Java platform standard edition 8 documentation." http://docs.oracle.com/javase/8/docs/, Apr. 2015.

[81] "Google sheets." https://www.google.com/sheets/about, Apr. 2015.

[82] "Network monitor." https://developer.mozilla.org/en-US/docs/Tools/Network_Monitor, Apr. 2015.

[83] "Openshift faq." https://developers.openshift.com/en/getting-started-overview.html, Mar. 2015.

[84] "Longblob." https://mariadb.com/kb/en/mariadb/longblob, Mar. 2015.

[85] J. Gao, X. Bai, and W.-T. Tsai, "Cloud testing-issues, challenges, needs and practice," *Software Engineering: An International Journal*, vol. 1, no. 1, pp. 9–23, 2011.

[86] "Cloud harmony benchmark scores." https://cloudharmony.com/benchmarks, May 2015.

[87] I. TechEmpower, "Web framework benchmarks." https://www.techempower.com/benchmarks/, Mar. 2015.

[88] B. Gregg, "Benchmarking the cloud - active and passive benchmarking." http://www.joyent.com/blog/benchmarking-the-cloud, May 2015.

[89] "Live benchmarks from the cloud." http://www.cloudlook.com/, May 2015.

[90] R. Longbottom, "Dhrystone." http://www.roylongbottom.org.uk/dhrystone%20results.htm, Feb. 2015.

[91] R. Longbottom, "Whetstone." http://www.roylongbottom.org.uk/whetstone.htm, Oct. 2014.

[92] P. John D. McCalpin, "Stream benchmark." http://www.admin-magazine.com/HPC/Articles/Finding-Memory-Bottlenecks-with-Stream, Mar. 2015.

[93] A. Carroll, "fio man." http://linux.die.net/man/1/fio, Dec. 2014.

[94] T. B. of Trustees of the University of Illinois, "What is iperf." https://iperf.fr/, Mar. 2015.

[95] F. of Science University of Amsterdam, "Iperf manual." https://staff.science.uva.nl/j.blom/gigaport/tools/man/iperf.html, Mar. 2015.

[96] T. Cook, "Nicstat." http://sourceforge.net/projects/nicstat/files/, Mar. 2015.

[97] H. Muhammad, "htop." http://hisham.hm/htop/index.php?page=main, Mar. 2015.

[98] "Munin monitoring." http://munin-monitoring.org, Mar. 2015.

[99] T. Oetiker, "Rrdtool." http://oss.oetiker.ch/rrdtool, Mar. 2015.

[100] E. Sverdlov and J. Ellingwood, "How to install munin on an ubuntu vps." https://www.digitalocean.com/community/tutorials/how-to-install-munin-on-an-ubuntu-vps, June 2013.